CoCa: a model for parallelization of high energy physics software

Erco Argante
CoCa: a model for parallelization of high energy physics software

PROEFONTWERP

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 7 oktober 1998 om 16.00 uur

door

Erco Argante

egenomen te Nijmegen
Dit proefontwerp is goedgekeurd door de promotoren:

prof.dr. M. Rem
en
prof.dr.ir. K. Kopinga

Copromotor:
dr. P.D.V. van der Stok

Druk: Universiteitsdrukkerij Technische Universiteit Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN
Argante, Erco

CoCa: a model for parallelization of high energy physics software /
by Erco Argante. - Eindhoven: Eindhoven University of Technology,
1998.
Proefontwerp. - ISBN 90-386-0601-X
Nugi859
Subject headings: parallel computer systems, high energy physics, software
systems
ACM: D.1.3, C.1.2, J.2, H.2.4
Contents

Chapter 1: Introduction .............................................. 1

Chapter 2: Parallel computing ....................................... 5

2.1 Introduction .................................................. 5

2.2 Parallel computers ............................................. 6
  2.2.1 Consequences for software design ....................... 8
  2.2.2 Present trend in parallel computing ..................... 9

2.3 Software parallelization ....................................... 10

2.4 (Explicit) component level parallelization ................ 10
  2.4.1 Flavors of component level parallelization ............ 12
  2.4.2 Resource management and load balancing ............... 13
    2.4.2.1 Deviation from deterministic temporal behavior .. 17
  2.4.3 Mapping components to hardware ....................... 20
  2.4.4 Summary ................................................ 21

2.5 Inter-component communication ............................... 22
  2.5.1 Message passing ........................................ 22
  2.5.2 Remote procedure calls ................................ 23
  2.5.3 Distributed shared memory systems..................... 24
  2.5.4 Data dependencies between collections of components 25

Chapter 3: Databases ............................................... 27

3.1 Introduction ............................................... 27

3.2 Data modeling .............................................. 27

3.3 Data access ............................................... 29

3.4 Database design .......................................... 30
  3.4.1 Parallel database architectures ....................... 30
  3.4.2 Concurrency control .................................. 32

Chapter 4: Parallelism in high energy physics ................. 35
Chapter 5: A programming model for HEP: CoCa
5.1 Introduction. ..... 45
5.2 Requirements for the CoCa programming model. ..... 45
5.3 Requirements for the CoCa communication model ..... 47
5.4 Architecture of the CoCa communication model ..... 48
  5.4.1 Physical aspects of the job space ..... 50
Chapter 6: Evaluation of hardware and OS facilities
6.1 Introduction. ..... 55
6.2 Programming model offered by operating systems ..... 55
  6.2.1 Measurements ..... 57
  6.2.2 Algorithms ..... 58
  6.2.2.1 A 1: semaphores for waiting and mutexes for exclusion. ..... 59
  6.2.2.2 A 2: condition variables for waiting and a mutex for exclusion ..... 61
  6.2.2.3 A 3: busy waiting for waiting and a lock for exclusion ..... 62
  6.2.2.4 A 4: consumer busy waiting with boolean empty (version a). ..... 64
  6.2.2.5 A 5: consumer busy waiting with boolean empty (version b) ..... 65
  6.2.2.6 A 6: busy waiting with booleans empty and noaccess ..... 66
  6.2.2.7 Conclusion ..... 67
  6.2.2.8 The mutual exclusion algorithm ..... 69
6.3 Inter-node communication ..... 70
Chapter 7: Design
7.1 Introduction. ..... 73
7.2 The job space. ..... 74
  7.2.1 Shared CoCa. ..... 74
  7.2.2 Distributed CoCa ..... 77
7.3 Job space access. ..... 77
  7.3.1 Shared CoCa. ..... 77
  7.3.2 Distributed CoCa ..... 78
7.3.2.1 Location transparency layer ........................................ 78
7.3.2.2 CoCa communication layer ........................................ 84
7.3.2.3 CoCa physical layer .................................................. 86
7.4 Job space consistency ...................................................... 86
  7.4.1 Shared CoCa .......................................................... 86
  7.4.1.1 Concurrency atomicity ............................................. 86
  7.4.1.2 Correctness and fail atomicity ................................... 88
  7.4.2 Distributed CoCa ..................................................... 93
7.5 The components ............................................................ 94
  7.5.1 Shared CoCa .......................................................... 94
  7.5.2 Distributed CoCa ..................................................... 95
7.6 Recombination of related data .......................................... 95
  7.6.1 Shared CoCa .......................................................... 96
  7.6.2 Distributed CoCa ..................................................... 99
7.7 Host language bindings .................................................. 101
  7.7.1 Configuration information ........................................... 102
  7.7.2 C++ binding .......................................................... 103
  7.7.3 C binding ............................................................. 105
  7.7.4 Fortran 77 binding ................................................... 106

Chapter 8: Validation ......................................................... 109

8.1 Introduction ............................................................... 109
8.2 Performance of the CoCa primitives ................................... 109
8.3 Test case 1: CPREAD event farming .................................... 112
  8.3.1 The CPREAD event reconstruction program ..................... 113
  8.3.2 Coding phase: adaptations to CPREAD for event farming .... 113
  8.3.3 Configuration phase: CPREAD event farming on the SC2000 .. 114
  8.3.4 Tests: CPREAD event farming on the SC2000 ................... 115
  8.3.5 Configuration phase: CPREAD event farming on the CS-2. .... 118
  8.3.6 Tests: CPREAD event farming on the CS-2. ..................... 119
8.4 Test case 2: CPREAD parallel track fitting .......................... 121
  8.4.1 Coding phase: adaptations to CPREAD for parallel track fitting . 122
  8.4.2 Configuration phase: CPREAD parallel track fitting on the SC2000 . 123
  8.4.3 Tests: CPREAD parallel track fitting on the SC2000 .......... 124
  8.4.4 Configuration phase: CPREAD parallel track fitting on the CS-2. 126
  8.4.5 Tests: CPREAD parallel track fitting on the CS-2 ............ 126
8.5 C++ binding ............................................................... 127
8.6 Conclusion and future tests ............................................. 129
Chapter 9: Discussion and conclusions

9.1 Introduction .............................................. 131
9.2 Discussion of CoCa ....................................... 131
  9.2.1 Using CoCa for parallelizing HEP software .......... 131
  9.2.2 Comparison of CoCa with other models for parallelization 132
  9.2.3 Parallel Hardware .................................... 134
9.3 Evaluation of the design process .......................... 135
9.4 Conclusion ................................................. 138

Summary ..................................................... 139
Samenvatting ................................................. 140
Acknowledgments ............................................ 141
Curriculum vitae ............................................. 142
References ..................................................... 143
Index ............................................................ 147
Chapter 1: Introduction

The increasing complexity of High Energy Physics (HEP) experiments results in continuously increasing performance requirements. Advanced hardware and software of high complexity are required to meet these requirements. The HEP experiments of the future Large Hadron Collider [LHC95] that will take place at CERN, the European Laboratory for Particle Physics in Geneva, foresee data rates of 100 GBytes/s and have data storage requirements of 1 PetaByte/year [CMS96]. They require an increase of computing power of several orders of magnitude compared to today [Rubbia92]. Industry offers high performance parallel computing platforms, based on commodity processors connected by low latency and high bandwidth networks, that could meet the performance requirements. We can distinguish distributed memory computers which aim at scalability and shared memory computers which aim at ease of programming. Furthermore, hardware technology is changing rapidly, whereas the foreseen LHC software development time is 8 years and its lifetime 20 years. Consequently, three main requirements for software development are: high performance, hiding of complexity for the application programmer, and hardware independence to allow updating of hardware during the software lifetime and to obtain portability.

These requirements have been the input for the CoCa programming model that we propose. CoCa, which stands for Communication Capability, is a model to parallelize HEP software. By abstracting from the underlying communication structure, complexity is reduced, but high performance and scalability are preserved. The CoCa model takes the problems into account that are caused by characteristics of HEP software. CoCa is not specific to HEP only; its generality allows other domains with similar characteristics to benefit.

The CoCa model

CoCa has a parallelization model in which multiple components run in parallel. A component performs a task and consists of several lines of code. CoCa communication primitives, which are embedded in the component code, realize inter-component communication. CoCa uses a communication model based on a so-called job space. Not yet processed data residing in job space can be accessed by any component in the system. A component selects appropriate data from job space by specifying predicates. CoCa uses the job space model to hide the structure of the underlying hardware platform from the application programmer, which results in portable software. By separating the realization of a parallel application into a coding and a configuration phase, it can nevertheless take advantage of specific features of the underlying hardware, thereby retaining the performance. In the coding phase, the programmer of the parallel application inserts CoCa primitive statements in the code for inter-component communication. In the configuration phase, the number of components of one type, the component/node mapping, the data allocation and data distribution strategies are specified. The aim of the data distribution strategies is to allow a component to find appropriate data items locally. This avoids that a
CoCa primitive includes an inter-node communication which has a latency that is large with respect to the intra-node communication latency.

The chosen model for parallelization allows us to cope with characteristics particular to HEP applications, like instant rejection of data, variable data item execution times, and variations in the input rate. In the kind of HEP applications we focused on, most of the data are written and read once, while a part of the data are written once, but read multiple times. CoCa supports this by offering a single reader/single writer and multiple reader/single writer model.

During runtime of CoCa data will be distributed among the components. Some of these data might be related and have to be correctly recombined. CoCa supports this with its automatic recombination feature. As it is very difficult to foresee all communication patterns that can occur in parallel applications and as an attempt to do this might result in a large overhead, the automatic recombination feature of CoCa can be bypassed, and the application programmer can adapt the communication pattern to the specific needs of the given application.

CoCa provides language bindings that present the CoCa functionality to the application programmer, modeled according to the general philosophy of the host language. There is a Fortran, a C and a C++ binding. The C++ binding allows CoCa to be used with future HEP software of the future LHC accelerator, as the emerging trend at CERN is to design software according to the object oriented paradigm, in particular using C++ [Bos97]. The Fortran binding allows CoCa to be used with existing HEP applications, as they are usually written in Fortran.

**CoCa with respect to other programming models and systems for software parallelization**

In the HEP context, other programming models and systems have been proposed for software parallelization. To be able to discuss them, we introduce a so-called event which is the main unit of data that is produced by HEP experiments. Events can be independently processed. Event level parallelism aims at event throughput increase, while sub-event level parallelism aims at event latency decrease.

PIAF [Rademakers94] is a system that supports farming at the event level, which is a proven technique for event throughput increase. The Mona Lisa programming paradigm [Schneider93] allows sub-event parallelism, besides event level farming. It provides communication and synchronization by means of exposing/hiding of global variables. However, the low performance of Mona Lisa makes this model inappropriate for production environments; it can only be used for development and educational purposes [Schiefer95]. CoCa allows parallelism at the event and sub-event level and its low overhead allows relatively fine-grained parallelism with good performance.

Most of the existing HEP software is programmed using Zebra [CN94], a package that adds dynamic data structures to Fortran. Its low level programming interface does not support data structuring on an abstract level. This deficiency has been recognized and models have been proposed in the HEP context to fill the gap. ADAMO [Rousseau94] is a system, built on top of Zebra, that provides data structuring facilities based on the entity-relationship model. For future HEP experiments, it is hoped that data structuring can be obtained by applying object-oriented techniques. The CoCa job space model and its use of database techniques allow the structuring of data such that they are readily usable by the environment.

If we look further than the HEP context, the job space model of CoCa has some similarities to the Linda model for parallel programming [Bjornson87]. Linda is based on a tuple space which provides a very large expressive power, but its generality implies low performance. With respect to Linda, CoCa is less general as the types of the data items that can be communicated, are defined at compile time, while Linda allows data types to be created at runtime. However, this type information allows CoCa to confine the search space for appropriate data items,
allows data distribution decisions that are based on the data type and the knowledge about the data item size allows cheaper memory management. These issues improve the performance.

CoCa could also be compared with distributed shared memory models [Protic96]. Their aim is to provide ease of programming by providing distributed memory computers with the shared memory programming model. As the level of implementation of distributed shared memory models is usually very low, they can be implemented using low overhead primitives, but they cannot make use of the application program semantics to increase the performance. CoCa has a unit of communication that can be adapted to the given parallel application, and it allows typing of the communicated data items, as has been mentioned above.

PVM [Geist93] and MPI [MPI94] are well-known message passing libraries. The message passing paradigm allows communication between components running in parallel, but with a much stricter producer/consumer identification than provided by the CoCa job space model. This results in less flexibility than offered by CoCa. A performance comparison of CoCa with native versions of PVM and MPI on the distributed memory platform we used for our tests, showed that CoCa performs slightly better, despite its richer functionality. Note though that although CoCa is intended for use on any shared or distributed memory computer, PVM and MPI are even more general, as they are also intended for distributed computing applications. This means that their design might contain more compromises than CoCa’s design, which might explain a part of the observed performance difference.

CoCa design and implementation

The major design issue of the CoCa model has been its performance. The CoCa design integrates many modern software techniques. It combines parallel computing (multi-threading) and distributed computing with object-oriented methods and database technology. By recognizing that some of the features of CoCa resemble the properties of database transactions, for the design of CoCa we could make use of the well-established database theory. Modern processors in parallel computers provide the possibility to overlap communication and computation, which is exploited by CoCa.

CoCa is implemented as a layer that resides between the parallel application and the native communication library of the parallel computer. CoCa is implemented on a 20-node SUN SPARCcenter 2000 shared memory computer and on a MEIKO CS-2 distributed memory computer. The very different architectures of these machines allow us to verify the parallel applications’ portability that CoCa promised to provide.

Context of the Ph.D. thesis

The Ph.D. project has been carried out at CERN, in the context of a collaboration between CERN and the Eindhoven University of Technology in the Netherlands. This Ph.D. is a designers Ph.D. which is a relatively new phenomenon in the Netherlands. The emphasis is on synthesizing already existing theories into artifacts that satisfy the needs and criteria imposed by the environment, rather than on obtaining new knowledge [BCO94]. There is an emphasis on how this synthesis is accomplished, therefore not only the result, but also the way to the result is of importance. Consequently, this thesis contains a section describing the design process.

The first half of the Ph.D. period has been carried out in the CERN ECP division in the context of the ESPRIT projects P7255, GPMIMD-2, and P5404, GPMIMD. The second half has been carried out in the CMS-CMC group, responsible for the software of the future CMS experiment [CMS96].
Introduction

Content

In the next chapter about parallel computing, parallelism and software parallelization are introduced. It does not provide a general overview of parallel computing, but focuses on the type of parallelism applied in CoCa. This parallelism is put in a context and its limitations with respect to the potential performance increase are discussed. The chapter also mentions some existing paradigms for inter-component communication. In chapter 3, database theory applicable to the CoCa project is discussed. We draw from this knowledge when the CoCa design is discussed. In the chapter about High Energy Physics, chapter 4, the motivation for the CoCa model is provided. It starts with a description of HEP experiments. From this environment description an environment analysis is performed in which we determine which parts of HEP experiments require parallelism and where parallelism can be applied. Furthermore, the chapter discusses the characteristics of HEP software that motivated the CoCa design. In chapter 5, the CoCa programming model is derived from the observations from chapter 4. Before we can discuss the design of CoCa, we evaluate the performance of facilities offered by hardware and operating system on which the design will be based (chapter 6). In chapter 7, the design of CoCa is discussed. In chapter 8, the CoCa programming model and the CoCa design are validated in terms of performance, promised portability of the parallel application, and ease of use. This is done on two different hardware platforms by means of tests of CoCa and two test cases, using an existing HEP application. Chapter 9, the discussion, contains general conclusions and contains observations about the design process. It also compares the Ph.D. result with the promises and planning made in the original project proposal [Argante94].
Chapter 2: Parallel computing

2.1 Introduction

The performance of a computer system (hardware and software) can be characterized by its latency and throughput. If the input of the computer system consists of a stream of jobs, then the job latency is the time it takes to process one job, and the job throughput is the rate at which jobs can be processed. An increase of the job throughput with respect to the sequential processing of jobs is necessary, if the job processing time is larger than the average inter-job arrival time. A decrease of the job latency with respect to sequential job processing is necessary, if there are constraints on the job processing time. Examples of such constraints are: (1) the results of the job are used in a control feedback loop with a maximum response time, (2) the results of the job are used for immediate interactive examination, or (3) the buffer space for jobs that are being processed is limited. The performance of a computer can be enhanced by increasing the computing power of the hardware. This can be done by increasing the processor speed or by equipping a computer with multiple processors. Moore’s law, stating that the processor speed increases exponentially, while prices remain the same, will soon collapse, since the physical limits will be reached in the near future [Lewis96]. In 1965, the number of transistors on a chip doubled every 12 months, corresponding to a factor of 1000 every decade. In the mid 1970’s, doubling occurred every 18 months, corresponding to a factor of 100 every decade [Brenner97]. The 1994 SIA Road Map assumes a growth of a factor of 10 between 1997 and 2007 [SIA94]. This makes multi-processor computers, also called parallel computers, an attractive alternative. To exploit the power of a parallel computer efficiently, the execution load of an application needs to be distributed among the processors. This is the basic motivation for software parallelization.

This chapter does not pretend to give a general overview of parallel computing. It provides the information necessary to understand the description of the CoCa model in the following chapters. Not all the concepts introduced in this chapter are explicitly used in the following chapters, but serve to put the CoCa model in a context. The chapter starts with a description of different types of parallel computers (section 2.2). In section 2.3 software parallelization is discussed. One type of parallelization is component level parallelization in which the application consists of components that run in parallel. In section 2.4 different types of component level parallelization are examined. Then it is analyzed how to parallelize an application at the component level. Two steps can be distinguished:

1. Division of an application into components that can run in parallel (section 2.4.2). Subsequently, components need to be mapped to hardware (section 2.4.3).

2. Data have to be exchanged between the cooperating components of the parallel application. This is the focus of this thesis and therefore it will be addressed in subsequent chapters.
Section 2.5 starts the discussion about inter-component communication and includes a discussion on some well-known communication models.

2.2 Parallel computers

A parallel computer is a collection of processors (also called CPUs) and memory that can be directly or indirectly accessed by the processors. Flynn [Flynn66], [Flynn72] looked at parallelism in terms of the instruction and data streams and distinguishes four classes of computers of which two are feasible parallel computers: single-instruction-multiple-data (SIMD) and multiple-instruction-multiple-data (MIMD) computers. The processors of a SIMD computer are synchronized and execute the same instruction in a clock cycle on different data. This is called data parallelism and it makes SIMD computers only suitable for applications where data parallelism is possible. In a MIMD computer, each processor can run a different program on different data, and processors are not synchronized. This allows different instructions to be executed at the same time, called algorithmic parallelism. MIMD computers can accomplish a wide variety of computing tasks, and more different types of MIMD computers have been built than of SIMD computers (p. 573 of [Patterson90]). We restrict ourselves in this thesis to MIMD computers.

The MIMD class consists of two types of computers: shared and distributed memory computers. First the physical lay-out of both types of computers is discussed, followed by the differences from a programmer’s point of view. To quantify our statements, some figures are supplied about the two machines used in the thesis, a SUN SPARCcenter 2000 (SC2000) [SUN96] shared memory computer, situated at the Linköping University in Sweden, and a MEIKO CS-2 hybrid distributed memory computer, situated at CERN.

(a) shared memory computer
(b) distributed memory computer
(c) hybrid distributed memory computer

![Diagram](image)

FIGURE 1: Lay-out of parallel computer types. A dashed box is a node, P a processor, C a cache, M a local memory and CP a communications processor. The hybrid distributed memory computer contains multiple nodes, although only one node is drawn. A distributed memory computer can also have a communications processor.

Shared memory computers

Figure 1a shows the lay-out of a shared memory computer, also called symmetric multi-processor computer, or tightly coupled multi-processor. It has one shared memory (512 MBytes for the SC2000) that is accessible to all processors via a common bus using implicit access. Implicit memory access, also used by sequential computers, is the access of memory via load/store operations. Since it is the means to exchange data between processors, it is also called implicit communication. The shared memory access performance is characterized by a low latency and a high throughput. The bus of the SC2000 provides a memory bandwidth of 625 MBytes/s. Access to any part of the shared memory yields the same performance. For any two
Section 2.2: Parallel computers

processors, the memory access performance is the same, meaning that the computer lay-out is symmetric. Support for synchronization must be available to coordinate processes.

Since all processors access shared memory via one common bus, the memory access bandwidth of this computer does not scale with the number of processors. Therefore, but also because it is technically difficult to build a common bus with good performance to which many processors can be attached (p.619 of [Patterson94]), shared memory computers usually contain not more than 64 processors. The SC2000 has 20 superSparc processors running at 50 MHz. To decrease the bus contention, every processor has a relatively large cache for the shared memory (the SC2000 has a 2 MBytes cache per processor). A processor together with its cache is called a node. If a processor updates a data item in the shared memory, the copies of this data item in the caches of the other processors become invalid. The control of the validity of cached copies is called multi-processor cache coherency maintenance [Patterson90] and is usually supported by hardware. The unit of access to shared memory and the unit of caching is usually equal to the width of the bus and is 64 bit for the SC2000. This is so small that thrashing is unlikely to occur. Thrashing is the effect that a given data distribution combined with a given data access pattern causes caches to be continuously flushed, resulting in a bad data access performance.

A shared memory computer usually has a single operating system image. This is the case for the SC2000 that runs the SUN SOLARIS 2.4 operating system.

Distributed memory computers

Figure 1b shows the lay-out of a distributed memory computer, also called loosely coupled multi-processor machine. Every processor has its own local memory (128 MBytes per node for the CS-2) that is accessed by implicit memory access and has a cache. A processor, its cache and its local memory together comprise a node. The CS-2 has 64 nodes. The local memories of other processors can be accessed by send/receive operations that transport messages over the network, interconnecting the nodes by links. This type of access is called explicit memory access or message passing (discussed in section 2.5.1). The network can be a switching network that implements point-to-point communications or a network with intrinsic (and therefore efficient) broadcast capabilities like ethernet or a satellite network. Ethernet and satellite networks are mostly used for distributed computing. As this is not the focus of this thesis, we concentrate on the faster switching networks. A Transputer [INMOS93] is an example of a processor with built-in switching capability, to which four links can be connected, resulting in processor networks without special switch components. The CS-2 network is built of Elite network switches [Meiko93/1] interconnected with bidirectional links. It is a tree-like shaped graph with four child edges per parent vertex. Each processor node has a dedicated ELAN communications processor [Meiko93/1] with which it is connected to a leaf of the network. The depth of the tree, also called the number of stages, limits the number of processor nodes that can be connected. Three stages allow 64 nodes.

Access to the local memory shows a low latency and a high throughput. Access to non-local memory is characterized by a high throughput, but with a start-up latency that is relatively high (the communication bandwidth of the CS-2 is 50 MBytes/s in each direction simultaneously from the memory of a processor to the memory of a non-local processor; it is controlled by the ELAN communication protocol). The memory access performance is the same for any processor, meaning that the computer lay-out is symmetric. Extension of the computer is done by adding nodes together with network links. This results in a total memory access bandwidth that is almost linearly scalable with the number of processors and allows computers with thousands of processors. This is an important reason for the existence of distributed memory computers.
Some studies, e.g. [Lukkien95], consider the performance of memory access on a more detailed level than just local and non-local. They take the relative position of processor pairs into account: the more switching stages a message has to pass before arriving at its destination, the longer the communication time. Optimizing the guiding of a message from one processor in the network to another processor is called routing. Routing complicates the programming of a distributed memory computer. The three stage network of the CS-2 results in a non-local memory access that uses five switches in the worst case and one switch in the best case. The communications processor start-up latency is 10 μs and the network switch latency is 200 ns. This means that the total communication start-up latency is between 10.2 and 11 μs depending on the relative position of the processor pair. Hence, the difference in the communication performance between different processor pairs in the switching network is very small in comparison with the total communication time. If also the time of the actual data transfer and the software overhead necessary to embed the communication in the parallel application are taken into account, we see that the relative position of processors has a small effect on the total communication time.

Usually, each node of a distributed memory computer runs a copy of the operating system. This is the case for the CS-2 that runs the SUN SOLARIS 2.3 operating system.

**Hybrid distributed memory computers**

Shared memory computers have good performance if the number of nodes is not too high. Distributed memory computers provide memory access bandwidth that is scalable with the number of nodes, but a high start-up latency for non-local memory access. A hybrid distributed memory computer (see figure 1c) is a distributed memory computer that combines these advantages. A node contains a number of processors connected via a common bus to a shared memory, and the nodes themselves are interconnected via a switching network. The CS-2 is a hybrid distributed memory computer with two 100 MHz hyperSPARC processors [Ross95] per node.

**Computation/communication overlap**

The tasks of a node of a distributed memory computer are (1) computation, (2) implicit access by a given processor to the local memory and (3) explicit access by a given processor to non-local memory. Many parallel computers have dedicated communications processors to take care of (3). This allows fast communications, but also parallelism between (1) and (3), and between (2) and (3), called the communication/computation overlap. A CPU still partly participates in a communication, for example to initiate the communication. We call the I/O load the percentage of CPU time of a processor that is not available for computation tasks during a communication. The I/O load might depend on the size of the communicated message. A parallel computer without dedicated communications hardware has an I/O load of 100%, since the processor takes care of all communication and since there is no possibility to perform a context switch during the communication. The T9000 transputer, an example of a processor designed for fine-grained parallelism, has an I/O load of 30% for a 1 kBytes message [Heeley96].

### 2.2.1 Consequences for software design

From a programmer's point of view, the difference between shared and distributed memory computers is in the performance of memory accesses, the memory access type (implicit or explicit) as seen by the application program, and the homogeneity of the memory with respect to these two features. For the design of our software we will use the following model of shared and distributed memory computers. We will argue why some features are (not) taken into account in the model.
Section 2.2: Parallel computers

The relative position of processors on the CS-2 has a small effect on the total communication time. We neglect the effect and distinguish only local and non-local memory access.

The performance of CS-2 local memory access is comparable to that of shared memory access on the SC2000. It might be slightly better, since the connection between processor and memory does not have to be shared among processors and neither is multi-processor cache coherency to be ensured, but we ignore this difference. The bandwidth of CS-2 non-local memory access is comparable to that of shared memory access on the SC2000. It is slightly smaller, but we ignore this difference. The main difference lies in the much higher start-up latency in the case of the CS-2. The software should be designed such that the latency penalty is decreased as much as possible. Possibilities are (1) to favor the inter-node communication of large jobs rather than small jobs, or (2) to map processes that communicate frequently to the same processor.

On a shared memory computer, all shared memory can be accessed implicitly and the memory access performance is homogeneous. This simplifies the programming model, since the location of data relative to the accessing processor is not of importance. This is not the case for the non-homogeneous memory access performance of a distributed memory computer, often called non-uniform memory access (NUMA). The resulting programming model is more complex, as data distribution is an issue. Secondly, implicit and explicit memory access have to be used to access local memory and non-local memory respectively, which makes the data location (local or non-local) non-transparent to the application programmer.

Concurrently running processes need to be synchronized which complicates the programming of parallel computers. On shared memory computers, this is mostly done by synchronizing access to shared data items by means of locks, which is the application programmer’s responsibility. This is error prone and increases the program’s complexity. On distributed memory computers, a communication is a natural synchronization between processes. According to some [Ward95], this natural synchronization and the modular structure of applications that run on a distributed memory computer, fit better to current software engineering standards. They use this as an argument to prefer distributed memory programming to shared memory programming. However, most people consider shared memory programming to be easier, as can be seen by the design of so-called distributed shared memory systems, which aim at simulating the shared memory programming model on a distributed memory computer (see section 2.5.3).

The symmetric lay-out of both the shared and distributed memory computers provides independence of the node on which a program is run, i.e. a permutation of the nodes does not lead to an incorrect behavior of the program and only marginally influences the performance.

Having two processors per CS-2 node favors mapping multiple processes to a node. The ELAN communications processor of a CS-2 node allows communication/computation overlap. This favors mapping multiple processes to a processor, although the expected gain should be compared with the increased context switch overhead.

2.2.2 Present trend in parallel computing

The initial popularity of distributed memory computers has abated in the nineties. Although very promising, distributed memory computing did not take off as expected. What are the reasons? Computing requirements in industry are not as intensive as in science applications, and hence the push to extend this niche of the hardware market is limited. The expected saturation of the processor speed increase over time, did not come yet. Therefore, hardware requirements of many applications could be satisfied using the cheaper sequential or shared memory computers. The ever increasing processor speed also pushed distributed memory computers towards using easily replaceable commodity processors, allowing easy upgrading. The software support
for distributed memory computers is less and also less reliable than the support for smaller computers, although the first have a higher complexity. This made distributed memory computers difficult to use. They required much maintenance, were not very reliable and the development of applications turned out to be time consuming and cumbersome.

### 2.3 Software parallelization

The motivation for software parallelization is to efficiently exploit the computing power of a parallel computer. The aim is to distribute the application execution load among the processors such that processors run the application in parallel. Parallelization can be performed at various levels of granularity. Parallelization at the processor instruction level results in a superscalar processor: per clock cycle, multiple independent instructions are executed [Patterson90]. Examples are the pipelining of independent processor instructions and the construction of parallel pipelines on the processor chip. This micro-parallelism is transparent to the application code and not the focus of the thesis.

The level of granularity on which this thesis focuses is the parallelization at the component level. Components, consisting of multiple lines of application code, are run in parallel. Component level parallelization can be implicit or explicit. In implicit parallelization, a compiler creates, possibly with the help of compiler directives, object code for programs that have parallel loop execution, parallel array operations and remote execution of procedures. This type of parallelization focuses on vector and matrix calculations. High Performance Fortran [HPF94] is an example of a language equipped with keywords that are used by the compiler as hints to parallelize the code. A problem of implicit parallelism is the generation of the right number of components of the right size. Too many components result in too much communication and administration overhead, too few components result in idle processors, and components of very different size result in little parallelism. Explicit parallelism causes more work for the programmer than the implicit variant, but may provide better performance, as the number and size of the components can be better controlled. Not all applications can be parallelized equally easily. Features that make an application a candidate for parallelization at the component level are:

- The application input is a stream of similarly structured independent jobs. This results in an application with a pattern consisting of independent repetitions that can be parallelized.
- Applications that by their nature consist of easily distinguishable tasks. This is a requisite for explicit component level parallelization. The structure of the application determines the division of the application into components and determines the granularity of parallelization. It is the programmer’s task to recognize the structure of the application and to divide it into modules that each perform a task. For existing software, the division effort can be enormous and should therefore be compared with the expected performance gain. Splitting the code usually induces additional programming complexity. Explicit division of an application into components on a level that is not suggested by the application is an almost impossible task.

In what follows we assume that the application we consider has either one or both features.

### 2.4 (Explicit) component level parallelization

Explicit parallelization (see figure 2 for a summary of the following definitions) at the component level involves the separation of the application into *components*. The application is executed by having components that run in parallel. The specification for a component is the *task* it has to perform. A *code module* is the implementation associated with a task. A component executes a code module. Tasks and components are recursive concepts: a task, performed by a component, may consist of sub-tasks that are performed by sub-components. The terms task
Section 2.4: (Explicit) component level parallelization

(component) or sub-task (sub-component) depend on the view of the observer. The system is the component under consideration, it can be the parallel application or one of its components. The input of the system coming from the environment (which is outside the system) is a job. A subjob is the input of a component and it can come from one or multiple components, or from the environment. A result is the output of a component and (part of) it can be the input for one or multiple components or it can be a system result if no further processing is required.

Jobs are unrelated. A job can consist of a collection of related subjobs. Subjobs belonging to different jobs are unrelated. As component is a recursive concept, so is subjob: a subjob can consist of a collection of related subjobs. Related subjobs can be the input of different components running in parallel. The term job is used for both job and subjob when no confusion can arise. The processing of related subjobs has mutual dependencies. This dependency can be defined via the following dependency relation on components:

- If (part of) the result of a component is input for another component (i.e. it is a subjob), the two components are data dependent.
- The results of components that process related subjobs often have to be recombined, which can be done by a component collecting the results. Often the recombining component can only proceed if all its inputs are there, which means that there is a time constraint on the feeding components that process related subjobs. We call the feeding components with a time constraint imposed on them synchronization dependent.

A data dependency causes data exchanges and synchronizations between components. This is called a communication: a producer component passes data to a consumer component. Data dependency is a directed relation and can be depicted as a directed graph of which the vertices are components and the edges represent data dependencies between components. Communications limit the possible parallelism, since the producer component needs to finish processing a job before the consumer component can start. A synchronization dependency causes synchronizations between components, and therefore might limit the possible parallelism. Synchronization dependency is a symmetric relation and can therefore be depicted as an undirected graph of which a vertex is a component and an edge a synchronization dependency [Papadimitriou82]. Components without data dependencies can process related subjobs in parallel.

Components that each execute the same code module have the same type. A collection of components of the same type is a group. A collection of components of possibly different types is a cluster. The type of a job is defined by the type of the component that processes the job.

![Diagram](image)

**FIGURE 2:** Definitions involved in explicit component level parallelization. Straight arrows indicate data dependencies, dashed lines synchronization dependencies. The branching of the data dependency relation is associated with the synchronization dependency.
2.4.1 Flavors of component level parallelization

Explicit component level parallelization occurs in different flavors. These plus their potential throughput increase, potential latency decrease and memory usage are discussed below. Throughput increase and latency decrease are with respect to the sequential version running on one processor. In intra-job parallelism, related subjobs are processed in parallel and therefore a job latency decrease (and consequently a job throughput increase) can be established. Inter-job parallelism processes jobs or unrelated subjobs in parallel and therefore establishes a job throughput increase, but not a job latency decrease. Notice that inter-job parallelism does not specify whether a job is treated entirely by one component or is split up in subjobs that are processed by multiple components. Figure 3 shows examples of the different types of parallelism.

Memory usage of the parallel application should be divided into memory usage for code and memory usage for jobs that are being processed. Assuming a continuous execution load, the number of jobs that are processed at a given time is equal to the job throughput times the job latency. Therefore, for a given throughput requirement, a latency decrease is a way to decrease the required buffer capacity for jobs that are being processed.

FIGURE 3: Types of parallelism. A box is a component. An arrow between two components indicates a data dependency. An arrow that does not start or end at a component indicates the input from or the output to the environment.

Data parallelism

Data parallelism is established by having a group of components (i.e. components executing the same code module) processing a set of jobs of the same type in parallel. Farming is data paral-
Section 2.4: (Explicit) component level parallelization

farming where a set of jobs or a set of unrelated subjobs are processed in parallel by a group of independent components and is a form of inter-job parallelism. Farming does not guarantee that the job ordering at the output of the system is identical to that at the input of the system. If unrelated subjobs are treated in parallel by a group of data dependent components, the parallelism is called data parallelism pipelining which is a form of inter-job data parallelism. Pipelining, as applied in systolic arrays, is an example of this type of parallelism. The difference between farming and data parallelism pipelining is that in farming, one job or subjob is processed by one component as a whole, while in pipelining one job is processed by multiple components, but by one component at a time. If related subjobs are treated in parallel by a group of components, the parallelism is called geometric data parallelism which is a form of intra-job parallelism.

If the components of a group run on different nodes of a distributed memory computer, data parallelism requires the replication of application code. On a shared memory machine, the application code does not have to be replicated in memory, since memory is shared, however each processor cache has a copy of a part of the application code.

Algorithmic parallelism

In algorithmic parallelism, different types of components run in parallel. If there are no data dependencies between the components, related subjobs can be processed in parallel. This intra-job parallelism is called task parallelism. If there are data dependencies between the components, they cannot process related subjobs in parallel, but they can process unrelated subjobs in parallel. This means that a given job is processed by multiple components, but only by one component at a time. This inter-job parallelism is called algorithmic parallelism pipelining. Algorithmic parallelism does not require replication of the entire application code.

Both algorithmic and data parallelism have forms of pipelining. A general definition for pipelining is that since there are data dependencies between the components, a given job is processed by multiple components, but by one component at a time. Therefore, pipelining is a form of inter-job parallelism.

2.4.2 Resource management and load balancing

In this section it is discussed how an application should be divided into components, taking the properties of the application into account, to achieve the following three goals: to satisfy the (1) throughput and (2) latency requirements and to obtain (3) efficient resource utilization. The basic resources in a parallel computer are the processors, the memory and the inter-processor communication facility. It is important to realize which of these resources can be a bottleneck. We focus on computing intensive applications and assume that the memory of MIMD computers is sufficiently large. The importance of the communication network utilization with respect to the processor utilization depends on the granularity of the parallelization. We do not discuss communication network utilization (see for example [Dobinson95]), but focus on processor utilization.

Properties of the system

The system is the hardware plus the software. For the discussion below, we assume the following system properties. A given system is assumed: the system has a fixed number (N) of identical processors, and a fixed component distribution. L_job is the execution time of a job that is processed by one processor. We assume that L_job is identical for all jobs. We assume a stationary situation (i.e. no transient effects), which is characterized by a constant job input rate. We consider the system behavior as a function of the job rate.
A system is overloaded when the job rate is higher than it can handle. An overloaded system is a non-stationary situation and can only be temporarily maintained. Overloaded systems that do accept any number of jobs usually show a decreasing throughput and an increasing job latency over time. Overloaded systems that do not accept more jobs than they can handle show a constant throughput and latency, but have a growing wait queue of jobs associated with them. Ethernet is an example of a system where overloading results in a decreasing throughput and increasing latency. We make no assumptions whether a system does or does not accept more jobs than it can handle. Figure 4 shows the system behavior as a function of the job rate.

**FIGURE 4**: The throughput and latency of a system as a function of the job rate.

**Definitions**

To make statements about the system behavior, we use the following definitions (see figure 4).

- An idle job is a job that is inside the system and is ready to be processed, but no component has yet started to process it. An idle component is a component that is ready to process a job. A (partly) idle processor is a processor of which the utilization is below 100%. A system is in imbalance if the load is not equally distributed among the processors. Assuming a sufficiently high job input rate, idle jobs and idle processors at the same time in the system indicate imbalance. Imbalances can have negative consequences for the throughput and latency.

- \( T_{\text{theor, max}} \): the theoretically maximum throughput, is the throughput without idle processors. This means \( T_{\text{theor, max}} = N/L_{\text{job}} \). Reasons to not have \( T_{\text{theor, max}} \) are (1) load imbalances or (2) a not sufficiently high job input rate (i.e., over-dimensioned hardware). Since we are interested in the effect of load imbalances, we define \( T_{\text{max}} \), the maximum throughput, as the...
Section 2.4: (Explicit) component level parallelization

throughput when the job input rate is sufficiently high, but without system overloading. The corresponding job input rate is rate_{T_{max}}. A system is in balance if T_{max} = T_{theor.max}.

- The latency (L) is the time between a job entering and leaving the system. L_{theor.min}, the theoretically minimum latency, is the latency when only one job is processed by the system without having idle processors. This means L_{theor.min} = L_{job}/N. A job is idle when it has to wait for other jobs. This increases the latency. L_{isol} is the latency of a job processed in isolation. L \geq L_{isol}, as a job might have to wait for other jobs. L_{isol} \geq L_{theor.min}, because not all processors necessarily participate in processing the job, and because subjobs might have to wait for each other. L_{T_{max}} is the latency at rate_{T_{max}}.

- The efficiency (E) is the computing power used to process jobs divided by the computing power provided by the system. It indicates the quality of the resource utilization. An efficiency below 1 can be caused by: (1) (partly) idle processors (2) communication overhead and (3) processors performing administrative tasks (e.g., the bookkeeping of subjobs, or context switching). In this chapter, we only consider (1) and we ignore (2) and (3). Consequently, the efficiency at T_{max} indicates the amount of load imbalance. As this is the most interesting quantity, the term efficiency is used where efficiency at T_{max} is meant. This means as well that E = T_{max}/T_{theor.max}. A system is in balance if E = 1.

- The system load is the job rate times L_{job}. System overloading is the increase of the job rate above rate_{T_{max}}.

Resource management

Some statements about the throughput, latency and resource utilization of a given system are the following. We assume that the following situations are stationary.

No idle processors and no idle jobs is the ideal situation. The latency is equal to L_{isol} and E = 1. Assuming no external wait queue of jobs, the system satisfies the throughput requirement.

No idle processors with idle jobs means that the E = 1, but the latency is larger than L_{isol}. Additional hardware might reduce the latency.

(Partly) idle processors and no idle jobs mean that the latency is equal to L_{isol} and E is below 1. Assuming no external wait queue of jobs, the system satisfies the throughput requirement. This description fits two possible situations, or their combination. In situation one, the system has T_{max} and the idle processors are caused by load imbalances. Removal of a processor results in idle jobs. To improve E, the load imbalances could be removed by changing the component distribution, after which processing power can be removed. In situation two, the system does not have T_{max} and there are no load imbalances. The processing power of the system is over dimensioned and removing processing power improves E.

(Partly) idle processors and idle jobs mean that E is below 1 and that the latency is above L_{isol}. The system is in imbalance. The throughput and latency can be improved by removing the load imbalances. Adding processing power might improve the throughput, but not the efficiency.

The purpose of load balancing is to avoid load imbalances by distributing the load evenly among the processors, and consequently increasing the efficiency. In static load balancing, the distribution of the load among the processors is determined at compile or initialization time and cannot be changed at runtime. For static load balancing we are only interested in the average system load. In dynamic load balancing, the distribution of the load among the processors can be adapted at runtime to cope with load variance.
Static load balancing

To investigate the effect of load imbalances, we consider the system at rate $T_{\text{max}}$ and are therefore interested in $T_{\text{max}}$ and $L_{T_{\text{max}}}$. We use the following calculation rules. $L_{\text{comp}}$ is the time spent by a component for a job. The subscripts “intra-job”, “pipeline”, “farming” and “sequential” indicate the type of parallelism for which a formula holds. We assume one component per processor.

1. $L_{\text{job}} = (\sum \text{comp} : \text{comp processes related subjob} : L_{\text{comp}})$
2. $L_{T_{\text{max,intra-job}}} = L_{\text{isol,intra-job}} = (\max \text{comp} : \text{comp processes related subjob} : L_{\text{comp}}) = L_{\text{comp,max}}$
3. $T_{\text{max,intra-job}} = 1/L_{T_{\text{max,intra-job}}}$
4. $L_{T_{\text{max,pipeline}}} = (\sum \text{comp} : \text{comp} \in \text{pipeline} \land \neg (\text{comp behind last comp with } L_{\text{comp,max}}) : L_{\text{comp,max}}) + (\sum \text{comp} : \text{comp} \in \text{pipeline} \land \text{comp behind last comp with } L_{\text{comp,max}} : L_{\text{comp}})$
5. $L_{\text{isol,pipeline}} = (\sum \text{comp} : \text{comp} \in \text{pipeline} : L_{\text{comp}})$
6. $T_{\text{max,pipeline}} = (\min \text{comp} : \text{comp} \in \text{pipeline} : T_{\text{max,comp}})$
7. $L_{T_{\text{max,farming}}} = L_{\text{isol,farming}} = L_{\text{comp}}$
8. $T_{\text{max,farming}} = (#\text{processors in farm})/L_{\text{comp}}$
9. $L_{T_{\text{max,sequential}}} = L_{\text{isol,sequential}} = L_{\text{theor.min,sequential}} = 1/T_{\text{max,sequential}} = 1/T_{\text{theor.max}}$

Note that in these formulas, a pipeline stage can also be an intra-job parallelism element or a farming element, instead of a single component processing an unrelated subjob. Also, a task that is part of intra-job parallelism can be a pipeline element or a farming element, instead of a single component processing a related subjob.

Rule 2 says that if related subjobs are processed in parallel, the slowest subjob determines the overall latency. Rule 3 says that if one job at a time is processed, the latency and throughput are inversely proportional. Rule 4 says that the slowest pipeline stage determines the latencies of the preceding pipeline stages. Rule 5 says that the latency of a job processed by a pipeline in isolation is the sum of the latencies of the pipeline stages. Rule 6 says that the slowest component in the pipeline determines the overall throughput. Rule 7 says that the latency of the farm is equal to the latency of a farm element. Rule 8 says that the farm throughput is proportional with the number of processors participating in the farm. Rule 9 uses that a sequential version cannot have load imbalances, which means that the efficiency is always 1.

Figure 5 together with Table 1 show some examples of load imbalances in parallel applications. A white box is a component. The number in a box indicates $L_{\text{comp}}$. One component per processor is assumed. 5a shows the sequential version. 5b shows a pipelined version: the tasks that can be distinguished are each mapped to a component. The pipeline is unbalanced, since component $g$ can only accept jobs every 40 time units, while its predecessor $f$ can deliver results every 10 time units. This has a negative effect on $L_{T_{\text{max}}}$, $T_{\text{max}}$ and $E$. Dependent on the implementation, there is a wait queue between component $f$ and $g$, if there is a possibility to buffer, or $f$ has to slow its throughput down to the throughput of $g$. This latter option is shown in the figure by means of the grey boxes indicating that $f$ waits 30 time units before it passes its job to $g$. Component $h$, the successor of $g$, also has to wait for component $g$. This has a negative effect on the $T_{\text{max}}$ and $E$, but not on $L_{T_{\text{max}}}$. This is in agreement with the figures in the table.

Figure 5c shows an unbalanced system with task parallelism and algorithmic parallelism pipelining. We assumed that task $g$ could be split up in tasks $g'$ and $g''$, that process related subjobs and that each have a different $L_{\text{isol}}$. The task parallelism part is not balanced, since $g'$ and $g''$ are not ready at the same time (indicated by the grey boxes in the figure), and neither is the pipeline part, since $f$ and $h$ have to wait for the $g$'s. The table shows that the task parallelism did result in
Section 2.4: (Explicit) component level parallelization

(a) sequential

\[ f \circ g \circ h \]
\[ 10 \ 40 \ 10 \]

(b) unbalanced pipeline

\[ f \]
\[ 10 \]
\[ 30 \]
\[ g \]
\[ 40 \]
\[ h \]
\[ 10 \]

(c) unbalanced task parallelism and unbalanced algorithmic parallelism pipelining

\[ f \]
\[ 10 \]
\[ g' \]
\[ 10 \]
\[ 10 \]
\[ g' \]
\[ 20 \]
\[ g'' \]
\[ 20 \]
\[ h \]
\[ 10 \]

(d) balanced pipeline by application of farming

\[ f \]
\[ 10 \]
\[ g \]
\[ 40 \]
\[ g \]
\[ 40 \]
\[ g \]
\[ 40 \]
\[ h \]
\[ 10 \]

FIGURE 5: Load imbalances and static load balancing.

<table>
<thead>
<tr>
<th>figure</th>
<th>5a</th>
<th>5b</th>
<th>5c</th>
<th>5d</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_{job}</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>L_{theor.min}</td>
<td>60</td>
<td>20</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>T_{theor.max}</td>
<td>1/60</td>
<td>1/20</td>
<td>1/12</td>
<td>1/10</td>
</tr>
<tr>
<td>L_{isol}</td>
<td>60</td>
<td>60</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>L_{Tmax}</td>
<td>60</td>
<td>90</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>T_{max}</td>
<td>1/60</td>
<td>1/40</td>
<td>1/20</td>
<td>1/10</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0.5</td>
<td>0.6</td>
<td>1</td>
</tr>
</tbody>
</table>

TABLE 1: Examples of static load balancing.

decrease of L_{Tmax} with respect to the sequential version. Notice the effect of the pipelining, which is a form of inter-job parallelism, since L_{Tmax} is not equal to 1/T_{max}.

Static load imbalances can be removed by changing the distribution of the tasks among components or by changing the distribution of components among processors. Figure 5d shows a balanced version of 5b by applying farming. This is not the same as the task parallelism shown in 5c, since in 5d the g’s process unrelated subjobs, while in 5c the g’s process related subjobs. Note that farming and pipelining are forms of inter-job parallelism. This means that the latency is at best (i.e. when jobs do not have job idle time, which is the case for 5d) equal to the latency of the sequential version.

2.4.2.1 Deviation from deterministic temporal behavior

The temporal behavior of the application to be parallelized has not yet been discussed. If the component execution times are known and fixed, the optimal parallelization strategy plus its expected performance, and the distribution of components among the processing power that gives the optimal resource utilization can in principle be calculated beforehand. However, we
take applications into account where the temporal behavior depends on the data they are processing in the following way (leading to dynamic load variations):

- Variation of the job input rate.
- Variable job processing time. The (sub)job processing time by a component depends on the (sub)job under consideration and might be variable.
- Conditional communication. Depending on the job, a component might produce no, one or multiple results which results in no, one or multiple communications, respectively.
- Concurrent solutions. A job latency decrease might be established by components that satisfy the same specification, but have different implementations (i.e. they run different code modules). Only the result of the component that finishes first is used. The other components can be stopped to save resources.

The difficulty to combine a low latency and a high efficiency is caused by dynamic load variations, which make it impossible to predict the job arrival times. To achieve a low job idle time (i.e. \( L_{isol} \)), components should preferably be idle, to allow them to immediately start processing a job upon its availability. However, idle components can result in idle processors and hence a decrease of efficiency. A trade-off between the required latency and the cost of the processing power has to be made, that depends on the throughput and latency requirements.

**Dynamic load balancing**

Static load balancing is not sufficient to avoid load imbalances caused by dynamic load variations. In dynamic load balancing, the job distribution among the components is decided at runtime. It allows load balancing in a system with big load variations, but causes overhead and a complexity (i.e. administration) increase. Buffering can also be used to cope with big load variations, as it averages out the temporary load variations. However, it causes job idle time and thus a latency increase, and therefore is an option for inter-job, but not for intra-job parallelism.

**Effects of conditional communication**

We investigate the influence of conditional communication on the throughput and latency. Minimizing the number of processed jobs results in the minimal system load and hence an enhanced \( T_{th,\max} \). Consider the components A and B that both have conditional communication and that are independent so that their execution order can be freely chosen. To simplify the example, we assume that A and B have only one or no result. We assume one component per processor. The number of processed jobs depends on the execution order of A and B:

1. Pipelining of A and B (A then B, or B then A). We define \( r_A \), the output fraction of component A, as the number of jobs with output divided by the total number of jobs. Ordering the components in ascending output fraction results in the minimal number of processed jobs and hence the minimal system load.

2. Task parallelism applied to A and B. We assume that a job has finished if all components finished their subjobs. Since A and B run in parallel, their execution order cannot be chosen in order to decrease the system load.

Option 1 is preferable if there is a strong throughput requirement. Option 2 is the only solution if there is a strong latency requirement. In Table 2 an example of the described effects is shown. It is filled using the calculation rules mentioned before plus the following additional assumptions and remarks that are usually not true in practice, but facilitate the calculations:

- The decision of a component whether to give or to not give an output is made at the end of the subjob processing.
Section 2.4: (Explicit) component level parallelization

- The arrival of accepted and rejected jobs is not interleaved. First all accepted jobs are processed, then all rejected jobs (or vice versa). In pipeline "A then B", an accepted job suffers from static load imbalance, resulting in idle processor time for the first processor in the pipeline. If accepted and rejected jobs are interleaved, the first processor might during its idle time process other jobs that will be rejected and therefore do not have to wait for the subjob processed by the second processor. Therefore, for pipeline "A then B", in practice $T_{\text{max}}$ is bigger and $L_{\text{Tmax}}$ smaller than the figures in the table. Interleaving has no effect on $L_{\text{isol}}$.

- In the beginning of section 2.4.2, we assumed that jobs have the same execution time. Here, the latency of accepted jobs is different from that of rejected jobs. The latency of accepted jobs is important for systems that have a time constraint on the usage of the results. The average latency is important for systems with a limited buffer space for non-processed jobs.

<table>
<thead>
<tr>
<th>execution order</th>
<th>pipeline A then B (option 1)</th>
<th>pipeline B then A (option 1)</th>
<th>A and B in parallel (option 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>system_load (=L_job,average)</td>
<td>$L_A + L_B*r_A$</td>
<td>$L_B + L_A*r_B$</td>
<td>$L_A + L_B$</td>
</tr>
<tr>
<td>$L_{\text{isol}}$ (accepted jobs)</td>
<td>$L_B+L_A$</td>
<td>$L_B+L_A$</td>
<td>$\text{max}(L_A,L_B)$</td>
</tr>
<tr>
<td>$L_{\text{Tmax}}$ (accepted jobs)</td>
<td>$2*L_B$</td>
<td>$L_B+L_A$</td>
<td>$\text{max}(L_A,L_B)$</td>
</tr>
<tr>
<td>$L_{\text{isol}}$ (average)</td>
<td>$L_A(1-r_A)+(L_B+L_A)*r_A$</td>
<td>$L_B(1-r_B)+(L_B+L_A)*r_B$</td>
<td>$\text{max}(L_A,L_B)$</td>
</tr>
<tr>
<td>$L_{\text{Tmax}}$ (average)</td>
<td>$L_A(1-r_A)+2<em>L_B</em>r_A$</td>
<td>$L_B(1-r_B)+(L_B+L_A)*r_B$</td>
<td>$\text{max}(L_A,L_B)$</td>
</tr>
<tr>
<td>$T_{\text{theor}}$max</td>
<td>0.04</td>
<td>0.017</td>
<td>0.015</td>
</tr>
<tr>
<td>$T_{\text{max}}$</td>
<td>$1/(L_A(1-r_A)+L_B*r_A)$</td>
<td>$1/(L_B(1-r_B)+L_A*r_B)$</td>
<td>$1/\text{max}(L_A,L_B)$</td>
</tr>
<tr>
<td>$E$</td>
<td>0.57</td>
<td>0.88</td>
<td>0.65</td>
</tr>
</tbody>
</table>

TABLE 2: Effects of conditional communication, an example. $L_A$ is an abbreviation for $L_{\text{comp}}$ of component A. The figures are calculated with $L_A$=30, $r_A$=0.2, $L_B$=100, $r_B$=0.5.

In Table 2, we can see that pipeline "A then B" suffers from load imbalances that affect $T_{\text{max}}$ (see difference between $T_{\text{max}}$ and $T_{\text{theor},\text{max}}$) and $L_{\text{Tmax}}$ (see difference between $L_{\text{Tmax}}$ and $L_{\text{isol}}$). The load imbalances in pipeline "B then A" only affect $T_{\text{max}}$. The difference is that in pipeline "A then B" the first component is faster than the second one, which causes jobs produced by the first processor to wait, while in pipeline "B then A" the first component is slower than the second one, which results in processor idle time for the second processor.

We now change the assumption made for option 2: a job is finished when all its subjobs have finished or one of the subjobs has no output. The other subjobs continue their execution, but their results are not used. This new assumption means that option 2 has become an example of a "concurrent solutions system" (introduced in the beginning of this section), since the job latency is equal to the latency of the subjob that finishes first. This has a positive effect on $L_{\text{isol}}$ (and on $L_{\text{Tmax}}$), but the system load did not change with respect to the old option.

For a "concurrent solutions system" a stop option is of interest. A system with stop option is a system where all related subjobs are stopped when any component finishes without an output. A stop option has a positive effect on $T_{\text{theor},\text{max}}$ (so also on $T_{\text{max}}$), as it reduces the system load by avoiding superfluous computation. It also can have a positive effect on $L_{\text{Tmax}}$, but no effect on $L_{\text{isol}}$. Option 2 with stop option performs better than option 1 and better than option 2 without stop option. In option 1 no-output decisions happen in sequential order while for option 2 with stop option always the earliest no-output decision is used.
An example of option 2 "concurrent solutions" with and without stop option is shown in figure 6 and Table 3. The figure shows a job consisting of four related subjobs. A subjob that finishes with (without) output is denoted by YES (NO). The numbers indicates $I_{\text{comp}}$. The job result consists of the results of the four subjobs, if all subjobs have an output, or the job finishes without output, if at least one subjob finishes without output. The speedup is defined as $T_{\text{max,sequential}}/T_{\text{max}}$. The processing order of the subjobs in the sequential version is assumed to be equal to the subjob numbering in figure 6. The parallel version without stop option has a speedup smaller than N. The parallel version with stop option has a speedup of 7.75, which is a super-linear speedup, as it is larger than N. Remark that the larger the variance in the subjob

<table>
<thead>
<tr>
<th>execution order</th>
<th>sequential</th>
<th>parallel without stop option</th>
<th>parallel with stop option</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>system load ($=L_{\text{job}}$)</td>
<td>1.4+1.3+0.4 = 3.1</td>
<td>1.4+1.3+0.4+1.5 = 4.6</td>
<td>4*0.4 = 1.6</td>
</tr>
<tr>
<td>$L_{\text{isol}}$</td>
<td>3.1</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>$1/T_{\text{theor.max}}$</td>
<td>3.1</td>
<td>1.15</td>
<td>0.4</td>
</tr>
<tr>
<td>$1/T_{\text{max}}$</td>
<td>3.1</td>
<td>1.5</td>
<td>0.4</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0.77</td>
<td>1</td>
</tr>
<tr>
<td>speedup</td>
<td>1</td>
<td>2.07</td>
<td>7.75</td>
</tr>
</tbody>
</table>

TABLE 3: The effect of the stop option, an example. 1 component/processor.

execution times and the more-fold the parallelism, the more positive is the effect of a stop option. As well, the stop option decreases the effect of dynamic load variations, because if a subjob has no output, the execution times of related subjobs that are still running are cut to the same length. In practice, the advantages of a stop option might vanish, as it is difficult to design a well-performing global abort mechanism that stops components processing related subjobs.

Summarizing, conditional communication has different effects for different kinds of parallelization. It has no effect for intra-job parallelization without stop option, a positive effect for inter-job parallelization and the sequential version, and a very positive effect for intra-job parallelization with stop option, with even a super-linear speedup possible.

2.4.3 Mapping components to hardware

Mapping one code module to multiple components provides potential parallelism and is associated with load balancing. Mapping multiple code modules to one component decreases the potential parallelism, but can decrease the communication overhead. This section focuses on the mapping of components to processors. The goals of component allocation are to introduce parallelism to satisfy throughput and latency requirements, to have a good processor utilization and to have a low communication overhead.

On a shared memory computer, components are usually not statically bound to a processor, since a context switch of a component to another processor is relatively cheap. This allows for good processor utilization. For example, the operating system of the SC2000 has an automatic
Section 2.4: (Explicit) component level parallelization

dynamic load balancing system that might migrate a component when it is blocked for more than 30 ms. The communication overhead is independent of the component allocation, if we assume that the communication overhead of two components on different processors is comparable to the communication overhead of two components on the same processor.

On a distributed memory computer, components have to be explicitly assigned to nodes. We have the following three options. Components on different nodes allow taking advantage of the parallelism provided by the parallel computer, but inter-component communications are relatively time-consuming. Components on the same node have faster inter-component communications and might allow components to cheaply migrate to other processors to achieve a good load balancing. Components on the same processor facilitate a good processor utilization since the probability of an idle processor is smaller. They can take advantage of overlapping communication/computation, if the processor allows this. However, they cannot run in parallel. The choice between these options depends on the data access pattern of the application under consideration and on the performance requirements.

2.4.4 Summary

For most applications, the variation in the job execution time is bounded. Therefore, if job throughput is the goal of the parallelization, job farming is the best solution, since (1) it is relatively easy to obtain a balanced system, (2) \( T_{\text{max}} \) can be increased arbitrarily high by adding more farm components, (3) it is possible to take advantage of conditional communication in the same way as this is possible for a sequential application, because one component runs the sequential version of the code, (4) the programming effort is relatively small, since the application code does not have to be divided into modules. To cope with the dynamic variation of the system load, buffering can be applied. Disadvantages of farming are that (1) buffering leads to a latency increase, (2) buffering costs memory space, (3) code replication is necessary on distributed memory machines, (4) the order of leaving is not necessarily equal to the order of arrival if no additional measures are taken and (5) the broad granularity makes good resource utilization difficult, if the job rate is low. Section 4.4 presents an example of an application where a job ordering requirement and a high job execution time variance made it impossible to apply farming because of disadvantage (4).

Pipelining can only increase the job throughput and has the following disadvantages in comparison to farming: (1) it has the danger of load imbalances that result in an efficiency below one and a latency increase, although these load imbalances can be taken care of by applying other types of parallelism, (2) \( T_{\text{max}} \) is limited, (3) it incurs in general more communication and administration overhead than farming, since in the latter case fewer data have to be communicated and the data are communicated in larger chunks, (4) conditional communication can be used in the same way as this is possible for a sequential application, although it complicates load balancing and (5) the application code has to be divided into modules. Applying intra-job parallelism usually leads to partly pipelined applications. Algorithmic pipelining as opposed to farming is only interesting if the code size is important. This is not the case for software applications on MIMD computers, since memory is in general not a scarce resource. However, it can be important for embedded systems, since the chip surface and the power consumption usually depend on the code size. Especially a low power consumption is very important for embedded systems [Slater96]. Indeed pipelined algorithms have been used for this purpose.

If latency decrease is a goal then intra-job parallelism is necessary. It has the disadvantage that (1) the potential latency decrease is limited, (2) it easily suffers from load imbalances (especially task parallelism), (3) whether it can be applied or not depends on the application in question, (4) throughput gain by using conditional communication is only possible if a global stop
option is available and (5) the application code has to be divided into modules. More latency decrease can be obtained by partitioning the application at a level of granularity different from that suggested by the structure of the application, although this might require an enormous implementation effort. The additional communication overhead that comes along with a finer granularity might nullify the expected latency decrease.

Parallelization incurs, except for communication and administration overhead, also initialization overhead. The initialization of an application can for example consist of loading of constants and creating data structures like lookup tables. In a parallelized version of the program, the same initialization is often performed by multiple components, since they all need to have access to the constants, and therefore results in an increased overhead with respect to the sequential version. For algorithmic parallelism on a distributed memory machine, it results in code replication (the absence of code replication is mentioned as one of the advantages of algorithmic parallelism) and if the structure of the application is such that reinitializations happen frequently, it leads to a decrease of the expected throughput. With additional measures these disadvantages can be avoided on a shared memory machine.

### 2.5 Inter-component communication

This section focuses on communications between components. Parallelization of software incurs communication overhead that diminishes the throughput and latency improvement. A finer granularity of parallelism results in a better theoretical performance, but possibly in more communication overhead. This is a trade-off depending on the computation/communication speed ratio of the hardware. Communication also introduces administration overhead consisting of bookkeeping of related subjobs, preparation of buffers and buffer access control. Günzinger [Günzinger96] shows that the administration overhead is often orders of magnitude larger than the raw hardware communication latency. The same can be seen if the performance of the message passing paradigms PVM or MPI (see next section) are compared to the raw hardware communication bandwidth and latency. Also our system, CoCa, shows this behavior as we will see in section 8.2. Günzinger built a computer with the communication administration programmable in hardware. This computer showed an orders of magnitude better performance compared to a computer with a ten times higher communication bandwidth, but without this hardware support. However, the results of this comparison should be interpreted with care. Günzinger's hardware can be considered to be special purpose, which can therefore always outperform general purpose computers, as the requirements for the latter result in a design with compromises. On the other hand, special purpose hardware has the disadvantage of long and tedious development and because it is not easy to upgrade, it becomes obsolete quickly.

We start describing some existing communication models that can support inter-component communication. These are message passing (section 2.5.1), remote procedure calls (section 2.5.2) and distributed shared memory (section 2.5.3). In section 2.5.4 we propose some desirable functionality of a communication model.

### 2.5.1 Message passing

A well-known method of passing data between components is via message passing paradigms. Native communication libraries provided with a parallel computer (e.g. ELAN for the CS-2, occam channels for Transputers, and PVM [Geist93] and MPI [MPI94] are examples of such paradigms. The communications between components are established via dedicated channels, meaning that the producer/consumer identification is very strict: the producer (called sender) knows the consumer (called receiver) and vice versa. The communication primitives are called
Section 2.5: Inter-component communication

send and receive. The channels can provide synchronous or asynchronous communication depending on the implementation. Synchronous communication means that a component participating in the communication statement has to wait for its communication partner to participate as well. In asynchronous communication, if the producer starts the communication before the consumer, the data are copied into a buffer and the producer can proceed. Asynchronous communication potentially increases concurrency, but overhead is incurred by copying the data into a temporary buffer. Additional to one sender/one receiver communications, message passing paradigms provide one sender/multiple receiver communications. Examples are broadcasting that distributes a job among all processors, or multi-casting that distributes a job among a selected subset of processors.

Disadvantages of message passing are the following. The application code contains explicit send and receive calls. It is not possible to hide the multi-process/multi-processor structure from the programmer. Inflexibility is introduced, since both sender and receiver need to know about each other’s identification. It causes administration overhead in the application code and the software becomes configuration dependent. Applications where the destination of a job is determined at runtime are not easy to implement and cause a complexity increase of the code. Attempts have been made (PVM, MPI) to provide more hardware independence by making the component location transparent to the application programmer: communications are not between processors, but between components. However, as application code still contains explicit send and receive calls, the multi-component nature of the program is not hidden.

2.5.2 Remote procedure calls

The remote procedure call (RPC) paradigm, proposed by Birrel [Birrel84], provides more flexibility than message passing, because of the larger freedom in the producer/consumer identification. The producer reaches the consumer by specifying the wanted service by name and the consumer does not know the producer. This asymmetric producer/consumer identification makes RPCs particularly suitable to be used as interface in client/server designed applications. Therefore, the producer is usually called the client, the consumer the server, the communication from the client to the server the request, and the communication from the server to the client the reply. The RPC is a natural extension of the normal procedure call. A procedure is executed on the same processor as its caller, while an RPC is executed on a different processor. The similarity between a procedure call and an RPC provides hardware transparency, as the client does not need to know whether it is doing a procedure call (executed locally) or an RPC (executed remotely). The application code looks like a sequential program. RPCs are most suitable for synchronous communication: the client waits for a result of the RPC before proceeding, as it then knows that the RPC has finished.

The RPC has a number of disadvantages. It is a distributed programming model not designed for use in parallel applications. Its synchronous character tends to suppress rather than encourage concurrency [Bjornson87]. Secondly, the RPC client/server identification requires a mechanism to notify clients about the available services. Thirdly, asynchronous communication is possible for RPCs, but introduces more complexity for the application programmer. Furthermore, the RPC has restrictions on the parameter model that do not exist for procedure calls. Restrictions exist for parameters that are references to variables defined in the client code, since these variables do not exist in the server code. The different semantics with respect to procedure calls complicate obtaining transparency of the procedure execution location and therefore hinders achieving hardware configuration independence.
2.5.3 Distributed shared memory systems

The goal of a distributed shared memory system [Protic96] is to simulate shared memory on a distributed memory computer with the aim to combine the scalability of a distributed memory computer and to allow programmers to use the simpler programming model of a shared memory computer. It is a distributed memory machine with a layer, in hardware and/or software, on top of it that presents the local memories of the nodes as one address space that can be accessed implicitly by the application. This facilitates the programming, but the application programmer loses the ability to control the distribution of data among the non-homogenous memory of a distributed memory computer. This might result in abundant non-local memory accesses.

Much of the research on distributed shared memory systems is based on Li's work [Li89]. Their approach is based on the implementation of virtual memory as used in sequential computers: if a page is not in memory, a page fault occurs and the page is retrieved from disk. For distributed shared memory this translates as follows: if a page is not in local memory, a page fault occurs and the page has to be retrieved from another processor (that loses the page). Various improvements to this algorithm have been proposed, mostly based on combinations of replication and migration of pages, to decrease the number of non-local memory accesses. The effectiveness of these improvements depends on the data access pattern of the application (e.g. the read/write ratio). Other approaches (e.g. Munin [Bennet90]) weaken the semantics of shared memory to increase the efficiency, but this increases the complexity for the programmer.

A disadvantage of distributed shared memory working on a page basis is that the communicated data items are often smaller than a page, dependent on the granularity of parallelization. The not used part of the page is copied anyhow and increases the communication time, as the packet size used in switching networks is usually smaller than the page size. A second problem is that two unrelated data items residing on the same page can cause competition for the page which results in thrashing, i.e. a page is communicated back and forth. So-called structured distributed shared memory systems have been proposed, which work on a data item basis rather than on a page basis. Since a data item is an entity that exists in the parallel application code, these systems depend more on the application: a change of programming style of the application requires changes of the communication system. Examples of such systems are Orca [Tanenbaum92] and Linda. Orca assumes an efficient broadcast mechanism and is therefore outside the scope of this thesis. We discuss Linda.

Linda [Bjornson87] is described as a coordination language, coordinating inter-component communications in a parallel program. The aim is to abstract from any specific machine architecture [Bjornson87]. It uses the tuple space concept. A producer can communicate data by putting them in the tuple space. A consumer can take them out of the tuple space on the basis of the content or type of the returned tuples. This establishes asynchronous communication and a very flexible producer/consumer identification as the producer does not know the consumer and vice versa. This flexibility facilitates having multiple consumers accessing the same data, as at runtime it can be decided which and how many consumers access the data. The set of consumers can be changed dynamically without any reprogramming of the producer component. Furthermore, a tuple has a type, which is called its signature. Tuples with new signatures can be constructed at runtime. By invoking a consumer primitive with a (simple) predicate that is based on the tuples content and signature, only tuples satisfying the predicate are returned. Linda has a large expressive power. In a very compact way parallel applications can be written that can run without modification on various hardware architectures.

Disadvantages of Linda mainly concern performance and are caused by its generality. Hashing techniques are used to find tuples in the global Linda tuple space without much searching, but this makes data locality difficult to establish, which results in long communication times. Sec-
ondly, the dynamic typing of Linda causes runtime overhead. Furthermore, a programmer using Linda still has to deal with two ways of memory access: implicit for local variables and explicit (using Linda primitives) for variables in Linda space.

2.5.4 Data dependencies between collections of components

The communications of a parallel application incur administration overhead that complicates the code. A communication layer usually takes care of correct point-to-point communications. We propose that a communication layer should provide additional functionality that helps to establish a correctly working and well-performing parallel program. Three features are proposed that tackle problems occurring in programs parallelized at the component level. The features are introduced here and will be further assessed in chapter 7. The three features are (1) dynamic communication patterns between sets of components, (2) correct recombination of related subjobs, and (3) a global stop option. By integrating these features into the communication layer, the complexity of the application code induced by bookkeeping is decreased, and it is avoided that coding of the involved bookkeeping has to be done for every new application again. None of the existing models, discussed in the previous sections, support these features. They only ensure the correctness of individual communications.

FIGURE 7: Communication between components.

Figure 7 shows an application parallelized at the component level. Circles are components, arrows data dependencies. A component consumes subjob(s), performs the specified computations and produces result(s). The numbers in the figure correspond to four types of communication:

1. One producer passes data to one consumer. This is typical for pipeline parallelism: a subjob is passed to the next component.
2. One producer passes different data to multiple consumers. This is typical for geometric data parallelism or farming: related or unrelated subjobs are distributed for parallel execution.
3. Multiple producers pass data to one consumer. This is typical for intra-job parallelism to reassemble the results of related subjobs that have been distributed for parallel execution.
4. Multiple producers pass data to multiple consumers. This is typical for combined inter- and intra-job parallelism: related and unrelated subjobs are distributed among components simultaneously.

The above classification differs from the single/multiple writer - single/multiple reader model. For example, item 2 has nothing to do with broadcasting, since in the general case not the same job is distributed among components, but different ones.

Dynamic communication patterns between sets of components

Figure 7 shows examples where subjobs produced by multiple producers are passed to multiple consumers. This is communication between two clusters of components. For data parallelism a cluster degrades to a group. If the communication pattern between the two clusters can be established at runtime, dynamic load balancing that is transparent to the application programmer is possible: the application programmer only deals with clusters, while the communication
model produces communication patterns between the components of the clusters such that
dynamic load balancing is established. The dynamic communications feature diminishes the
complexity of the application code induced by the parallelization, as is shown in chapter 8.
The dynamic load balancing takes care of the distribution of jobs among the members of a clus-
ter. This distribution is (1) fully dynamic, if a component receives a job when it becomes idle,
(2) semi-dynamic, if jobs are distributed among members of a cluster according to data distri-
bution strategies that base their decision on runtime information, or (3) fully static, if the distribu-
tion is determined at compile-time. The first option usually induces a relatively large
overhead. Furthermore, it might be implemented by components that request a job when they
become idle. This allows a correct reaction on load imbalances, but might cause component idle
time, because there is a delay between a component’s request for a job and its reception.
The second option can be realized with a negligible performance loss. It uses information about
the jobs to determine their destination. This predictive nature does not have the delay effect of
option 1, but suppresses immediate reaction to instantaneous load imbalances. Option 3 can
also be realized with a negligible performance loss. Since the communication pattern is deter-
mained at compile-time, it cannot use runtime job information for its distribution decision and is
therefore less flexible than option 2.
A dynamically adjustable cluster composition allows reacting on dynamic load variations by
changing the number of components in a cluster.

Correct recombination of related subjobs

Related and unrelated jobs can be processed simultaneously. A component that receives sub-
jobs from multiple components needs to correctly reassemble the related subjobs results to
yield a job result. This requires administration to keep track which components process a set of
related subjobs. The overtaking problem is an example of an erroneous reassembly of related
data, when the processing time of the subjobs is different and no additional measures are taken.
We discuss an example of the overtaking problem. In Figure 8, a component of type A receives
two related data items x1 and y1, and sends them for treatment to the component of type X and
the upper component of type Y. The component of type X finishes before the component of
type Y and sends p1. Consecutively, the component of type A receives two related data items
x2 and y2 (independent of x1 and y1) that it sends to two free components, namely the compo-
nent of type X and the lower component of type Y. It is possible that the component of type B
receives p1, q2, p2 and q1 in this order and recombines unrelated data if no extra complexity
increasing measures are taken. We propose that the communication layer should hide this prob-
lem from the programmer, such that data items are correctly recombined.

![Diagram](image)

**FIGURE 8**: Component level parallelization.

Global stop option

Section 2.4.2.1, deviation from deterministic temporal behavior, showed the benefits of a glo-
bal stop option. Such an option involves the stopping of the components that process related
subjobs without influencing unrelated subjobs that are processed simultaneously. This means
that administration is required to keep track which components process a set of related subjobs.
Chapter 3: Databases

3.1 Introduction

The reason to include a chapter about databases is the later introduction of a communication facility that resembles a database in many respects. Some aspects of database theory and design are covered to refer to this knowledge for the design of the communication model.

A software application contains a collection of data that together represent a model of the real world. The collection of data is called a database. If the amount of data is large and the data complexity is high, a database management system (from now on called database) can be used, which is specially designed to efficiently and conveniently store and retrieve large quantities of data of high complexity. To model the real world and to access data efficiently, data must be stored in a structured way. The database offers the means to do this. An important requirement on databases concerns the quality of the data. This translates into support for data consistency, data security and data durability. Database usage can be divided into data modeling (section 3.2) and database access (section 3.3). Section 3.4 covers some topics concerning database design.

3.2 Data modeling

A database uses different data models that each describe the data at a different abstraction level. Examples of models that structure data according to the structure in the real world and express relations between the data as they exist between the real world objects are the entity-relationship model and the object-oriented model. The way of data representation of the object-oriented model is based on an extended entity-relationship model. The relational model is an example of a data model that describes the data at a conceptual level, but also provide a high-level description of the implementation. We briefly discuss the entity-relationship model, the relational model and the object oriented model.

The entity-relationship model

The entity-relationship model (see for example [Korth91]) models the real world as a collection of entities and relationships between these entities. An entity set is defined by a set of attributes describing properties of the entities of the entity set. An entity is an instance of an entity set consisting of a set of values; a value is an instance of an attribute. A relationship is an association between entities. A relationship set is defined on a number of entity sets. The cardinality is of importance for binary relationships and can be one-to-one, one-to-many or many-to-many. An example of an entity-relationship diagram is shown in figure 9. Boxes represent entity sets; links between boxes represent relationships sets. The cardinality is indicated by the figures at the link extremities. Three relationships are drawn. One high energy physics event has many associated tracks (one-to-many) and many associated vertices (one-to-many). A track can
belong to multiple vertices whereas in a vertex multiple tracks begin or finish (many-to-many). See chapter 4 for an explanation about high energy physics events, tracks and vertices.

FIGURE 9: An example of an entity-relationship diagram.

Entities in an entity set can be distinguished by means of the values of their attributes. A key of an entity set is the set of attributes that allows unique distinction of the entities. A strong entity set is an entity set for which it is possible to define a key. For a weak entity set there are not sufficient attributes to allow a unique distinction. Weak entity sets are only meaningful, if they are used in a one-to-many or one-to-one relationship with a strong entity set, in such a way that it is possible to distinguish between the weak entities related to one strong entity. In figure 9, event is a strong entity set with the event number as key. Track and vertex are weak entity sets, as track number and vertex number only have a meaning in the context of a specific event.

The relational model

The relational model, first introduced by Codd [Codd70], represents entities and their relationships by means of tables. To every entity set corresponds a table. An entity corresponds to a tuple in a table. A tuple is a set of labeled values. The concepts of the entity-relationship model map to the concepts of the relational model as follows. A one-to-one relationship set can be represented by one table that contains both entity sets. A row of the table contains related entities. A one-to-many relationship set is represented by two tables. One table contains the “one” entity set, the other table contains the “many” relationship with an additional column containing the key of the “one” entity set. A many-to-many relationship set is represented by three tables. Two tables containing the entity sets and the third table containing the two keys of the entity sets. A set of tables that could represent the example of figure 9 is: (event number, event data), (event number, track number, track parameters), (event number, vertex number, position), and (event number, track number, vertex number).

The object-oriented model

Many programming languages propose ways to structure a program’s data and functionality, resulting in a modular structure. The object-oriented (OO) model proposes a method of structuring data and functionality in which related data types and associated functionality are clustered together in so-called classes. The aim of this way of structuring is to improve the reusability of software. As a definition of the OO model we use an adapted version of a definition provided by Meyer [Meyer88]: OO design is the construction of software systems as structured collections of abstract data type implementations. It has the following features:

1. Systems are modularized on the basis of their data structures, resulting in an object-based modular structure.
2. The modules are data types and every non-simple data type is a module. These modules are called classes. Classes combine data types and the associated functionality. Their instances are called objects.

3. Data abstraction. Objects should be described as implementations of abstract data types. This is also referred to as encapsulation: the outside world only sees the object interface, not the internal structure. Therefore, the object usage becomes independent of the actual object implementation; different implementations can be used as long as the interface remains the same, thereby encouraging the reuse of classes.

4. Inheritance. Classes are constructed by successive extension and specialization. A derived class can be defined as an extension or restriction of its base class. Inheritance encourages reuse, since classes with a general functionality can be provided that can be reused to build specific class implementations. Inheritance relies on overloading and dynamic binding.

5. Polymorphism. Polymorphism is the ability to define program entities that may take more than one form:
   
i. **Dynamic (or runtime) polymorphism.** An identifier or reference of a certain class type can be associated with objects of that type or objects of a derived class type.

   ii. **Overloading** is the ability to attach more than one meaning to a name, ambiguities being resolved by looking at the context of each occurrence of the name, either at compile-time or at runtime. It can be used to express the similarity of action on different classes. Each overloaded instance has a different implementation.

   iii. **Dynamic binding** allows overloading that is strongly related to dynamic polymorphism. Operations can have different realizations for different classes. At runtime the object’s class type is determined and the associated operation implementation is executed.

   iv. **Deferred functions** are a variation on the overloading as mentioned in iii. and relies on dynamic binding. For the base class, only function signatures are provided. Associated function implementations are provided in the derived classes.

   v. **Genericity** is the ability to define parameterized classes, the parameters being usually types. It is a form of static (i.e. compile-time) polymorphism and encourages the reuse of source code.

Examples of object-oriented languages are C++, Eiffel, Smalltalk and Java. Since Meyer is the creator of Eiffel, the features in Eiffel have names similar to the ones used in the definition of OO design given above. In C++, genericity is called templates and deferred functions are called virtual functions.

### 3.3 Data access

A database can be viewed as a set of data items and a set of predicates (integrity constraints). A database is consistent if the set of predicates holds over the data items. Retrieving data that satisfy a specified predicate is also called querying. Storing and retrieving data is done via transactions. A transaction is composed of a set of totally ordered actions that each execute on individual tuples. Basic actions that every database provides are the insert, to store a tuple in the database, the select, to retrieve a tuple satisfying some predicate from the database, and the delete, to remove a tuple from the database. A transaction should obey the ACID constraints:

- Atomicity. A transaction finishes completely or aborts without leaving an effect. We call this property fail atomicity to distinguish it from concurrency atomicity.
- Correctness. A transaction executed on a consistent database should leave the database in a consistent state.
• Isolation. A transaction executed concurrently with other transactions yields the same result as its execution in isolation. We call this property concurrency atomicity.

• Durability. Permanence of the results of modifications to the database should be ensured in spite of failures of the underlying hardware.

We call a transaction atomic, if it is fail atomic and concurrency atomic. The concurrency control mechanism of a database ensures atomicity. A successfully completed transaction is said to be committed and its result is permanent. If a transaction cannot complete because of a data conflict with another transaction, two strategies are possible: the transaction is aborted and restarted or the transaction waits for the data conflict to resolve. The wait strategy sometimes leads to deadlock situations. Abortion can give more overhead than waiting, because fail atomicity might require a roll back operation, and because the transaction has to be recomputed.

To allow concurrency within a transaction the nested transaction model is proposed: a transaction can consist of (1) a set of transactions, or (2) a list of atomic actions on data items. As a consequence, the actions of one transaction are no longer totally ordered.

Databases are classically looked upon as necessary, but slow systems. This is related to the durability property: data are stored on a usually slow secondary storage device. Dropping the durability requirement allows the suppression of the secondary storage device: data can be stored in the faster main memory of a computer. The result is a main-memory database. For a shared memory machine, the main memory is the memory shared by all nodes. For a distributed memory machine, main memory is the total of the local memories of the nodes.

An index is applied to improve the data retrieval performance. An index in a database has a similar functionality as an index of a book. It is a list of references to tuples in a table. This list is ordered according to a search key, which can be the value of an attribute. The efficiency of an index depends on the data access pattern. Indices increase the overhead of delete, update and insert actions, since the index has to be maintained. If the read/write ratio is high, indices are useful; if it is close to one, the advantage of indices is small or negative. Two types of indices are linear and B-tree indices. Linear indices provide a table access time that is proportional to the table size. A B-tree index is a balanced tree and provides an access time that is logarithmic in the table length, but it has more maintenance overhead than linear indices.

3.4 Database design

Since our interest is in parallelism, we discuss parallel database architectures (section 3.4.1) and concurrency control algorithms allowing transactions to be processed in parallel (section 3.4.2). Many design aspects (e.g. fault tolerance, integrity constraints and security) that are important for normal databases are not discussed, since they are not relevant for this thesis.

3.4.1 Parallel database architectures

A parallel database uses parallel hardware to improve its performance. It can perform requests of different users in parallel which increases the query throughput (inter-query parallelism) and/or it can perform one request in parallel leading to a higher query throughput but also a decreased query latency (intra-query parallelism).

Three main architectures are known for parallel database systems: the shared everything, the shared disk and the shared nothing approach (see figure 10). In the shared everything architecture, all computer nodes have direct access to a common memory and to secondary storage devices (e.g. hard disks). In the shared nothing architecture, each node has its own directly accessible memory as well as its own directly accessible disks. The system is fully distributed
Section 3.4: Database design

and inter-node communication goes via message passing. In-between shared everything and shared nothing is the shared disk architecture: a node has its own directly accessible memory and can directly access all disks, but it cannot directly access other nodes’ private memory.

A shared nothing database has relatively autonomous units executing per node: every node hosts a small database. This small database can be relatively autonomous (e.g. it has a private lock manager) or some services can be centralized (e.g. a central lock manager). As for any parallel or distributed system, centralized services can decrease the scalability and the fault tolerance. A shared everything database has usually a very cooperative structure. Often, a node runs a process with a specialized task like the lock manager, the optimizer or the deadlock monitor.

Many advantages and disadvantages of the three architectures are identical to those mentioned for shared or distributed memory computers (section 2.2). The main advantage of shared nothing is its scalability. Shared everything systems are difficult to scale to a large number of CPUs because of the shared bus constant memory bandwidth, cache coherency maintenance, and the technological difficulty to build such a system. Scalable shared disk systems are difficult to realize, since special expensive hardware is required, although cheaper than shared memory hardware, and a global locking mechanism is required which reduces scalability. Disadvantages of shared nothing are its complexity and that data availability at the node level is problematic which makes data distribution important. Shared everything and shared disk systems are less complex and data distribution is of less concern: data can be striped among disks for efficiency, but there is a single point of access.

The three architectures are each closely related to a hardware architecture. Shared nothing architectures go well together with distributed memory computers, while shared everything architectures go well together with shared memory computers or distributed shared memory computers. Shared disk databases do not map very well to shared or distributed memory computer and therefore not many implementations exist.

Data distribution

Data distribution, also called data partitioning, is an issue for shared nothing databases, since non-local data have to be retrieved via a network that is in general slow in comparison to local access. Replication and migration enhance locality of data that are read more than once, but incur overhead. Replication is advantageous if the read/write ratio of data is high. If this ratio is close to one, migration is preferable. For the transfer of the data from the environment into the database, which we call the (initial) data distribution among the computer nodes, distribution strategies can be used to obtain a certain data distribution. Several strategies are possible: tuples can be distributed cyclically among the nodes (round robin), tuples can be clustered according to one of the tuple’s data item values (range partitioning), or tuples can be clustered according to some hash function based on the tuple content (hashing).
Function shipping versus data shipping

The choice between function versus data shipping is an issue that mainly holds for shared nothing database architectures. Function shipping on a distributed memory machine means that a request is executed remotely. This is the client/server protocol: a request is sent to a node that has the required information, the request is executed by the callee, and the result is sent back to the caller. So only requests and results are transported over the network. If the information needed to process a request is distributed among multiple nodes, the request has to be split in multiple requests that are each sent to one of these nodes. Data shipping on a distributed memory machine means that a request is executed locally. Data that are needed, but not locally available, are transported to the node that executes the request. The difference between function and data shipping is shown by an example of a select operation that searches for a specific tuple in a table. In the function shipping approach the query is sent to the nodes that contain the table. If the tuple is found, it is sent to the requesting node. In the data shipping approach the nodes that contain the table send the complete table to the requesting node. The requesting node performs the query on the table it received.

In most cases of shared nothing configurations (on a distributed memory machine) function shipping leads to little and data shipping to much network traffic, since the size of the specification of the function and the function result are usually smaller than or as big as the data needed to produce the result. A counter example is a compiler where a small input (the application code) and few data (the rules for target code generation) lead to a large result (the target code). The data location transparency that distributed shared memory computers (section 2.2) offer, leads to "heavy client" applications where the required data are shipped (replicated or migrated) to the client. On a distributed memory machine, this results in much network traffic. Shared everything databases (on a shared memory machine) should be regarded as data shipping systems, since every node can access all data.

3.4.2 Concurrency control

To improve the transaction latency and throughput, actions of different transactions can be allowed to interleave. However, some schedules of interleaved actions can destroy the database consistency as they do not satisfy concurrency atomicity. The concurrency control mechanism ensures fail atomicity and concurrency atomicity in spite of the interleaved transactions.

A schedule is a sequence of chronologically ordered actions. $R_i(x)$ ($W_i(x)$) stands for a read (write) action performed by transaction $i$ on data item $x$. A serial schedule is a schedule in which for every two transactions, all actions of one transaction are executed before any of the transactions of the other transaction. Serial schedules preserve concurrency atomicity. A conflict serializable schedule is a schedule $S$ that can be transformed into some serial schedule containing all actions of the schedule $S$ by swapping non-conflicting actions. Non-conflicting actions can be: (1) actions that work on different data items stored in the database or (2) actions that work on the same data item, but that are both read actions.

The class of view serializable schedules includes conflict serializable schedules. View serializability recognizes that some writes that are conflicting, do not influence the final result. A more formal definition of view serializability is as follows. A schedule is view serializable if it is view equivalent to any serial schedule. Two schedules $S$ and $S'$ are view equivalent if:

- For each data item, if a transaction in schedule $S$ reads the initial value of the data item, then this transaction must also read the initial value of the data item in $S'$.
- For each data item, if a transaction $T_1$ in schedule $S$ reads a value produced by transaction $T_2$ (if any), then transaction $T_1$ must also read the value produced by $T_2$ in schedule $S'$. 
For each data item, the transaction (if any) that performs the final write to the data item in schedule $S$ must perform the final write to the data item in schedule $S'$.

The schedule $S = R_1(x) W_2(x) W_1(x) W_3(x)$ is view equivalent to the serial schedule $S' = R_1(x) W_1(x) W_2(x) W_3(x)$, since in both schedules $T_1$ is the first transaction to read $x$, $T_3$ is the last transaction to write $x$ and there are no further read actions. Since $S'$ is serial, $S$ is view serializable. However, $S$ is not conflict serializable, since $R_1(x)$ and $W_2(x)$ are conflicting and $W_1(x)$ and $W_2(x)$ are conflicting and can therefore not be swapped to obtain a serial schedule, hence the schedule is not conflict serializable.

Examples of algorithms that ensure conflict serializable schedules are two-phase locking, timestamp ordering, optimistic concurrency control and multi-version schemes. Two-phase locking uses locks to suspend actions on data items in case of data conflicts until a conflict serializable transaction execution is possible, or until a deadlock situation is detected after which some of the conflicting transactions have to be aborted. A problem of this algorithm is that it can deadlock which requires deadlock detection mechanisms that cause overhead. Timestamp ordering allows schedules that are conflict equivalent to a serial schedule in which transactions are ordered according to the time on which they started execution. Timestamp ordering is deadlock free, but the strategy to resolve conflicts with aborts, might cause more overhead than a wait strategy. An optimistic concurrency control algorithm first executes a transaction and then validates, before committing, if the transaction execution was serializable. Optimistic concurrency control performs well if the number of data conflicts is low [Huang91], [Stok96/1], for example when there are many read-only transactions. This is not the case for our application as will be seen later. In a multi-version scheme, each write operation creates a new version of a data item. Therefore, read requests never fail. When a read action on a data item has to be performed, the concurrency control mechanism selects a version of the data item that ensures serializability. Multi-version schemes are useful if data items are often updated [Stok96/1].

A design consideration is at what level the concurrency control algorithm should work: table level, tuple level and data item level are candidates. A fine granularity allows more parallelism, but has more concurrency control overhead. A large overhead may considerably increase the transaction execution times and consequently increase the probability of data conflicts.
Chapter 4: Parallelism in high energy physics

4.1 Introduction

This chapter introduces high energy physics (HEP) experiments and the associated software. We will mention some properties of HEP experiments with the resulting requirements for the software. This chapter uses the Compact Muon Solenoid (CMS) experiment [CMS94] of the Large Hadron Collider (LHC) [LHC95] that will be operational in 2005, as an example. We start with an explanation of HEP experiments. In section 4.2 the processing of the data produced by the HEP experiment are described. Section 4.3 looks at the different data flows that occur in a HEP experiment. In section 4.4 the importance of parallelism in HEP data processing is discussed. In section 4.5 data management aspects of a HEP experiment are treated. The importance of hardware independence in the HEP environment is shown in section 4.6.

In the LHC, two counter rotating particle beams cross every 25 ns. During a beam crossing, multiple interactions between highly energetic particles (proton/proton pairs in the case of the LHC) produce several new particles. The position where an interaction took place is called a primary vertex. Created particles with a short lifetime decay into other particles. The position where a decay takes place is called a secondary vertex. The created particles pass through detectors that allow the deduction of their physical parameters. On the occurrence of interesting interactions, the data from the detector electronics are written into detector buffers. The data associated with a beam crossing that contains interesting interactions are called an event. It is predicted that the CMS experiment will produce $10^{12}$ interesting events per year, each with a size of about 1 MBytes.

The main aim of the CMS detector is to prove the existence of the predicted Higgs boson [Veltman86] and to establish its mass. There are multiple theories predicting its mass. A measurement of the mass therefore may allow the rejection/verification of the theories. A Higgs boson is a particle with a short lifetime that can decay in certain ways, called channels, into other particles. By detecting these other particles, the Higgs boson can be discovered and its mass and other physical parameters can be determined. Two important decay channels are a decay into two photons and a decay into four leptons. The CMS detector is specifically designed to detect these channels in the predicted energy range. Figure 11 shows a 3D view of the CMS detector. The uniform magnetic field generated by the superconducting coil bends the trajectories of charged particles. The CMS detector contains a number of sub-detectors: the inner tracker, the ECAL, the HCAL, the very forward calorimeter and the muon chambers. The inner tracker and the muon chambers consist of cells. If a cell detects a passing particle, it has a so-called hit. The hits allow reconstruction of the curved tracks of charged particles. A particle that passes the muon chambers and is detected by them can be identified as a muon. The calorimeters, consisting of the electro-magnetic calorimeter (ECAL), the hadronic calorimeter (HCAL), and the forward calorimeter measure the energy dispositions of charged and neutral
FIGURE 11: A 3D view of the CMS detector\(^a\).

\(a\). Courtesy of the CMS collaboration, CERN.

particles. These energy dispositions occur in clusters with specific forms that allow particle identification. An example of a cluster is a jet, which is a high energy disposition caused by many particles that go through a small solid angle of the detector. The calorimeters each detect different particles. The ECAL detects photons and electrons, and therefore allows detection of the photon decay channel. The HCAL and the very forward calorimeter detect hadrons. The energy clusters and curvatures of the particle tracks of an event, together with physics laws of conservation, allow determining the identity and the physical parameters of the particles.

\subsection{4.2 Event reconstruction}

Event reconstruction is the process of translating raw event data produced by the detector into parameters that describe the physical properties of the created elementary particles. Figure 12 shows an example of a simulated event that is a four lepton channel, i.e. four electrons are created. The side view shows the tracks of all created particles, the cross-section shows the tracks of the four electrons and a jet. The enormous number of particles created by a beam crossing heavily complicates the event reconstruction. Event reconstruction consists of a number of tasks that are closely related to the parts of the detector:

1. The reconstruction task starts with translating the detector outputs into physical quantities like hits and energies, using calibration and alignment constants that together provide information about the lay-out and state of the detector, like the position of the detector parts, values of the magnetic field and zero currents of AD converters.

2. The track finding task finds and reconstructs tracks using the hits in the inner tracker and muon detector.

3. The track fitting task fits mathematical trajectories to the observed particle tracks using physics laws of conservation.
Section 4.3: Data flows in the CMS experiment

FIGURE 12: A simulated decay of a Higgs boson into a ZZ* pair in the CMS detector. The ZZ* pair decays into four electrons.

a. Courtesy of the CMS collaboration, CERN.

4. The calorimeter task finds energy clusters measured by the calorimeters and calculates the energy of the particle.

5. The vertex fitting task calculates the primary vertices of the particle trajectories, using the fitted particle trajectories and energy clusters.

6. After the primary vertex finding, the particle identification task identifies the types of the particles that were produced by the interactions. Furthermore, it attempts to unravel the interactions and decides which tracks belong to which interaction. This is not easy, since one beam crossing generates 20 interactions on the average, and one interaction involves many particles. Therefore, precise measurements of tracks and vertices are required. Secondary vertices are determined and jets are reconstructed. Finally, particle types and physical parameters are determined.

Not only task 1, but also the other tasks use the calibration and alignment data. The event reconstruction process considers events as independent. No information from previous events is used for the reconstruction of the event under consideration. This has the advantage that: (1) no buffering of previous events is necessary, which reduces the required buffer memory, and (2) events can be reconstructed in any order, since the order does not influence the result. This means that farming can be used to satisfy the throughput requirement.

4.3 Data flows in the CMS experiment

In the CMS experiment a number of data flows can be distinguished, some online (during data taking), some offline (after data taking): (1) data acquisition, (2) control, (3) recalibration, (4) offline re-reconstruction, (5) analysis and (6) simulation. We discuss the data flows separately and mention some of their functional and performance requirements.

The data flows can be divided into production data flows and control/interactive data flows. A production data flow has as goal to produce values: data are read from the input, processed and results are written to the output. It has as characteristic that the read/write ratio is close to one and there are no updates: data are written and read once. The associated production software implements the functionality. It is not interactive and there are no feedback loops. A control/interactive data flow controls a part of the environment, e.g. the detector. It comprises gathering of information about the controlled environment, comparing the actual state of the controlled environment with the desired state, possibly taking corrective action and informing users about the state of the controlled environment. It has as characteristic that the read/write ratio can be larger than one, for example reading of status information might have to be done frequently,
Parallelism in high energy physics

and that data might be often updated, for example status information might be updated frequently. We call the associated software control/interactive software. It can have interactive behavior and can have feedback loops, for example if it is used for control purposes.

Data acquisition

The 40 MHz rate at which events take place in the CMS experiment makes it impossible to store all raw events on relatively slow mass storage. Each step in the reconstruction process reveals new information that can be used to sift interesting from non-interesting events. The latter ones are rejected. Since the event rate is too high to directly perform a full event reconstruction, a system of triggers selects interesting events in order to decrease the event rate before a full reconstruction is performed. A trigger rejects or accepts an event dependent on whether it satisfies a set of required physical or geometrical properties. The triggering process is usually divided into a number of steps, called trigger levels, that each perform a (small) part of the event reconstruction process. The quality of a trigger can be determined with the following parameters: (1) the rejection power of a trigger, defined as the number of rejected events divided by the number of incoming events, and (2) the efficiency of the trigger, defined as the number of accepted interesting events divided by the number of interesting events. The trigger decision latency is defined as the time required to make a rejection/acceptance decision. A trigger of good quality has a good rejection power and a high efficiency. For a given efficiency, the rejection power of the trigger software depends on the type and quantity of data used as input and the discriminating power of the trigger algorithm. A higher discriminating power of the algorithm usually results in an increased trigger decision latency, which requires more buffer space, since the required buffer size for events that are being triggered is proportional to the trigger decision latency.

For the CMS experiment, a three level trigger system is foreseen. The level 1 trigger should reduce the event rate from 40 MHz to 100 kHz. These high rates require an implementation of the trigger in hardware. If an event passes the level 1 trigger it is written in parallel into 1000 read-out buffers, where every read-out buffer contains part of the event. The level 2 and 3 triggers are foreseen in software and are executed by a 1000 node CPU farm. This CPU farm is connected to the 1000 read-out buffers by means of a fast switching network. A central event manager controls the switch. The rejection power of the level 2 trigger should be such that only one out of ten events pass level 2. The level 2 trigger should realize this by only using 10-25% of an event to take a rejection/acceptance decision. The level 2 trigger will probably use data of the calorimeters or the muon system, since these produce less data than the tracker. The throughput requirement for level 2 is that it should deal with an input rate of 100 kHz. The maximum level 2 decision latency depends on the size of the circular detector read-out buffer that is used to avoid that data are overwritten. Currently a read-out buffer of 250 MBytes per read-out channel is foreseen that allows a maximum level 2 latency of 2.5 sec. The level 3 trigger considers the complete event and can therefore be a (almost) full event reconstruction. There is no important latency requirement, as the memory associated with the CPU farm is not a scarce resource. The level 3 trigger should bring the event rate down from 10 kHz to 100 Hz, since 100 Hz corresponds to 1 PetaByte per year, which is the maximum amount of data per year that can be stored on mass storage. The data acquisition can be considered as a production data flow. Further reconstruction occurs offline.

Control

Control consists of three sub-flows:
Section 4.3: Data flows in the CMS experiment

- Slow control. Control and monitoring of slowly changing settings and parameters of the detector, like temperatures and supply voltages of the individual detector elements. This system has very strict safety aspects.
- Run control. This is the monitoring of the status of the data acquisition (e.g. trigger rates).
- Quality checking system. Automatic and interactive tools for checking the quality of the data written to secondary storage. It comprises gathering of statistics (e.g. histograms) about data and reconstructed data and direct (visual) analysis of interesting events.

Control is of course considered as a control/interactive data flow.

Recalibration

There is a need to update the calibration constants, since the state of the detector is subject to changes. Examples of changing parameters are zero currents of AD converters, or changes due to temperature. Updating the calibration constants is done as follows:

1. During a recalibration, instead of gathering normal event data, the detector produces so-called special events. An example of a special event is a minimum bias event, which is an event that is filtered by trigger level 1, but not by trigger levels 2 and 3.
2. By processing the special events of which the characteristics are known, the new calibration constants can be computed. Processing the special events is done using a reconstruction program that is slightly different from the one used for normal event data.

The reconstruction of a given event uses the set of calibration constants that were computed using special events that were produced shortly before the given event was produced. Historically, event reconstruction has been performed offline and therefore also updating the calibration constants could be performed offline. For the CMS experiment, recalibration has to be performed online, as it is foreseen that event reconstruction will be online. This requires the immediate availability of the special events and it imposes a time constraint on the computation of the calibration constants. By choosing the level 3 trigger not to be a full reconstruction process, it does not depend on the latest calibration constants and consequently, the time constraint on the calculation of the calibration constants can be weakened. If the detector parameters change slowly, the calibration constants will change slowly as well, which means that recalibrations will not have to be performed frequently. For the CMS experiment, recalibration intervals of hours are foreseen. Online recalibration can be considered as a production data flow.

Offline event re-reconstruction

The level 3 trigger considers the complete event and performs a (almost) full event reconstruction. This is called the pass 1 reconstruction. However, accepted and stored events are also reconstructed offline. Such a reconstruction is called a pass N (with N>1) reconstruction. This reconstruction is usually more detailed than the online reconstruction. Re-reconstruction of data is done for two reasons: (1) improved event reconstruction software and (2) optimization of calibration constants. During the lifetime of the experiment, the characteristics of the detector become better known, which allows improving the event reconstruction software and consequently, improving the accuracy of the reconstructed data. Calibration constants can be improved in an iterative process by comparing properties of reconstructed events of a given well-known type as determined by the experiment with their properties as established in other earlier experiments. This comparison gives information about the accuracy of the calibration constants and might allow improving them.

Often just a part of the reconstruction process will be repeated, for example if only that part of the reconstruction program or calibration constants has improved. This means that events with
properties satisfying sophisticated selection criteria, specific parts of events, or specific intermediate reconstruction results need to be efficiently retrieved. Re-reconstruction can be considered as a production data flow.

All the different ways in which event reconstruction software is used do not only result in requirements for data access, but also for the software itself: the software has to be easily reconfigurable. Changing the order in which tasks are performed, replacing a task with another version and executing only part of the reconstruction software should all be easy to establish.

Analysis

Analysis is the stage in the data processing that comes after the reconstruction. The border between reconstruction and analysis is not so clear. Usually, analysis is defined as the part of the data processing that only considers high level objects (e.g. particles, tracks and jets). Determination of the non directly detectable particles like the W, the Z, the Higgs boson or the top quark are defined as part of the analysis. A major activity in the analysis is to determine the event sample containing interesting events for the analysis under consideration. This is an iterative process to refine the selection criteria for the event sample based on the results of the previous cycle. Analysis is done by physicists. The analysis uses event visualization and statistical methods (e.g. comparing the results with Monte Carlo simulations) as its tools. Analysis can be considered as a production data flow.

Simulation

Simulation (e.g. Monte Carlo) is a very important part of a HEP experiment. Simulation is used in the design and building phase of the experiment: for the design of the detector and for the testing of software. Simulation also provides predicted results that can be compared with the measurements. The amount of data used for simulation is considerable (in the order of Tera-Bytes). Simulation can be considered as a production data flow.

4.4 Parallelism and HEP data processing

In this section the needs and possibilities for parallelism in the CMS experiment or any other HEP experiment are discussed.

Needs for parallelism

The data acquisition system has very demanding throughput and functional requirements for all trigger levels. These will be satisfied by event farming using about 1000 instances of the programs. The data acquisition is designed such that strong latency requirements are avoided. The reason is that latency requirements are considered to be difficult to satisfy, therefore the strategy is to use more resources to relax latency requirements. The level 2 trigger is a nice example: the system contains 256 GBytes (1000*256 MBytes) of read-out buffer memory to relax the latency constraints to 2.5 sec. This has to be in-house designed dual ported memory, because every buffer has to transfer data at 400 MBytes/s. The argument for this enormous and very expensive amount of buffer memory is that by the time that the CMS detector will be operational, memory will be cheap. This still has to be seen, considering the read-out buffer design problems that are currently experienced: initially a data transfer rate of 100 MBytes/s was foreseen, but this had to be increased to 400 MBytes/s. In the context of an amount of CPU power that is tuned to meet the throughput requirement, a latency decrease by means of parallelization can decrease the amount of required read-out buffer memory and still respect the functional requirements of the trigger without using more CPU power! The reason that latency requirements are considered to be hard to satisfy, is the lack of experience with proper techniques to
do this. However, if we show the viability of techniques to decrease the latency, future applications need not circumvent strong latency requirements at all costs and consequently, save resources.

Current and future HEP experiment data acquisitions have strong latency requirements which are met by applying parallelism. ATLAS is one of the other future LHC experiments. Its level 2 trigger has a strong latency requirement of 10 ms per event [ATLAS94]. This trigger distinguishes so-called Regions Of Interest (ROIs) of an event, which are the regions of the sub-detectors that detected many passing particles. 5 ROIs per event are expected. The latency requirement will be met by processing in parallel every ROI for each of the 4 sub-detectors, resulting in 20-fold sub-event parallelism.

Another example [Fortner96] of a strong latency requirement can be found at the D0 experiment. The D0 experiment [DZero94] at Fermilab in the USA has a level 2 trigger with a 10 kHz input rate and a 1 kHz output rate. This trigger has the requirement that the order of leaving of events should be equal to the order of arrival (this is a property that we see often in HEP and also in other data acquisition systems). Since the event size distribution has a large variance, and therefore also the level 2 trigger execution time per event, the ordering requirement would result in much processor idle time (i.e. many load imbalances) if farming was used. Therefore, track finding in the muon preprocessor has been parallelized using 2400 processors that work in a SIMD fashion. The parallelization is based on dividing the (translationally symmetrical) detector into volumes. Each processor is assigned a volume and each processor does the same calculation for every event. This results in a track identification time that is independent of the event size. For the currently expected luminosity, a speedup of a factor 10 is expected in comparison to a sequential computation.

We saw above that the HEP experiment control data flow involves human interaction. The interactive analysis and visualization can benefit from a latency reduction. The other control flows require real-time behavior, but the small amount of data that is involved does not require the application of parallelism.

Above we discussed the possibility of online recalibration. For the CMS experiment, it is foreseen to compute the calibration constants online on the 1000 node CPU farm. To have the calibration constants available quickly, the computation latency will be reduced by computing the calibration constants on several nodes in parallel.

Since re-reconstruction and analysis are offline tasks, they do not have to show real-time behavior. Therefore coarse grained farming seems to be the best way to reduce the processing time, that can be considerable because of the large amount of data that is involved.

Possibilities for parallelism

The energy cluster finding task is independent of the track finding task and therefore these tasks can be done in parallel. The track fitting task uses both tracks and energy clusters. Consequently, the possibility for further parallel branching comes to an end, and it means as well that the energy cluster finding task and track finding task are synchronization dependent. Also within a task there is room for parallelism. Finding of tracks can be done in parallel and finding of energy clusters can be done in parallel: the (symmetrical) detector can be divided into volumes and for all volumes the same search algorithm can be performed in parallel. The fitting of trajectories to tracks can be done in parallel for different particles. Stok [Stok96/2] and Schiefer [Schiefer95] look at possibilities for parallelism in event reconstruction.

The execution time of a task depends on the job or subjob under consideration and is therefore variable. This complicates the load balancing if (task) parallelism or geometric data parallelism is applied. Examples of subjobs are the finding of tracks, finding of energy clusters and fitting
of trajectories to tracks. A task rejects the complete event if the part of the event it is considering, does not satisfy some set of criteria. This is an example of conditional communication and complicates load balancing.

To reduce the computation latency of the calibration constants farming of the (independent) special events can be applied.

**How can the parallelism be established?**

Parallelizing compilers have been developed, but the mediocre performance, caused by the currently still existing disadvantages of implicit parallelism, prevented their wide application in HEP [Fox94]. Explicit parallelization at the module level can be more successful, because for HEP software applications, it is possible to discern relatively large modules that exchange relatively small data sets, which results in a relatively small communication overhead (depending on the used technology). The remainder of this thesis will focus on how to obtain an application parallelized at the module level in an explicit (the word explicit is not mentioned every time) way in the context of event reconstruction programs.

### 4.5 Data management and HEP data processing

As mentioned above, there is a need to efficiently select from a very large data sample (1 PetaByte/year) events, specific parts of events or specific intermediate reconstruction results with properties satisfying sophisticated selection criteria. This requires structured data storage and access. A currently investigated solution is to store raw and reconstructed event data into a commercially available distributed OO database combined with a hierarchical storage system. This can provide the experiment collaborators that are spread around the world with a consistent view on the data and allows easy data access. Replication of data at the distributed sites is necessary to deal with network bandwidths that are low with respect to the amount of data.

Data structuring techniques, as for example provided by databases, can be of use for the data flows mentioned earlier. Until now the data flows were implemented and managed separately resulting in different structuring techniques. Raw data, which were never deleted or updated, were stored on tape, because of their enormous size. The high rate of the raw data production could not be handled by databases. Reconstructed and simulation data were also stored on tape, although their lifetime was less than that of the raw data, but they were too big to store on disk. An exception was OPAL [OPAL91] that had a set of reconstructed data on disk. Calibration constants, slow control data, and the detector description were stored in databases, since these data were updated and version control was necessary. The slow rate of update and insertion and the modest data sizes allowed the use of databases. A possibility for the LHC is to integrate some of the data flows by using the same data structuring technique. For example, event data and calibration constants can be stored in one database. A database offers support for association of events with the correct calibration constants and version control of the calibration constants.

Processing of raw data produces reconstructed data. Processing of reconstructed data produces analysis results. A question is which result data should be stored and which result data should be recomputed on demand. The answer to this question is determined by the volatilility, the computing power required for recomputation, and the storage requirements of the data in question. This means that the answer also depends on CPU speeds, network bandwidths, and storage capacities and speeds. Historically, raw and reconstructed data have been stored on tape, because of their size and non-volatility, and because disks were relatively slow. Also intermediate results of the analysis were stored on so-called mini data summary tapes (MiniDSTs), because of their size, although they were more volatile. This resulted in the situation where
Section 4.6: Hardware independence and HEP data processing

analysis was done with data of which the conditions of production were not under one's own control. Moreover, changing compilers, changing code, changing hardware and changing calibration constants made it impossible to recompute certain data. These disadvantages favor the option to only store raw data and to recompute results on demand, since this allows one to use the latest calibration constants and software versions. This option however, requires more care for code version management, as for recomputing results, the production environment (i.e. compilers, hardware platforms, code, calibration constants) has to be reproducible. Whichever option will be chosen, version management of code and data is important. It is believed [CMS96] that (OO) databases can be of help. Calculations [CMS96], using predicted values for computing power, storage capacity and network speeds for the time that the LHC will be operational, showed that for the LHC only storing the raw data and recomputing all other data on demand is not feasible. However, the use of parallelism might improve the feasibility of this option.

The need of data structuring techniques in many areas in HEP has been recognized in the past. The HEP community has a long standing tradition of programming in the Fortran programming language, but Fortran does not provide abstract data types or dynamic variables. Zebra [CN94], an extension to Fortran to supply abstract data types, is widely used, but contains drawbacks:

- A large overhead is generated when a Fortran program selects data from Zebra structures.
- Zebra is based on arrays with index counting that is visible to the application programmer. It is difficult to ascertain correctness of created Zebra structures, because of their complexity. Zebra supports no checking on reference errors.
- Zebra is very low level and therefore does not provide support on how to structure the data. As there is no generally accepted method to structure data in Zebra, the reusability of software is diminished.

These disadvantages resulted in a focus on OO programming. Software should be developed in a collaboration wide OO development framework, leading to real reuse of software. This should result in so-called component software with plug-in modules that allow easy replacement of a version of a software module with a newer one.

### 4.6 Hardware independence and HEP data processing

HEP software applications usually have a lifetime of decades. The quickly evolving hardware technology requires hardware independent, easily portable software. Secondly, considering the long development time of large software applications, hardware independent software allows HEP to postpone the hardware choice for as long as possible. New HEP experiments tend to use commodity hardware and software. Using commodity components results in systems in which components can be easily upgraded to more recent, better performing ones. This increases the lifetime of the systems. Where possible, industrial hardware platforms and standardized software technology are used. In-house hardware and software are developed only for applications where no industrial effort is going on. In-house hardware and software have long development times, often lack portability, inhibit exchange and complicate upgrading to newer better performing versions, and therefore cannot compete with industrially supplied hardware and software.
Chapter 5: A programming model for HEP: CoCa

5.1 Introduction

This chapter combines the information of chapter 2 about parallel computing and chapter 4 about high energy physics and proposes a programming model, called CoCa (Communication Capability), that supports parallelization of HEP event reconstruction software. Seen from a software engineering point of view, this chapter summarizes the requirements, provides a specification, and then presents the architectural model. Presenting an architectural model involves taking decisions based on the requirements. In this chapter we aim at keeping the level of detail such that all the requirements that resulted from the discussion about HEP event reconstruction are covered. This allows us to base the chapter on design entirely on the specification and architectural model presented in this chapter. The CoCa programming model will be validated by means of test cases presented in chapter 8.

The CoCa programming model supports component level parallelization as a technique to obtain a parallel application. The CoCa communication model is part of the programming model and offers the required inter-component communication capability. In chapter 4, needs and possibilities for parallelism in HEP event reconstruction applications have been discussed and the importance of data structuring and hardware independence has been mentioned. Furthermore, it was shown that HEP event reconstruction software has many of the properties mentioned in chapter 2. In section 5.2 some of the properties of HEP reconstruction are discussed, which results in a specification of the programming model. The properties of HEP reconstruction are dealt with in more detail in section 5.3, resulting in requirements for the communication model. The architectural model is presented in section 5.4.

5.2 Requirements for the CoCa programming model

Parallelization

We showed that HEP event reconstruction software has the properties mentioned in section 2.3 which means that explicit component level parallelization is an appropriate way to satisfy the performance requirements. Most of the features of component level parallelization are already thoroughly addressed in chapter 2 and will not be repeated here.

Data structuring

It has been mentioned that data should be structured to obtain effective and structured data access. Two aspects are important:

- Efficient data access by components of the system.

  This aspect will be addressed in the discussion about the communication model.
• Efficient data access by the environment.

Some component results are results of the system as a whole. These results should be (physically) stored in such a way that they are efficiently accessible to the environment. The goal is that results can be used by the environment without (a physical and therefore time-consuming) reorganization. This is independent of whether the hardware platform has NUMA or not. Some examples of environments are:

i. a disk, which suggests that results should reside in memory in large chunks of a standard size.

ii. a relational database, which suggests results to be structured according to an appropriate relational database scheme.

iii. an OO database, which suggests results to be structured in objects according to the database scheme.

iv. another application using the system results to perform more data processing, which suggests results to be structured such that they are readily usable by the other application.

Hardware independence

Combining hardware independence and good performance is difficult, since hardware independence does not favor taking advantage of specific features of the hardware platform. The working hypothesis is that hardware independence combined with a good performance can be obtained by separating the realization of a parallel program into a programming phase and a configuration phase. In the programming phase, the code modules are programmed without taking features of the hardware into account, i.e. the application code contains no information about the target hardware platform. This should guarantee hardware independence of the parallel application. In the configuration phase, the module/component mapping and the component/processor mapping are determined. This configuration information is not part of the application code and is presented to CoCa separately to establish these mappings.

The separation of the application code and configuration information should provide an increased flexibility. The configuration can be changed without changing the code modules, which allows fast testing of different configurations. The code modules can be changed without changing the configuration, which allows easy integration of (small) changes and improvements of the code.

The division into two phases supports the realization of the application to be an iterative process where programming and configuring alternate. After the programming of the modules has finished, a configuration has to be found that is adequate in terms of throughput, latency and resource utilization. It is not possible to find a good configuration for every decomposition of the application into modules. Testing of different configurations provides feedback to the programming phase and possibly results in an adaptation of the decomposition. In that case an adequate configuration has to be found for the adapted decomposition.

Furthermore, the separation of application code and configuration information should facilitate the adaptation to a changed hardware configuration and the porting of the application to other hardware platforms. To support the portability to other hardware platforms, the interface of CoCa towards the parallel application should be the same for all envisaged target hardware platforms, to prevent that changes to the application code are required. Of course, the configuration may need to change when the application is ported to another hardware platform.

Because CoCa has to be available on all target hardware platforms where the parallel application is to be ported, portability of CoCa itself is also important. By basing the interface between CoCa and the native communication software delivered with the hardware platform on only a
5.3 Requirements for the CoCa communication model

The previous section focused on the general features of the programming model. This section discusses the consequences of these general features which result in requirements for the CoCa communication model handling inter-component communications. Furthermore, the properties of HEP reconstruction software and the environment are treated in detail and are related to what has been discussed in chapter 2, which results in more requirements for the communication model.

Single and multiple reader access

We showed that the main data flow in HEP is a production data flow. An aspect of a production data flow is that the read/write ratio is close to one. However, the presence of the other control/interactive data flows results in part of the data being read more than once. This means that for the bulk of the data single writer/single reader (SWSR) access is required and for the rest of the data single writer/multiple reader (SWMR) access. The decision, which of the potential consumers consume a data item, depends on information that is only available at runtime. This runtime decision should be based on information that is available locally, to avoid excessive network traffic. Consequently, it cannot be the producer that determines the set of consumers of a data item, because the producer does not have the required runtime information locally available. Here we introduce the term job space, which is the set of jobs that are in the system at a point in time and that are not (yet) being processed. Flexibility is obtained if a consumer has the possibility to access all jobs in job space, but decides itself whether it wants to consume a job or not. A broadcast mechanism can offer this functionality, but it only implements SWMR access: all consumers receive all jobs and it is difficult to assure that a data item is consumed by only one consumer. This means that implementation of SWSR access based on broadcasting would cause much administration overhead. Furthermore, it is difficult to build broadcast mechanisms with little synchronization and communication overhead on distributed memory computers [Tanenbaum92].

Repetitive behavior

Since the input is a stream of similarly structured events, the HEP reconstruction application has a very repetitive behavior. This repetitive behavior leads to many jobs of the same type and a repetitive inter-component communication pattern, which means that for every cycle a component follows the same communication pattern and produces or consumes jobs of the same type.

This suggests that the search space of jobs that are potentially interesting for a consumer can be limited by logically structuring jobs according to their type: the job space is divided into subsets of which all members have the same type. This is possible since there is only a limited number of known job types. A consumer now only has to choose between jobs that are in the subset corresponding to the required type.

Event ordering requirement

HEP reconstruction often requires that the order of leaving of reconstructed events to the environment is equal to the order of arrival of raw events from the environment. This means that jobs, but also the subjobs communicated between the components, will usually be retrieved
from job space in the same order as they arrive in job space. To improve the performance of this data access pattern, jobs of one type can be logically ordered according to their arrival time in job space.

**Data rejection**

Event rejection in HEP reconstruction is an example of conditional communication: a component has or has no output dependent on an acceptance or rejection decision. For a parallelized reconstruction application, event rejection might also be an example of concurrent solutions: multiple components try to make a rejection decision in parallel. Consequently, HEP can benefit from a global stop option (feature 3, section 2.5.4). This subject will be addressed in chapter 7, the design.

**Non-deterministic temporal behavior**

We have seen that HEP event reconstruction has all the properties that were mentioned in section 2.4.2.1, titled “deviation from deterministic temporal behavior”. Consequently, HEP can benefit from dynamic communications (feature 1, section 2.5.4), to allow dynamic load balancing to obtain a good processor utilization and HEP requires measures for correct recombination of related data (feature 2, section 2.5.4).

The dynamic communications feature allows the job destination to be decided at runtime. This makes it possible for the job destination to depend on information that is only available at runtime, for example the job’s content, type or age, or information regarding the various component loads. Moreover, the dynamic communications feature avoids that the application contains code to determine a job’s destination that is configuration dependent.

Correct recombination of related data will be addressed in chapter 7, the design.

**Performance requirements imposed by the environment**

- Superfluous copying of a job should be avoided, since this is time consuming.
- Communication times under CoCa should, as much as possible, be independent of the size of the job space, i.e. number of jobs, to allow scalability.

**5.4 Architecture of the CoCa communication model**

The CoCa communication model is based on communication via the physical counterpart of the abstract term job space as used before: the set of jobs that are not yet being processed and that are stored in main memory that is accessible to all components. This physical concept, that we call job space from now on, is the basis of the architectural model.

A job is a piece of memory with a given content. During the lifetime of a job the content can change. A job in job space can be selected by a component after which the component can access the job. CoCa has six primitives: \( \text{rd, in, out, release, create and destroy} \) that can be invoked by components. These primitives work on one job at a time without influencing the state of other jobs. Dependent on the order in which primitives are executed, a job can be in the following five states: \( \text{in job space, in job space and read, assigned, assigned and read, and read} \). The job state diagram shown in figure 13 shows the states of a given job and the state changes caused by the primitives.

A component can create a job, after which the job is in the assigned state. Subsequently, the component to which the job is assigned, can define the job content. By invoking an \( \text{out} \), the job can be inserted in job space without deciding on the job’s (physical and logical) destination; this is taken care of by the CoCa communication model. After insertion, the job is in job space,
which means that no component (including the producer) can access the job, but all components can select the job. The inserted job itself contains no direct information about its sender or destination.

A consumer does not have to know where to find a job with certain characteristics; this is done by the CoCa communication model. If a consumer needs a job, it can select one from job space by invoking an in or a rd with a predicate that is based on the job’s type, content and age. The predicate can be used by the consumer to select an appropriate job. An in searches for a job in job space satisfying the predicate, and if it finds one, takes it out of job space and assigns the job to the consumer. The consumer can now read or write to the job and other components cannot select it any more. Consequently, the out together with the in primitive provide SWSR access.

A rd does the same as an in, except that it does not take the job out of job space. A job selected from job space with a rd is in state in job space and read, which means that the job can be accessed by the consumers that selected the job with the rd, but other components can continue selecting the same job. Consequently, the out together with the rd provide SWMR access. The consumer indicates that it has finished reading the job by invoking the release primitive. The rd and release operations are paired, which means that when the number of release operations is equal to the number of rd operations, the number of reading consumers of a job is zero. After the last reading consumer has invoked a release, the job is in state in job space.

A job in state in job space and read can be selected until a component performs an in which takes the job out of job space, after which the job is in state assigned and read. The selected job is accessible for reading and writing to the component that invoked the in and remains at the disposal of the reading consumers of the job. The job cannot be selected any more, so the number of readers cannot increase. The current reading consumers indicate that they have finished reading the job by invoking release primitives. After the last reading consumer has invoked a release, the job is in state assigned. An out on a job in the assigned and read state brings a job back in the in job space and read state, possibly with a modified content with respect to its previous stay in job space.
Components that select a job by means of a \textit{rd} or \textit{in}, should all see the job content as defined by the component that executed the last \textit{out} preceding the \textit{in} or \textit{rd} actions. This has been accomplished by (1) allowing a component that selected a job with a \textit{rd} to read the job, but not to write to it, and (2) by measures that are different for implicitly or explicitly accessed jobs and that are discussed later in this chapter.

The introduction of the \textit{release} primitive allowed us to implement the concurrency as described in the above model for implicit as well as explicit access with a reduced overhead, as will be explained later in this chapter.

A job that is \textit{assigned} to a component is destroyed by that component by invoking the \textit{destroy} primitive. In general, if a job becomes superfluous, it will be destroyed by a component. If a job in the \textit{assigned} state is destroyed, the memory associated with the job is given up and no primitives can be invoked to work on that job. If a job in the \textit{assigned and read} state is destroyed, it will be in the \textit{read} state, which allows reading consumers to finish accessing the job. On a job in the \textit{read} state only \textit{release} primitives can be invoked. Upon the last \textit{release}, the memory associated with the job is given up.

The only way of taking a job out of job space is by invoking the \textit{in} primitive. The reason for not having a special remove primitive is that all jobs are part of the production data flow, which means that they will all be selected by the \textit{in}. The introduction of a state \textit{in job space and read} in the model allows all jobs in job space to be permanently selectable by the \textit{in} primitive and therefore ensures an undisturbed production data flow. A model in which consumers reading a job would make a job non-selectable for the \textit{in} primitive, could disturb the production data flow by interactive/control data flows that invoke \textit{rd} primitives. The requirements did not show the need for an update primitive. As we will see later, this has the advantage that replication can be used without increasing the overhead to maintain consistency.

The CoCa method of communication makes a producer unaware of the job's consumer(s) and consumers unaware of a job's producer. It is a very flexible system, since a consumer decides at runtime which job it wants to select, or in other words, a job's destination is determined at runtime. Furthermore, the predicate system allows a consumer to decide at runtime \textit{what} job to select. Summarizing, sender/receiver pairs are determined at runtime: dynamic communications are established.

The CoCa communication model is data driven: a component will only start processing if all its inputs are available. No other features are provided (e.g. pure synchronization primitives) that can prevent a component from processing a job if all its inputs are there. Component synchronization is solely determined by the (un)availability of data. Since multiple components access the job space concurrently, concurrency control is required.

**Logical structure of the job space**

The job space is structured according to the requirements discussed in the previous section. The reason for logical structuring is to decrease search times for jobs in job space. The logical structuring is as follows:

1. The complete set of jobs in job space is divided into subsets according to a job's type.
2. Within a subset of jobs of the same type, jobs are ordered according to their arrival order (i.e. a job's age).

**5.4.1 Physical aspects of the job space**

Physical location transparency of jobs to the components is a desirable property of the communication model, since it helps to establish hardware independence. A producer component
Section 5.4: Architecture of the CoCa communication model

invoking an out should not have to decide on the job’s physical location. Similarly, a consumer invoking an in or rd should not have to know the physical location of the job to be selected, but should just specify what job it wants to select. The communication model addresses the question where jobs should be physically located and where jobs with certain properties can be found. Some of the physical aspects of CoCa are discussed in this section.

Optimizing the communication performance: reference and move access

Physically moving of jobs incurs overhead and should therefore be avoided whenever possible. In section 5.2 it was mentioned that CoCa should be available on shared and distributed memory computers, which means that we have to deal with both implicit and explicit data access. We will optimize the communication performance for implicitly and explicitly accessible data separately. Furthermore, we will introduce replication to allow concurrent access by components that are reading and writing to the job, and to allow concurrent job access by distributed components. Replication means that multiple copies of a job exist that we will call job copies. The states of the job copies constitute the state of the job: (1) the sum of the readers of the job copies of a job is equal to the number of readers of the job, (2) if a job is in the assigned and read state, one component has a reference to one of the job copies, and (3) if a job is in the in job space and read state, one job copy is referenced from job space. A CoCa primitive acts on one job copy.

If a component accesses data implicitly, a communication can be performed by exchanging only references, with the result that a job is not physically moved. This way of communication will be called reference access:

- An out provides the job space with a reference to the job and removes the component’s reference to the job.
- A rd provides the component with a reference without removing the job space’s reference to the job and increments the number of readers of the job by one.
- An in on a job with state in job space provides the component invoking the in with a reference and removes the job space’s reference to the job. For the in on a job with state in job space and read, replication of a job is introduced to ensure that readers see the job content as defined by the component that performed the last out preceding their rd operations, while reading and writing components can concurrently access the job. Due to the replication mechanism, an in on a job with state in job space and read can find a job copy that is or that is not referenced by readers. An in on a job with state in job space and read that finds a job copy that is not referenced by readers, is similar to an in on a job with state in job space. An in on a job with state in job space and read that finds a job copy that is referenced by readers, removes the job space’s reference to the job copy, makes another job copy and supplies the component invoking the in with a reference to the new job copy. Making a copy of a job can be very time consuming in comparison to exchanging references and therefore slows down the in primitive, but we should keep in mind that for only a part of the jobs SWMR access is required and that the alternative is to wait until all readers finished reading.

Another solution is to make a job copy of a job upon the first rd on that job. Subsequent rd operations first find the original job, but are redirected to the new job copy, if job copy and original job are identical. If job copy and original job are not identical, a new job copy is made. The advantage of this solution is that the in primitive is not slowed down by the copy operation. Disadvantage is that for every invocation of a rd, a job copy will be made, while in the solution chosen by us copying is not required for jobs for which the reading has finished before the in takes place.
A programming model for HEP: CoCa

- A release decrements the number of readers of the job by one and removes the reference to the job of the component that invoked the release. If the job is in the assigned and read state, the in job space and read state, or the read state, the release invoked by the last reader of a job copy also gives up the memory associated with that job copy.

In case of explicit memory access, a job has to be physically moved between the node on which the CoCa primitive is invoked, the “local node”, and the collaborating node. This way of communication is called move access. We discuss the consequences for the CoCa primitives:

- An out copies a component’s job from the local node to the destination node, removes the component’s reference to the job, gives up the memory associated with the original job, and provides job space with a reference to the job copy.

- A rd copies the job from the node on which it resides to the local node, where the component that invoked the rd is supplied with a reference to the job copy. The job space’s reference to the original job is not removed and the memory associated with the original job is not given up. In other words, the job has been replicated to allow concurrent job access by distributed components.

After a rd, only one component has a reference to the job copy. Furthermore, when the reading is finished, the job copy is not reused. This means that the job copy does not have to reside in memory that is accessible to all components. Another consequence is that the component could write to the job copy without destroying consistency, but we do not allow this, since our aim for portability requires compatibility with reference access.

An optimization is introduced to decrease the network traffic involved in correctly counting the number of readers, which is needed to ensure job space consistency. This is done by implementing a rd with move access as three actions: (1) a rd with reference access is done on the node where the job resides, after which (2) the job can be copied to the destination node, and when the copying is finished, (3) a release with reference access is performed on the node where the original job resides. The release can be immediately performed after the copying, since the requesting component is supplied with a private job copy and there is no danger of other components accessing this private job copy.

If two reading components reside on the same node, an optimization is possible by making only one copy of the job, and supplying both components with a reference to that job copy. However, because the second rd operation needs to verify whether an in has been done since the first rd operation, this optimization unnecessarily complicates the design without a viable performance improvement and therefore has not been implemented.

- An in on a job with state in job space copies the job from the node on which it resides to the local node, provides the component that invoked the in with a reference to the job copy, removes the job space’s reference to the original job and gives up the memory associated with the original job. An in on a job in state in job space and read with local readers is similar, except that the memory associated with the original job is not given up. In other words, the job has been replicated for two reasons: (1) to allow concurrent access by components that are reading and writing to the job, and (2) to allow concurrent job access by distributed components.

- A release does not have to decrement the number of readers of the original job, because of the optimization described above, but removes the reference to the job of the component that invoked the release and gives up the memory associated with the job copy.

The job space communication model implements asynchronous communication. Asynchronous communication allows better processor utilization, but means that jobs have to be buffered (p84. of [BenAri90]). Reference access allows asynchronous communication without the over-
Section 5.4: Architecture of the CoCa communication model

head of copying a job into a buffer. A producer only becomes blocked if the memory require-
ment for job space exceeds the resources. In the case of move access, the copying required to
transport the job to the destination node is equivalent with the copying into a buffer required for
asynchronous communication. Therefore, move access implements asynchronous communica-
tion without additional copying overhead with respect to synchronous communication.

Physical structure of the job space

Two reasons for physical structuring, that have been mentioned already in section 5.2, are effi-
cient data access by the environment and efficient data access by the components of the system.

Physical structuring can be divided into two issues:

1. The relative location of data in (shared and/or distributed) memory. This can be subdivided
   into:

   i. Clustering of data that will usually be retrieved together, which makes the clustering
      dependent on the data access pattern of the application. An example of a clustering
      method is to store data in memory segments consisting of consecutive addresses. This
      type of clustering is used for jobs: data items belonging to a job are stored contiguously,
      since a job is the unit of access.

   ii. Modeling of data according to the requirements of the environment to avoid a time con-
       suming physical reordering when they are to be transferred to the environment. Examples
       have been mentioned in section 5.2.

2. The fact whether data are local or non-local with respect to the components by which they
   will be accessed. This is only an issue for hardware platforms with NUMA and the goal is to
   decrease the communication overhead by decreasing the number of non-local accesses. This
   is strongly related to dynamic load balancing, which is addressed below.

Dynamic load balancing

Dynamic communication is the mechanism that allows job destinations to be determined at run-
time and therefore facilitates the realization of dynamic load balancing. We discuss implicitly
and explicitly accessible jobs separately.

When components access jobs implicitly, we choose to have fully dynamic load balancing,
since implicit access is fast and allows reference access, which means that the expected over-
head will be relatively small. The fully dynamic load balancing is implemented as follows:
when a component becomes idle, it requests a new job. We refer to this method of communica-
tion as pull mode communication. When an appropriate job is available, a reference to the job is
passed to the requesting component.

For explicit memory access, the job has to be moved from the producer node to the consumer
node. This moving is the responsibility of the CoCa communication model and is taken care of
by the CoCa access primitives. Possible methods for data distribution are:

1. Upon an out, the job is inserted into job space on the local node. CoCa knows on which
   nodes jobs of a certain type and with a given content can reside. Upon an in, the job is
   moved to the destination node with move access. This implements pull mode communica-
tion, but there is a delay between a consumer’s request and the actual reception of the job,
   which leads to component idle time and possibly to processor idle time, and hence an ineffi-
cient resource utilization.

2. Upon an out, the job is moved to a predicted consumer node with move access. The predic-
tion can be based on runtime information, e.g. a job’s type, content and age, and processor
   load information. We will refer to this semi-dynamic load balancing approach as push mode
communication. If the prediction was correct, upon an \textit{in}, the job can be accessed locally without delay. If the prediction was wrong, the job has to be transferred from the predicted consumer node to the correct consumer node, meaning that two non-local communications were needed plus that the delay of pull mode communication was experienced as well.

The data access pattern of the application determines whether consumers are very selective in their consumption of jobs, which will be visible in the discriminative power of the specified predicates. If the predicates are very selective, it is difficult for a producer to correctly predict the consumer node of a job, since a producer does not know the precise selection requirements of the consumers. This situation favors the first method. Less selective predicates favor the second method. Therefore, the application programmer can specify in the configuration information for each job type separately whether the communication should be pull or push mode. In pull mode, jobs of a given type are stored on the producer nodes of the job type, in push mode, on the consumer nodes of the job type.

For push mode communication to be successful, the strategy used to predict the consumer node should be adaptable to the application and configuration under consideration. This is established by giving the application programmer the possibility to specify a data distribution strategy (see section 3.4.1) in the configuration information, which should help to limit the network traffic and node access contention.
Chapter 6: Evaluation of hardware and OS facilities

6.1 Introduction

The hardware and operating system (OS) offer facilities to realize CoCa. The design of CoCa is influenced by the choice of the used facilities. In advance we cannot determine which facilities to use, as we do not know their performance. Documentation about hardware and OS provides some information about the relative performance of the facilities, but the design choices are also influenced by the absolute performance. Furthermore, the relative and absolute performance of the facilities depend on the hardware platform in question. CoCa’s performance depends heavily on the most frequently used facilities, therefore the performance of facilities offered by hardware and OS is assessed. As we cannot test all the offered facilities, we have to choose which ones to test. This reveals a difficult part of the design process: it is an iterative process with side steps that turn out to be dead ends. Early prototyping can be used to choose a set of facilities that allows an implementation with an adequate performance. We limit the set of tested facilities with the final CoCa design (described in the next chapter) in mind. We chose not to present the dead ends in the design process, but only the successful design path. Since we test the facilities on the SC2000 and the CS-2 only, we compromise with respect to the portability requirement, as the set of offered facilities and their performance might not be representative for equivalent hardware platforms, which influences the design choices. Section 6.2 describes the performance measurements and the results for those facilities offered by the OSs of SC2000 and CS-2, that concern implicitly accessible data. Section 6.3 focuses on facilities for inter-node communication on the CS-2, which is handled by a proprietary solution that is not part of the OS.

6.2 Programming model offered by operating systems

This section discusses some of the facilities offered by the Solaris OS running on SC2000 and CS-2. Two types of executing entities are available, UNIX processes and POSIX threads. A process has a private address space and a private execution stack. The private address space can contain global variables, which are accessible by every part of the application code the process is executing, but not accessible by other processes. The global variable lifetime is equal to the process lifetime. The private execution stack can contain automatic variables, which have a limited scope, meaning that only part of the application code can access automatic variables. Automatic variables come in existence when they come into scope and are destroyed when they go out of scope. Multiple processes can run concurrently: on one processor in an interleaved way or on multiple processors in parallel. The Solaris OS provides two methods to exchange data between processes:

- messages, which imply physical copying of data. As our focus is on implicitly accessible data, and as we aim for reference access, messages are not further investigated.
using *process shared memory*: a fixed part of a process' private address space is shared with a number of processes. Global variables that reside in the shared part of the address space are implicitly accessible by all processes, hence data can be exchanged by means of implicit communication. Participating processes have to reside on a shared memory computer or on the same node of a distributed memory computer.

The multi-threading model allows multiple threads belonging to one process to execute concurrently. Each thread has a private execution stack, but the threads of one process share the process' address space. This means that threads of one process can all access the global variables of the process. Multiple threads of one process can run on one processor in an interleaved way or on multiple processors of a shared memory computer in parallel. Thread creation and context switching between threads of the same process are much cheaper than process creation and context switches between processes, respectively. Implicit inter-thread communication between threads of one process is possible via shared global variables without additional measures. Inter-thread communication between threads of different processes can be done by means of the inter-process communication mechanisms described above. The POSIX thread standard [IEEE96] helps ensuring portability of thread based implementations.

Since we did not yet decide on the mapping of CoCa components to executing entities, we use the term *component* to commonly refer to either a process or a thread. We distinguish *component local memory*, which can be implicitly accessed by only one component, and *component shared memory*, which can be implicitly accessed by multiple components.

**Solaris synchronization primitives**

Concurrent access of components to global variables in component shared memory requires synchronization. Solaris offers four synchronization primitives that are listed in order of increasing execution overhead according to the Solaris manual: mutexes, condition variables, semaphores, and read/write locks. According to the Solaris manual, synchronizing processes incurs more overhead than synchronizing threads of the same process. We describe the primitives as implemented by Solaris.

Mutual exclusion locks (mutexes) are used to ensure exclusive access to a critical section. Two atomic operations are defined on a mutex. A component invoking `mutex_lock` either locks the mutex, after which the component has exclusive access, or it suspends itself if the mutex was locked by another component, until the mutex becomes unlocked. `mutex_unlock` can only be invoked by the component that locked the mutex. After it unlocks the mutex, the invoker loses the right of exclusive access. As nesting of `mutex_lock` operations on the same mutex is not allowed, `mutex_lock` and `mutex_unlock` come in matched pairs. The overhead involved in entering or leaving a critical section consists of the execution time of the mutex operations.

Condition variables, also referred to as monitors [BenAri90], can be used to suspend components until a particular condition is true. Two atomic operations are defined on condition variables. `cond_wait` is used to suspend a component, if the tested condition expression is false. `cond_signal` is used to wake up suspended components and is invoked by the component that changes the value of the condition expression to true. A `cond_signal` operation invoked before a `cond_wait` has no effect, i.e. the `cond_signal` operation is memoryless. It is the responsibility of the application programmer to ensure the correspondence between the value of the condition expression and the invocation of the operations on condition variables. The application programmer should ensure that testing the condition expression and invoking `cond_wait` is done under protection of a mutex, and that invoking `cond_signal` is done under protection of the same mutex. This ensures that (1) no `cond_signal` can be invoked between testing the condition expression and invoking `cond_wait`, causing the component to be mistakenly suspended, and
that (2) the condition expression is tested atomically. To avoid that a component that locked the mutex becomes suspended which results in deadlock, the cond_wait automatically unlocks the mutex before suspending itself and automatically re-locks the mutex after it has been woken up. Unlocking the mutex and suspending itself is done atomically to avoid that cond_signal can be invoked between the unlock and the suspension, causing the component to be mistakenly suspended. Waking up and re-locking the mutex is also done atomically to avoid that the value of the condition expression is changed between the cond_signal operation and the re-locking the mutex, which would mean that the component is woken up with the condition being false. After a component has been woken up and the mutex has been re-locked, it is the responsibility of the application programmer to test the condition expression again, because between the cond_signal followed by the mutex unlock, and the waking up of the components, the value of the condition expression could have been changed.

Semaphores are integer-valued variables that can only have non-negative values. Two atomic operations are defined on semaphores. sema_wait attempts to decrease the value of the semaphore by one, but suspends the process, if this would make the semaphore negative. sema_post increases the semaphore value by one.

Read/write locks are used to ensure exclusive access to a critical section for "writing" components and shared access to a critical section for "reading" components, thereby allowing more concurrency than mutexes, although they incur more overhead. Three atomic operations are defined on read/write locks. A component invoking rw_rdlock locks the read/write lock for reading, if the read/write lock is unlocked or locked for reading, and otherwise suspends itself. A component invoking rw_wrlock locks the read/write lock for writing, if the read/write lock is unlocked, and otherwise suspends itself. rw_unlock can only be invoked by a component that locked the read/write lock. If rw_unlock is invoked by a writing component, it unlocks the read/write lock. If rw_unlock is invoked by a reading component, it unlocks the read/write lock if it was the only reader.

For all synchronization primitives, if multiple components are suspended on a primitive, the order in which they are woken up is undefined. Condition variables and semaphores have similar power and expressibility, but condition variables are more structured, as they allow encapsulation of the synchronization data and associated procedures into modules [BenAri90]. The interface of the conditional variable to the remainder of the application is a set of high abstraction level procedures.

### 6.2.1 Measurements

We did not test the Solaris synchronization primitives in isolation, as the measurement results turned out to be context dependent. Therefore, we chose a context that resembles the communication of jobs between a producer and a consumer via job space, as described in chapter 5, and of which the functionality is motivated in chapter 7: we test the behavior of the Solaris synchronization primitives by using them in algorithms solving the producer-consumer problem [BenAri90]. To facilitate interpreting the measurement results, the chosen context has a simplified functionality with respect to the design presented in chapter 7.

The buffered multiple producer-consumer problem is an abstraction from many communication problems present in computer systems [BenAri90]. There are two groups of components: producers and consumers. Producers have jobs that must be passed to the consumers. To allow asynchronous communication, a buffer is used. Consumers contend for non-empty buffer locations, called jobs. Producers contend for empty buffer locations, called slots. Analogous to chapter 5, we distinguish (1) job and slot selection and (2) job and slot access. A consumer
selects a job from the buffer, accesses it, and returns it to the buffer as a slot. A producer selects a slot from the buffer, fills it with a produced value, and inserts it into the buffer as a job.

In this chapter, we do not further investigate job and slot access. As will be motivated in chapter 7, the buffer is organized as a list of jobs and a list of slots, and components exclusively access that list. As slots are treated in the same way as jobs, we focus on measurements of job selection, and for simplicity assume the availability of operations defined on the slot list: getslot to select a slot from the slot list, returnslot to insert a slot into the slot list.

The aim of the algorithms solving the producer-consumer problem is (1) to ensure that a producer can atomically insert a job into the job list and that a consumer can atomically select a job from the job list, (2) to provide short communication times and as much concurrency as possible while ensuring buffer consistency, (3) to keep the interval between the producer starting the communication and continuing the next statement as short as possible. For simplicity, we assume the availability of two operations defined on the job list, that implement the required functionality, if no concurrency had been allowed: out_ to insert a job in the job list, and in_ to select a job from the job list. They return a status indicating success or failure. The producer-consumer problem can be subdivided as follows:

- List exclusion. We allow the following concurrency: (1) only one component at a time can access the job list, (2) only one component at a time can access the slot list, and (3) the slot list and job list are independent and can be accessed concurrently.

Tools to ensure list exclusion are the Solaris synchronization primitives and mutual exclusion algorithms based on busy waiting. Busy waiting means that a blocked component continuously checks whether it can proceed. Consequently, blocked components are still assigned processor time, meaning that busy waiting during long intervals is not attractive, if multiple components run on one processor. However, the overhead involved in entering or leaving a critical section for busy waiting mutual exclusion is lower than for the Solaris synchronization primitives. We evaluate the use of Solaris primitives for list exclusion, and in section 6.2.2.8 we investigate a mutual exclusion algorithm based on busy waiting, as the overhead of the Solaris primitives appears to be relatively large.

- Empty list waiting. Consumers should wait, if the job list is empty. Producers should wait, if the slot list is empty. An empty job (slot) list results in a high probability that most consumers (producers) are contending for jobs (slots) and therefore results in a high contention on job (slot) list access. Depending on the algorithm, contention on job (slot) list access can result in contention on access to a critical section, which often deteriorates the performance. Therefore, we aim for algorithms for which high contention on job (slot) list access does not result in high contention on access to a critical section.

Tools to implement empty list waiting are (1) the Solaris synchronization primitives, (2) busy waiting, or (3) a hybrid solution. In the hybrid solution a waiting component is suspended for a short interval. Every time it wakes up, it checks if it can proceed, and if this is not the case, it is suspended for another interval. We try all three options.

6.2.2 Algorithms

This section presents some algorithms that solve the producer-consumer problem and assesses their performance. This allows us to obtain context dependent information about the Solaris synchronization primitives, for example their behavior when used in processes or threads. The algorithms are intended for components that can access component shared memory. The global variables of the algorithms can be implicitly accessed by producers and consumers. The algorithms are shown in C code.
Section 6.2: Programming model offered by operating systems

Performance test conditions

- The algorithms are tested on the SC2000. The SC2000 has a global clock, which allows us to do time measurements in producers and consumers running on different processors.
- Components are processes, unless stated differently.
- We do tests with one producer, although the algorithms allow multiple producers.
- We measure the execution time of the first communication after initialization of the processes. Therefore, data will in general not be in the cache.
- The producer is started some time after the consumers have started. This corresponds to one of the problematic situations described before: an empty job list.
- As a measurement of the contention on access to a synchronization primitive (e.g. locking or unlocking of the primitive), the execution time of an operation on the primitive inside the algorithm is compared with the execution time of the same operation in isolation. However, interpreting this measurement has to be done with care, as caching effects might improve the execution time of the operation inside an algorithm in comparison to the execution time of the operation in isolation.

How to read a table with test results

A value in a table cell gives the time in μs that the specified operation took with a standard deviation of less than 10 percent. If a table cell contains two values separated by a slash, a large variation in the measured values was observed, with the two provided values as the measured boundaries. A value in a table cell of which the corresponding operation is marked with two asterisks gives the time between the finishing of the operation with one asterisk and the finishing of the operation with two asterisks. If such a value is negative, the operation with two asterisks finished before the one asterisk operation finished. This notation allows table cells to contain intervals rather than absolute times and still provides information about the relative execution times of producer and consumer. Table 4 shows an example. Operation1 took 100 μs, operation2 50 μs, and operation3 finished 20 μs before operation_2 finished.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation1</td>
<td>100</td>
</tr>
<tr>
<td>operation2*</td>
<td>50</td>
</tr>
<tr>
<td>operation3**</td>
<td>-20</td>
</tr>
</tbody>
</table>

TABLE 4: An example.

6.2.2.1 Algorithm 1: semaphores for waiting and mutexes for exclusion

A classical solution for the buffered producer-consumer problem uses semaphores [BenAri90] (see figure 14). Empty list waiting is implemented by suspension on semaphores. The semaphore jobs, whose value is the number of jobs in the job list, implements consumer waiting. It is initialized with value 0, as the job list starts empty. List exclusion is implemented by suspension on mutexes. The mutex joblist_excl serializes the job list access. The results of algorithm 1 are shown in Table 5. Some interesting aspects are:

- A semaphore operation costs roughly 250 μs.
- A mutex operation costs roughly 50 μs. The consumer mutex_lock is relatively expensive. Since we also see the effect for the one consumer case, the explanation that all consumers try to lock the mutex after they returned from the sema_wait does not hold.
Evaluation of hardware and OS facilities

```c
sema_t jobs = 0;
mutex_t joblist_excl;

producer() {
    slottype *job;
    job = getslot();
    access(job); /* to fill */
    mutex_lock(&joblist_excl);
    out_(job);
    mutex_unlock(&joblist_excl);
    sema_post(&jobs);
}

consumer() {
    slottype *job;
    sema_wait(&jobs);
    mutex_lock(&joblist_excl);
    in_(&job);
    mutex_unlock(&joblist_excl);
    access(job); /* to process */
    returnslot(job);
}
```

**FIGURE 14:** Algorithm 1: semaphores for waiting and mutexes for exclusion.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
<th>0 cons</th>
<th>1 cons</th>
<th>5 cons</th>
<th>10 cons</th>
<th>19 cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock</td>
<td></td>
<td>54</td>
<td>55</td>
<td>45</td>
<td>46</td>
<td>44</td>
</tr>
<tr>
<td>out_</td>
<td></td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>mutex_unlock</td>
<td></td>
<td>41</td>
<td>40</td>
<td>36</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>sema_post*</td>
<td></td>
<td>130</td>
<td>240</td>
<td>270</td>
<td>230</td>
<td>230</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 5:** Results of algorithm 1.

**Cache effects**

Execution times of the operations on the Solaris synchronization primitives turned out to be rather long. This could be caused by the components residing on different processors which invalidates the processor cache. This is tested with two small experiments. The first experiment uses algorithm 1, but the components are forced to run on the same processor. We expect to see faster semaphore operations than in the previous test, because the components run on the same processor, meaning that the semaphore value can be cached. On the other hand, the components run sequentially and additional context switches are needed which might increase operation execution times. In the second experiment, a single component consecutively performs some semaphore operations. We hope to see decreasing semaphore operation times, since the semaphore value can be cached. The results of the first experiment are shown in Table 6. We observe the following interesting aspects:

- The values for the producer in Table 6 are roughly equal to the 0 consumer case in Table 5. This is as expected, because in both cases no components on other processors need to have the value of the semaphore, which makes the operation cheap.

- The sema_wait operation of the consumer is more time consuming than the one in Table 5. This is probably because the context switch time is contained in this value.

The results of the second experiment are shown in Table 7. We remark the following:

- The first sema_post in both Table 7 and Table 6 is relatively cheap (150 μs), since no components on other processors need to have the value.

- Cache effects are very clear. Not shown measurements of copying memory segments on the SC2000 also showed cache effects: subsequent copy actions of the same memory segment were roughly 10 times faster than the 1st time. These measurements also showed that the
difference between the time to access component shared memory and the time to access component private memory is negligible.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
<th>1 cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>out_</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>mutex_unlock</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>sema_post*</td>
<td></td>
<td>127</td>
</tr>
<tr>
<td>sema_wait**</td>
<td></td>
<td>420</td>
</tr>
<tr>
<td>mutex_lock</td>
<td></td>
<td>77</td>
</tr>
<tr>
<td>in_</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>mutex_unlock</td>
<td></td>
<td>29</td>
</tr>
</tbody>
</table>

TABLE 6: Results of algorithm 1 with the producer and consumer forced to run on the same processor.

<table>
<thead>
<tr>
<th>producer</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sema_post</td>
<td>150</td>
</tr>
<tr>
<td>sema_post</td>
<td>12</td>
</tr>
<tr>
<td>sema_post</td>
<td>8</td>
</tr>
<tr>
<td>sema_wait</td>
<td>45</td>
</tr>
<tr>
<td>sema_wait</td>
<td>8</td>
</tr>
</tbody>
</table>

TABLE 7: Results of the second experiment where one component consecutively performs some semaphore operations.

Resume

This first algorithm gives an impression of the Solaris synchronization primitive overhead. The cost of 500 µs for the sema_wait/sema_post combination is high.

6.2.2.2 Algorithm 2: condition variables for waiting and a mutex for exclusion

The Solaris manual advises the use of condition variables rather than semaphores, since they should give less overhead and allow better structured code. Empty job list waiting is implemented by suspension on the condition variable jobs (see figure 15). The associated condition expression is njobs<=0 with njobs initialized with value 0, since the job list starts empty. The while loop in the consumer takes care that the condition is retested when the mutex is re-locked. Condition variables have to be used in conjunction with a mutex. This mutex, called excl, implements list exclusion by means of suspension. As this mutex should be used for both consumer exclusion on job list access and producer exclusion on slot list access, buffer access is fully serialized. Consequently, algorithm 2 allows less concurrency than algorithm 1.

The results of algorithm 2 are shown in Table 8. Since we do not know whether inter-process synchronization is more expensive than inter-thread synchronization, both are tried. Column 3 shows the results for the process-based version and column 4 for the thread-based version. Some remarks are:

- The most remarkable result is the long time the cond_signal and cond_wait operations take. For the condition variable there is no difference between the process-based and thread-based version.
- Mutex operations for the thread-based version are faster than for the process-based version. If we compare the isolated execution times of mutex operations (not shown), no performance difference between the process- and thread-based versions is observed. An explana-
mutex_t excl;
int njobs=0;
cond_t jobs;

producer() {
    slottype *job;
    job = getslot();
    access(job); /* to fill */
    mutex_lock(&excl);
    out_(job);
    njobs++;
    cond_signal(&jobs);
    mutex_unlock(&excl);
}

customer() {
    slottype *job;
    mutex_lock(&excl);
    while(njobs <= 0)
        cond_wait(&jobs, &excl);
    in_(job);
    njobs--;
    mutex_unlock(&excl);
    access(job); /* to process */
    returnslot(job);
}

FIGURE 15: Algorithm 2: a condition variable for waiting and a mutex for exclusion.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
<th>process based (1cons)</th>
<th>thread based (1 cons)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock</td>
<td>mutex_lock</td>
<td>220</td>
<td>41</td>
</tr>
<tr>
<td>out</td>
<td>mutex_lock</td>
<td>50</td>
<td>11</td>
</tr>
<tr>
<td>jobs++</td>
<td>mutex_lock</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>cond_signal</td>
<td>mutex_lock</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>mutex_unlock*</td>
<td>mutex_lock</td>
<td>400</td>
<td>510</td>
</tr>
<tr>
<td>cond_wait**</td>
<td>mutex_lock</td>
<td>55</td>
<td>47</td>
</tr>
<tr>
<td>in</td>
<td>mutex_lock</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>mutex_unlock</td>
<td>mutex_lock</td>
<td>65</td>
<td>12</td>
</tr>
</tbody>
</table>

TABLE 8: Results of algorithm 2.

Resume
The algorithm has communication times of about 1 ms caused by the condition variables.

6.2.2.3 Algorithm 3: busy waiting for waiting and a lock for exclusion

Because of the large overhead of semaphores and condition variables, an implementation without has been tried: empty list waiting is achieved by busy waiting. List exclusion is achieved with a lock. We tried exclusion with a mutex and with a read/write lock. The first should give better performance, but the latter is done with read-only consumers in mind. This algorithm has two disadvantages when the job list is empty: (1) busy waiting takes up processor power, and (2) for many consumers, the contention on access to the job list lock is large, which is expected to cause long lock/unlock operation times. Figure 16 shows the read/write lock implementa-
Section 6.2: Programming model offered by operating systems

ination. The mutex implementation is similar, with all read/write lock operations replaced by mutex operations.

An attempt to implement a hybrid solution to the empty list waiting problem, using the Solaris nanosleep operation that suspends a process for a specified interval, failed, because of the peculiar behavior of this operation on the SC2000. The minimum suspension interval turned out to be 10 ms, which resulted in a relatively long service times.

```c
rwlock_t joblist_excl;
consumer() {
    slottype *job;
    bool success=FALSE;
    while(!success) {
        rw_wrlock(&joblist_excl);
        success = in_(&job);
        rw_unlock(&joblist_excl);
    }
    access(job); /* to process */
    returnslot(job);
}
producer() {
    slottype *job;
    job=getslot();
    access(job); /* to fill */
    rw_wrlock(&joblist_excl);
    out_(job);
    rw_unlock(&joblist_excl);
}
```

FIGURE 16: Algorithm 3: busy waiting for waiting and a lock for exclusion.

Table 9 shows results of the read/write lock version of algorithm 3. Interesting aspects are:

- When the number of consumers increases, the producer rw_wrlock time becomes long and unpredictable. This is in agreement with expectations. The consumers continuously lock and unlock the read/write lock to check whether the job list contains jobs. Hence, there is a wait queue that contains all waiting consumers (except the one that has the lock and is checking for jobs) and the producer at the moment it wants to insert a job. Since the order in which components in the queue are woken up is not defined, the time for the producer to acquire the lock is very unpredictable.

- We see a very long producer rw_unlock time and negative rw_unlock times for the consumer. This means that the consumer performed the rw_wrlock, the in, and the rw_unlock, before the producer finished its rw_unlock. Apparently, the read/write lock is already released by the producer, as the consumer is able to acquire it, but it takes a very long time before the producer finishes the rw_unlock command and continues with its next statement.

At the moment the producer has the lock, there is a wait queue containing all waiting consumers. The lock is not accessed any more to check its status. This implies that the producer should not have problems to perform the rw_unlock, after which the OS wakes up one of the suspended consumers. Therefore, we cannot explain the observed effect with the here presented information.

- Although we see negative times for the consumer rw_unlock, the actual communication time, which is the time between the producer starting to insert the job and the consumer ready to access the selected job, is still of the order of a few hundred μs.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
<th>0 cons</th>
<th>1 cons</th>
<th>7 cons</th>
<th>13 cons</th>
<th>18 cons</th>
<th>19 cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw_wrlock</td>
<td></td>
<td>50</td>
<td>60</td>
<td>50/350</td>
<td>70/2400</td>
<td>50/70</td>
<td>60</td>
</tr>
<tr>
<td>out_</td>
<td></td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>rw_unlock*</td>
<td></td>
<td>43</td>
<td>48</td>
<td>50/400</td>
<td>400/1200</td>
<td>700/900</td>
<td>230/1300</td>
</tr>
<tr>
<td>rw_unlock**</td>
<td></td>
<td>400/50</td>
<td>30/-230</td>
<td>1000/-160</td>
<td>-400/-600</td>
<td>500/-900</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 9: Results of the read/write lock version of algorithm 3.
Table 10 shows the results of the mutex version of algorithm 3. We see that the mutex_unlock by the consumer is very fast, but that the mutex_unlock by the producer is slow.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
<th>1 cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock</td>
<td>55/130</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>mutex_unlock*</td>
<td>40/350</td>
<td></td>
</tr>
<tr>
<td>successful in**</td>
<td>120/60</td>
<td></td>
</tr>
<tr>
<td>mutex_unlock</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 10: Results of the mutex version of algorithm 3.

Resume

An algorithm with contention on lock access is not a good choice. It results in long and unpredictable lock operation times that cause long and unpredictable communication times.

6.2.2.4 Algorithm 4: consumer busy waiting with boolean empty (version a)

In algorithm 3, the producer has problems to acquire the lock, because with a high probability the consumers access the same lock. This leads to high lock contention and long lock/unlock times. To avoid this, we separate list exclusion and empty list waiting into two parts as in algorithm 1. Most consumers do not contend for the lock, but wait for the setting of empty to FALSE by the producer. Consequently, empty list waiting is implemented for the major part as busy waiting on empty. Access to the boolean variable is not protected by synchronization primitives. Figure 17 shows the algorithm. We should show that if the job list is not empty, empty should (possibly after some time) become FALSE until the job list is empty again, otherwise deadlock is possible. If the job list is empty, empty is allowed to be FALSE or TRUE. empty can only become TRUE in the critical section of the consumer, if the job list is empty. Jobs are inserted by a producer also inside a critical section, after which the producer sets empty to FALSE. Thus, empty will become FALSE, if the job list is not empty and hence, deadlock is not possible.

```c
rwlock_t *joblist_excl;
bool empty = TRUE;

producer() {
    slottype *job;
    job=getslot();
    access(job); /* to fill */
    rw_rwlock(&joblist_excl);
    out_(job);
    rw_unlock(&joblist_excl);
    empty = FALSE;
}

consumer() {
    slottype *job;
    bool success=FALSE;
    while(!success) {
        while(empty);
        rw_rwlock(&joblist_excl);
        success = in_(&job);
        if(joblist_is_empty())
            empty = TRUE;
        rw_unlock(&joblist_excl);
    }
    access(job); /* to process */
    returnslot(job);
}
```

FIGURE 17: Algorithm 4: busy waiting on boolean variable empty (version a).

Table 11 shows its results using a read/write lock. Some remarks are:

- The rw_rwlock time in the producer is very low, which is in agreement with expectations.
- We see also that the rw_unlock time in the producer is very low. The effect that we could not explain in algorithm 3 disappeared. Comparing algorithm 3 and 4, we do not see a difference
Section 6.2: Programming model offered by operating systems

in the state of the producer and consumers at the moment of the producer rw_unlock. Hence, we cannot explain why the effect disappeared. Producer lock/unlock times are now of the order of the isolated lock/unlock times.

- As also observed in algorithm 3, the lock acquisition time in the consumer is high. As the test only had one consumer, contention caused by consumers trying to acquire the lock is not relevant. We have no explanation for this effect.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
<th>1 cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw_wrlock</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>out</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>rw_unlock</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>empty=FALSE*</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>out of while loop**</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>rw_wrlock</td>
<td></td>
<td>190</td>
</tr>
<tr>
<td>rw_unlock</td>
<td></td>
<td>70</td>
</tr>
</tbody>
</table>

TABLE 11: Results of algorithm 4.

6.2.2.5 Algorithm 5: consumer busy waiting with boolean empty (version b)

A problem of algorithm 3 was the long lock release time in the producer. Therefore, in algorithm 4, the producer has the empty = FALSE statement after the rw_unlock statement. This ensured a short producer lock release time, but consumers select a job slightly later than necessary. In algorithm 5 (see figure 18), the producer has the empty = FALSE statement before the rw_unlock statement. This should improve the communication time, but increases the probability of multiple simultaneous lock accesses producers and consumers, which might increase the lock release time of the producer. Deadlock is not possible, as outside a critical section, empty is FALSE, if the job list is not empty.

```c
rwlock_t *joblist_excl;
bool empty = TRUE;

consumer() {
    /* as in algorithm 4 */
}

producer() {
    slottype *job;
    job = getslot();
    access(job); /* to fill */
    rw_wrlock(&joblist_excl);
    out_(job);
    empty = FALSE;
    rw_unlock(&joblist_excl);
}
```

FIGURE 18: Algorithm 5: busy waiting on boolean variable empty (version b).

Table 12 shows the results of the mutex version of algorithm 5 and Table 13 the results of the read/write lock version. A processor is assigned to each of the 19 consumers. Algorithm 5 works well, as the rw_unlock time in the producer did not increase with respect to algorithm 4 and the value to leave the while loop is negative, meaning that a shorter communication time is established than in algorithm 4. Interesting aspects are:

- Execution times of operations on mutexes or read/write locks are almost equal.
- All lock and unlock times are of the order of the isolated lock and unlock times (isolated times are in the 0 consumer column).
Evaluation of hardware and OS facilities

- We see a similar effect as in algorithms 3 and 4: for both the mutex and read/write lock versions, the rw_wrlock by the consumer is slow for the 1 consumer case, but this is not the case for the multiple consumer cases. We do not have an explanation for this effect.

- The lock and unlock times in the producer are independent of the number of consumers. This also holds for execution times of other commands.

- The consumer response time improves when the number of consumers increases. An explanation is that the check frequency of the variable empty increases with the number of consumers.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
<th>0 cons</th>
<th>1 cons</th>
<th>2 cons</th>
<th>5 cons</th>
<th>10 cons</th>
<th>19 cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock</td>
<td></td>
<td>45</td>
<td>45</td>
<td>45</td>
<td>52</td>
<td>47</td>
<td>43</td>
</tr>
<tr>
<td>out_</td>
<td></td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>18</td>
</tr>
<tr>
<td>mutex_unlock*</td>
<td></td>
<td>38</td>
<td>38</td>
<td>39</td>
<td>37</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>out of while loop**</td>
<td></td>
<td>-26</td>
<td>-35</td>
<td>-33</td>
<td>-40</td>
<td>-40</td>
<td></td>
</tr>
<tr>
<td>mutex_lock</td>
<td></td>
<td>80/240</td>
<td>52</td>
<td>52</td>
<td>53</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>mutex_unlock</td>
<td></td>
<td>38</td>
<td>34</td>
<td>39</td>
<td>42</td>
<td>39</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 12: Results of the mutex version of algorithm 5.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
<th>0 cons</th>
<th>1 cons</th>
<th>2 cons</th>
<th>5 cons</th>
<th>10 cons</th>
<th>19 cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw_wrlock</td>
<td></td>
<td>48</td>
<td>46</td>
<td>44</td>
<td>47</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>out_</td>
<td></td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>rw_unlock*</td>
<td></td>
<td>41</td>
<td>37</td>
<td>38</td>
<td>34</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>out of while loop**</td>
<td></td>
<td>-32</td>
<td>-35</td>
<td>-34</td>
<td>-33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rw_wrlock</td>
<td></td>
<td>48/220</td>
<td>46</td>
<td>46</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rw_unlock</td>
<td></td>
<td>42</td>
<td>38</td>
<td>37</td>
<td>39</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE 13: Results of the read/write lock version of algorithm 5.

Resume
Algorithm 5 provides a communication time of about 150 µs, independent of the number of consumers up to 19 consumers.

6.2.2.6 Algorithm 6: busy waiting with booleans empty and noaccess

Although no negative effect was observed in the performance of algorithm 5 when the number of consumers increased, an optimization is proposed. We try this algorithm mainly to get more insight in the temporal behavior of the algorithms. We can introduce a second boolean variable noaccess that stops consumers from contending for the exclusion lock, when it is true (see figure 19). With high probability, noaccess is TRUE if a consumer has the lock and is FALSE if the lock is free. This boolean allows the consumer that holds the lock to release it without contention on the lock caused by consumers. The effectiveness of the boolean noaccess can be measured by counting how many consumers did not get out of the while loop. We call this the number of stopped consumers and denote it as 7/19 if we started with 19 consumers and 7 were stopped. We should show that if no consumers are accessing the job list, or no consumers will shortly access the job list, because they exited the while loop, noaccess should be FALSE, otherwise deadlock is possible. If an access to the job list is taking place, noaccess is allowed to be FALSE or TRUE. Any consumer that sets noaccess to TRUE will eventually access the job list and set noaccess to FALSE, allowing other consumers to access the job list. Hence, deadlock is not possible.

Table 14 shows the results of algorithm 6. We notice the following:
Section 6.2: Programming model offered by operating systems

mutex_t joblist_excl;
bool empty = TRUE;
bool noaccess = FALSE;

producer() {
   /* as in algorithm 5 */
}

consumer() {
   slottype *job;
   bool success=FALSE;
   while(!success) {
      while(!empty && !noaccess) {
         noaccess = TRUE;
         mutex_lock(&joblist_excl);
         success = in_(&job);
         if(!joblist_is_empty())
            empty = TRUE;
         noaccess = FALSE;
         mutex_unlock(&joblist_excl);
      }
      access(job); /* to process */
      returnslot(job);
   }
}

FIGURE 19: Algorithm 6: busy waiting with booleans empty and noaccess.

- The lock, unlock, and other command execution times are not different from algorithm 5 and do not depend on the number of consumers. Both observations were expected, as already algorithm 5 showed independence of the number of consumers. A comparison of the individual runs with the same number of consumers, did not show a dependence on the number of stopped consumers.

- The time it takes for a consumer to leave the while loop is slightly longer than for algorithm 5. This is probably because the condition in the while loop is more complex and because the time measurement was done just after the assignment noaccess = TRUE.

- We see the same effect as in algorithm 3, 4 and 5: the lock acquisition in the consumer for the 1 consumer case takes a long time. We do not have an explanation for this effect.

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
<th>0 cons</th>
<th>1 cons</th>
<th>5 cons</th>
<th>10 cons</th>
<th>19 cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock</td>
<td>48</td>
<td>49</td>
<td>44</td>
<td>43</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>14</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>mutex_unlock*</td>
<td>39</td>
<td>38</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>out of while loop**</td>
<td>-25</td>
<td>-25</td>
<td>-24</td>
<td>-26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mutex_lock</td>
<td>50/150</td>
<td>49</td>
<td>45</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mutex_unlock</td>
<td>34</td>
<td>40</td>
<td>41</td>
<td>46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>average #stopped consumers</td>
<td></td>
<td>0.1/5</td>
<td>1.1/10</td>
<td>10.1/19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE 14: Results of algorithm 6.

6.2.2.7 Conclusion

Remarks about Solaris synchronization primitives

Many of the differences between the synchronization variables that have been found are too small and too unpredictable to take into account for design decisions, especially since we aim for portable software. We summarize the main observations.

- The typical execution time of a (non-cached) mutex operation is about 50 μs, which is quite slow. No performance difference between mutexes and read/write locks has been observed, which is not in agreement with the Solaris manual. Semaphores are more expensive than locks. The typical execution time of a (non-cached) semaphore operation is about 250 μs.
Evaluation of hardware and OS facilities

Condition variables are more expensive than semaphores. The typical execution time of a (non-cached) condition variable operation is about 500 µs.

- For the isolated execution of any of the Solaris primitives, no difference in performance between thread- or process-based versions was found. For the execution of mutex or read/write locks inside an algorithm, a thread-based version showed a better performance than a process-based version, which we attribute to caching effects. For the execution of semaphores or condition variables inside an algorithm, no performance difference between thread- or process-based versions has been observed.

Related to the differences between process- or thread-based applications are the execution times of memory copies from/to process shared memory/process private memory (all possible combinations). Measurements showed that differences are too small and too unpredictable to take into account for design decisions.

- Cache effects are significant. The order in which operations on a synchronization variable are executed has a major effect on the execution times: the first operation in an application is roughly 10 times slower than subsequent operations. Also operations on different synchronization variables of the same type have a large mutual influence. An operation on a mutex has a caching effect on another mutex that is just as large as if the operation had been performed on the same mutex. An operation on a mutex has no caching effect on a read/write lock. An explanation of these effects is that the synchronization variables have shared data structures, e.g. global tables with the lock states, that are cached after the first operation on the lock. As CoCa components in general run on different processors, we expect to deal mostly with the non-cached execution times (which are the ones we measured).

Remarks about the algorithms

- Algorithm 5 provides 150 µs communication times and empty list waiting is established with busy waiting. Algorithm 1 provides 700 µs communication times but empty list waiting is established with suspension. Whether to choose suspension or busy waiting depends on the requirements. If communication latency is important, busy waiting can be preferable. If processor utilization is important, suspension can be preferable.

- Introduction of the boolean variable empty reduces the contention on lock access. Introduction of the boolean variable noaccess is not advantageous.

- Some of the observed effects have not been explained, but we found work-arounds.

General remarks

- We observed behavior that did not correspond with what is mentioned in the Solaris manual. Examples are (1) the large overhead of the Solaris synchronization primitives, (2) the similar performance of thread- and process-based versions, and (3) the similar performance of mutexes and read/write locks. These effects probably have the same origin. What is said in the Solaris manual is probably true for most computers, but for the computers tested by us, the expected differences mentioned in the Solaris manual are negligible in comparison to other overheads. We expect that there is a relatively large overhead caused by the SC2000 hardware or SC2000 specific features of Solaris. Consequently, design decisions only based on the relative performance differences as described by Solaris, can lead to a bad design as these relative differences turn out to be negligible compared to the absolute figures.

- To assess the performance of Solaris synchronization primitives, naively, one would measure their performance in isolation. However, the results of the measurements with the algorithms show that this would provide results that are not representative for real applications. As it is impossible to foresee and test all situations in which the primitives will be used, we
decided to test the primitives when used in a realistic example, hoping that this covers most of the situations that will occur in the actual target application. A disadvantage of this approach is that it complicates the measurements and their analysis.

6.2.2.8 The mutual exclusion algorithm

The previous measurements showed that Solaris synchronization primitives incur a relatively large overhead. A possible reason is the cache coherency maintenance overhead. Another reason could be that to ensure atomicity, operations on synchronization primitives are executed in kernel mode, which requires a (time consuming) context switch to kernel mode and back. We do not need such "strong" mutual exclusion; mutual exclusion guaranteed between components is sufficient. To exploit this weaker requirement, and also to obtain more insight in the behavior of the Solaris primitives, we decide to implement a mutual exclusion algorithm ourselves. An easy way to do this is by using an atomic test-and-set instruction [BenAri90] provided by the hardware. However, not all computers offer this operation, for example, sequential computers and many multi-processor computers based on commodity processors do not offer this mechanism. Relying on the availability of a test-and-set instruction would decrease CoCa's portability. Furthermore, the operation is very low level and therefore not directly accessible from the OS API. Therefore, we decide to use a fully software implemented algorithm based on shared variables proposed by Lamport [Lamport87], although on shared memory computers fully software based mutual exclusion is in general slower than hardware supported algorithms [Zhang96].

The Lamport mutual exclusion algorithm uses busy waiting. We expect an improved service time with respect to the Solaris primitives, because of the experience with the previously described algorithms: the implementation of empty list waiting using busy waiting instead of suspension improved the communication time of the algorithms radically. In the absence of contention, the Lamport algorithm requires only seven shared memory accesses to lock a mutex and therefore is potentially fast. Furthermore, it is deadlock free, but allows starvation of individual processes. Since the Lamport algorithm performs well in the absence of contention, we decrease the contention by using the boolean variable empty as described before.

Memory models with non-atomic read and write actions

The Lamport mutual exclusion algorithm requires atomic read and write actions. Therefore, we discuss the behavior of memory models of shared memory computers with respect to atomic read and write operations. In the strongly ordered memory model, a modification of the memory by one processor is immediately visible to other processors, which ensures atomic read and write actions. However, many shared memory computers relax the memory model to improve the performance. Some shared memory computers have so-called store buffers that connect a cache to a processor [SUN94]. Writes by the processor are written into the store buffer before they are written into the cache, although the order of writing values into the cache is identical to the order of writing into the store buffers. Consequently, there can be a delay between the time that a value has been written by a processor and the time that it is visible to other processors (the caches are always consistent with each other and with the shared memory). This means that read and write actions are not guaranteed to be atomic. This memory model is called a total store order (TSO). Another weaker memory model exists in which the order of becoming visible is not guaranteed to be equal to the order of writing.

Certain algorithms can have incorrect behavior for TSO memory models. The algorithm in figure 20 [SUN94] obtains mutual exclusion for two threads for the strongly ordered memory model, but not for the TSO memory model as can be seen by the following example: the two
threads running on separate processors each write into their store buffer that their slot of the interested array is TRUE. Subsequently, they read the slot of the interested array of the other thread from shared memory and both find FALSE. Therefore they both assume that the other party is not present and both enter the critical section.

```c
void mutex(int me /* 0 or 1 */) {
    static int loser;
    static int interested[2] = {FALSE, FALSE};
    int other;
    other = 1-me;
    interested[me] = TRUE;
    loser = me;
    while(loser == me && interested[other])
        ;
    /* in critical section */
    interested[me] = FALSE;
}
```

**FIGURE 20:** An algorithm for mutual exclusion that requires a strongly ordered memory model.

SUN recommends to use the Solaris synchronization primitives on shared memory computers to avoid situations as described above, since they flush the store buffers. Although not explicitly confirmed by literature, the memory model of the SC2000 seems to be TSO, since the Lamport mutual exclusion algorithm does not always provide mutual exclusion. The algorithm’s mutex operation times are about 25 µs on the SC2000 which is half of that of the Solaris mutex. On the CS-2 and on sequential machines, the algorithm appears to work correctly. Table 15 shows the results of the Lamport mutual exclusion algorithm tested in algorithm 5 on the CS-2.

<table>
<thead>
<tr>
<th></th>
<th>1 cons</th>
<th>2 cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock producer</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>mutex_unlock producer</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>mutex_lock consumer</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>mutex_unlock consumer</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

**TABLE 15:** Results of the Lamport mutual exclusion algorithm on the CS-2.

**Resume**

If not more than one component per processor is allocated, busy waiting to achieve exclusion is acceptable, and the Lamport mutual exclusion algorithm would be preferable with respect to the Solaris mutex, because of its better service time. Especially on the SC2000, the difference in service time between the Solaris mutex and the Lamport mutual exclusion algorithm is large. However, this large difference might indicate a third reason (besides cache coherency maintenance overhead or kernel mode execution) of the bad service time of Solaris synchronization primitives: the flushing of the store buffers. Concluding, since the Lamport mutual exclusion algorithm does not work correctly on the SC2000, and since its service time on the CS-2 is only slightly better than the Solaris mutex, we decide to use the Solaris mutex.

**6.3 Inter-node communication**

Until now we looked at OS facilities to realize communications between components that can access component shared memory. On the CS-2 also communications between components on different nodes occur, which requires the transfer of data between nodes. The CS-2 ELAN
native communication library offers the following methods of communication, in order of increasing functionality and increasing overhead according to the CS-2 manual:

- ELAN DMA transfers. A DMA transfer is a memory copy between two nodes. The processor on the receiving node does not actively participate in the communication. A subsequent DMA transfer to the same address should only be done after the data received during the previous DMA transfer are obsolete or copied to another part of the memory to avoid overwriting. This can be ensured by having the sending node wait for an acknowledgment before it does the next DMA transfer.

Signals. A signal is a special kind of DMA transfer in which no data are transported. Signals can be used to notify that an event has occurred. As a signal is not sent to a process, but to a node, all processes can catch the signal. To catch a signal, a process does a wait_signal call that blocks until the signal arrives and then clears the signal, meaning that only one process can catch it. If the signal has been sent before wait_signal is called, wait_signal returns immediately, and clears the signal. Signals sent to a node on which the previously sent signal has not yet been cleared, do not have an effect.

- ELAN Channels. Channels provide bidirectional communication between two processes residing on different nodes. Both the sending and receiving process actively participate in a Channel communication and both have to specify the size of the communicated message. As Channels are unbuffered, only one outstanding communication operation is allowed in each direction. Send and receive operations are non-blocking, but as there are operations to make them blocking, no explicit acknowledge is required. A Channel has to be explicitly set up between the two participating processes, before usage. This means that one Channel only connects two fixed processes.

- ELAN Tports. Tports provide bidirectional buffered communication between two processes residing on different nodes, which means that an arbitrary number of communications between two processes can be outstanding. Messages between a sender and receiver pair are received in the order sent. Tports do not have to be explicitly set up between two participating processes. Any two processes that set up a Tport can communicate with each other. This means that with one Tport, messages can be sent to different destinations and messages can be received from different senders. A receiver does not have to specify from which process it wants to receive a message, and consequently, a message has a tag indicating its sender. Both sender and receiver have to specify the size of the communicated message.

- Message passing libraries like PVM. These are built on top of Tports and provide a subset of the Tport functionality.

Not every distributed memory computer can be expected to have a communication library that offers the functionality of Tports. Hence, for portability reasons we discard them. We decide to measure the performance of DMA transfers and Channels for different message sizes, used in isolation. Results are shown in Table 16. If we compare DMA communication times with the Channel communication times, the relative difference between DMA and Channels becomes less important for messages of 1000 bytes and larger. Therefore, we prefer Channels to DMA, as no explicit acknowledge is needed.

Channel communication time as a function of the inter-communication interval

Measurements showed that the Channel communication times can be different, depending on the application in which they are used. This is one more example that isolated measurements of OS primitives should be considered with care. Later, as we present the results of the CoCa measurements (section 8.2), we observe also that the communication times of Channels measured during the running of CoCa are very different from the isolated communication times. We did
Evaluation of hardware and OS facilities

<table>
<thead>
<tr>
<th>message size (bytes)</th>
<th>ELAN DMA transfer (μs)</th>
<th>ELAN Channels (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (signal)</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>100</td>
<td>13</td>
<td>23</td>
</tr>
<tr>
<td>1000</td>
<td>33</td>
<td>41</td>
</tr>
<tr>
<td>10000</td>
<td>239</td>
<td>250</td>
</tr>
<tr>
<td>23040</td>
<td>555</td>
<td>575</td>
</tr>
<tr>
<td>100000</td>
<td>2366</td>
<td>2432</td>
</tr>
</tbody>
</table>

TABLE 16: Communication times of the ELAN library primitives on the CS-2.

not perform a systematic investigation of this effect, but suspected that the Channel communication times depend on the interval between two subsequent communications. Therefore, we measured the Channel communication time as a function of the inter-communication interval. The results are shown in figure 21. After an interval of 500 ms, the Channel communication time drastically increases. Further investigation showed that this effect disappears if the receiver is busy waiting instead of suspended during the inter-communication interval (shown by the results in the right low corner of the graph). The effect does not depend on whether the sender is suspended or busy waiting during the inter-communication interval, indicated by the measurement in the graph using the sleep system call. This makes the following explanation feasible. Just after a Channel communication, processes controlling the communication are active or have a high priority. After some time, these processes become inactive. Upon the next Channel communication, the processes have to be rescheduled which requires context switches.

![Graph showing communication time versus inter-communication interval.](image)

FIGURE 21: Channel communication time versus the inter-communication interval.
Chapter 7: Design

7.1 Introduction

This chapter discusses the detailed design of the architectural model discussed in chapter 5. Design decisions are based on the results of chapter 6. CoCa is a layer residing between the parallel application at the upper side and the native communication software of the parallel computer below. The upper interface of CoCa, the application programmer's interface (API), consists of the CoCa primitive calls embedded in the language in which the parallel application is programmed, the host programming language. Bindings are provided for the following host languages: C++, since this is foreseen to be the language to implement future HEP applications, C, since CoCa has been implemented in this language, and Fortran 77, since most of the existing HEP applications have been written in this language.

![Diagram of CoCa interfaces and operating systems](image)

**FIGURE 22:** The top and bottom interface of CoCa.

CoCa has to deal with different host languages at the upper side, and different architectures of target parallel computers plus native communication libraries and operating systems below (see figure 22). To facilitate CoCa’s portability, a layered design is chosen, which allows us to encapsulate target hardware specific or host programming language specific information and allows us to treat the design of the interface for the shared memory computer, the interface for the distributed memory computer and the host language bindings separately. We call CoCa for a shared memory computer *shared CoCa* and for a distributed memory computer *distributed CoCa*. Note that a user of CoCa does not see this distinction and only deals with one CoCa system.

An important decision is whether the designs of shared and distributed CoCa are overlapping or that both are started from scratch. Overlapping designs provide flexibility, since changes in the CoCa implementation only have to be made in one location in the code, but enforce the same decisions for both architectures, which might not be optimal. On the other hand, if a node of a distributed memory computer (hybrid or not) runs multiple components, the functionality as provided by shared CoCa is required. Therefore, we decide to base the design of distributed CoCa on shared CoCa. Figure 23 shows the layer structure of shared and distributed CoCa. The
reason for this particular layer structure will become clear during the discussion of the layers. Shared and distributed CoCa have the same language binding layer and local CoCa layer.

<table>
<thead>
<tr>
<th>shared CoCa</th>
<th>distributed CoCa</th>
</tr>
</thead>
<tbody>
<tr>
<td>language binding layer</td>
<td>language binding layer</td>
</tr>
<tr>
<td>local CoCa layer</td>
<td>location transparency layer</td>
</tr>
<tr>
<td></td>
<td>local CoCa layer</td>
</tr>
<tr>
<td></td>
<td>communication layer</td>
</tr>
<tr>
<td></td>
<td>physical layer</td>
</tr>
</tbody>
</table>

FIGURE 23: The layer structure of shared and distributed CoCa.

The remainder of the chapter is focused on the design of the CoCa job space and its primitives. First, the logical and physical organization of the job space is discussed (section 7.2). Next, it is presented how the job space can be accessed (section 7.3). Section 7.4 concentrates on ensuring the consistency of the job space, which is concurrently accessed by components. Section 7.5 deals with the components. Section 7.6 treats the recombination of related data. Each section is divided into two parts: first the design of shared CoCa is presented and next the required modifications for distributed CoCa are discussed. The host language bindings are discussed in section 7.7.

7.2 The job space

7.2.1 Shared CoCa

The aim of this section is to construct a job space residing in component shared memory that allows an implementation of the CoCa primitives with a good access performance. The functionality described in this section is part of the local CoCa layer. The section separately discusses the logical and physical structure of the job space. Decisions are based on the expected HEP data access pattern as presented in chapter 5.

Logical organization

To improve the access performance, the set of jobs of a given type can be logically structured. An unordered list has a job search time that is linear with the list length. A tree has a job search time that is logarithmic with the number of jobs in the tree, but inserting and removing jobs from the tree is slower than for the unordered list. Considering the expected limited number of jobs present in job space at a point in time, we decide to optimize the out and in primitives at the expense of job searching and therefore decide to use the list implementation. We use the list as a queue: references point to the tail and head of the list and jobs that arrive in job space are appended to the tail, meaning that retrieving the oldest job from the list can be done fast and in a time independent of the list length. The job queue is doubly linked to allow fast removal of jobs anywhere in the queue. Indexes allow fast searching for specific jobs, but inserting and removing jobs is more time consuming. Therefore, indexes are not used.

Summarizing, the execution time of a CoCa primitive operating on a job queue is short and independent of the number of jobs in the queue, except for job selection using a rd or an in with a predicate, which is linear in the number of jobs in the queue. Furthermore, CoCa primitive execution times are independent of the job size as reference access is used.
Physical organization

The goals of the physical organization are good access performance and good shared memory utilization. Since jobs have to be implicitly accessible to all components, they have to reside in component shared memory. Component shared memory can be treated as one homogeneous and contiguous segment. Since a job is the unit of access, we decide to store it in a contiguous segment, which is called a bucket. A filled bucket is a job, an empty bucket is a slot.

As the number of job types is finite, the number of required bucket sizes is finite. Bucket allocation can be done at compile time according to a predefined number of buckets per job type. During execution this can lead to the situation where no slots of one size are available, while still many slots of another size are available, and hence a bad memory utilization. Dynamic bucket management avoids this problem: a bucket is allocated when a job copy is created, and freed when the job copy is destroyed, leaving the memory to be used for other buckets. However, since buckets can have different sizes, dynamic bucket management can cause fragmentation. Furthermore, finding appropriately sized free segments incurs overhead.

As a compromise we use a semi-dynamic bucket allocation management: buckets are allocated at runtime when needed, but once a bucket has been allocated, it is not freed any more. This means that the number of buckets of a certain size can grow dynamically and adapt to load fluctuations, but cannot shrink. Consequently, buckets are contiguously allocated. If a job copy is destroyed the corresponding bucket is reused as a slot. This is done by means of a slot queue that is created for every bucket size and contains references to slots. This queue is singly linked, since only the head and the tail of the queue are accessed. The slot queue allows the reuse of slots without the need to search through component shared memory. The semi-dynamic bucket allocation strategy results in relatively many (few) buckets for job types with high (low) peaked load fluctuations. If the job access pattern changes completely or rarely shows high peaks, the chosen strategy could lead to bad shared memory utilization. To avoid monopolizing of the component shared memory by buckets of a given job type, which can lead to individual starvation, the variable \( N\text{bucketmax} \), that is defined per job type and that is part of the configuration information, limits the number of buckets of one size that can be allocated. Logical and physical organization of the job space are shown in figure 24.

![Diagram of job and slot queues](image)

FIGURE 24: Logical and physical organization of the local CoCa job space. An empty box indicates a slot, a filled box a job. The dashed box indicates not yet allocated component shared memory. A single-headed arrow indicates a singly linked list, a double-headed arrow a doubly linked list. Only the job queue and slot queue of job type A are shown, although every job type has them.

getslot is a function that provides the invoker with a slot of the appropriate size. It recognizes the requested job type and returns a reference to the slot at the queue head, if the corresponding slot queue is not empty; if the corresponding slot queue is empty, it allocates a new bucket in component shared memory. To decrease the allocation overhead, not one but \( N\text{buckettext} \) buckets of one size are allocated at a time. \( N\text{buckettext} \) is defined per job type and part of the config-
uration information. \textit{returnslot} is a function that returns a slot to the tail of the slot queue for later reuse.

The functions \textit{getslot} and \textit{returnslot} do not fully correspond to the CoCa primitives \textit{create} and \textit{destroy}, which is caused by the use of replication as described in chapter 5. \textit{getslot} is invoked by the \textit{create} primitive, or is invoked if a job is replicated, which is the case for an \textit{in} invoked on a job in state \textit{in job space and read}. \textit{returnslot} corresponds to "giving up the memory associated with a job copy", which is the case for the \textit{destroy} primitive and some invocations of the \textit{release} primitive. Figure 25 shows the bucket state diagram for shared CoCa (meaning that the CoCa primitives use reference access) and shows the relation between \textit{getslot} and \textit{returnslot}, and \textit{create} and \textit{destroy}.

![Bucket state diagram for shared CoCa](image)

\textbf{FIGURE 25:} Bucket state diagram for shared CoCa.

Summarizing, this semi-dynamic bucket allocation allows faster job copy creation and destruction than the fully dynamic case, since a free slot can be found without searching. Hence the slot acquisition time is independent of the job size, the number of jobs in job space and the shared memory segment size. Furthermore, it avoids fragmentation.

\textbf{Data types and their storage}

From the previous it follows that besides job data, data are required to describe the structure of the job space. We call these meta and catalog data. Furthermore, control data are required to ensure consistency of the concurrently accessed job and slot queues. Meta data describe the logical and physical structure of the job space, e.g. pointers involved in a linked list or the number of jobs in a queue. Catalog data consist of read-only data that are specified at compile or initialization time and do not change during runtime. Configuration information is part of the catalog data. Control data control the access of components to the three other types of data and consist of locks and variables describing the accessibility of the data.

These four types of data have different storage requirements. Job data have to be implicitly readable and writable by all components and therefore have to reside in component shared memory. CoCa primitives change the structure of the job space and consequently, also the meta data, which means that meta data have to be readable and writable by all components and have to reside in component shared memory. Catalog data have to be readable by all components and do not change. Therefore, they can (1) reside in component private memory, but this means that there has to be a replica for every component, or (2) reside in component shared memory. We
Section 7.3: Job space access

decide for the last solution to save memory space and because component private and component shared memory have equal access times. Control data have to be implicitly readable and writable by all components and therefore have to reside in component shared memory.

7.2.2 Distributed CoCa

The jobs of a given type are distributed among queues on nodes with at most one job queue of a given type per node. The question arises whether there are dependencies between the job queues of a given job type. The expected HEP data access pattern allows us to consider the job queues of a given job type as independent, except for the case that jobs have to be processed in the order of arrival on a global basis. As global ordering is rarely required, we decide to enforce it with special mechanisms, if needed. For the general case, we decide that the job queues of a given job type residing on each node are independent.

This decision allows us to realize the required functionality by using, for each node, a local CoCa system plus additional support for inter-node job access (see figure 26). In figure 26 the locations of the job queues are shown as function of push or pull mode communication. Both communication modes are supported by CoCa. This additional support is provided by the CoCa location transparency layer, CoCa communication layer and CoCa physical layer. The fact that the local CoCa systems for each node are independent has the following consequences for the storage of the four data types. For job data, we follow the replication and migration schemes as described in chapter 5. Every node has its own, independent set of meta, catalog and control data, that are stored and managed according to what has been decided for shared CoCa.

![Logical and physical organization of the distributed CoCa job space.](image)

FIGURE 26: Logical and physical organization of the distributed CoCa job space.

7.3 Job space access

7.3.1 Shared CoCa

Predicates

A CoCa primitive deals with one job at a time. It only considers the job queue corresponding to the requested job type, which limits the set of retrievable jobs and therefore reduces the execu-
tion time. The component's syntax determines the requested job type. Section 7.7 describes how this is done. An in or rd without predicate returns the oldest job of the job queue. An in or rd with a predicate returns the oldest job that satisfies the predicate. We describe two solutions to implement this. In the first solution CoCa once determines the set of job values that satisfy the predicate and then searches in the appropriate job queue for a job with one of these values. Another possibility is to test each job in the appropriate job queue against the predicate until a satisfactory job is found. The first solution is difficult to implement and incurs more overhead than the second for simple predicates tested on few jobs. As we do not expect very complex predicates, we decide for the second solution with the job queue walked through linearly, starting from the job queue head.

The job representation depends on the parallel application and the host language. As predicates use the job content, they also depend on the parallel application and host language, which means that predicates have to be specified in the parallel application code. To allow CoCa to deal with application dependent predicates, we introduce so-called predicate functions, which have a standard signature known to CoCa. The predicate body does not have to be known to CoCa and can be customized to the needs of the application. Predicate functions can be tested on jobs and return a boolean indicating whether the job satisfies the predicate. To specify the desired job content, a predicate function can have input arguments set by the consumer. Additional advantages of predicate functions are (1) that the predicate (function body) can be compiled, which avoids time consuming interpretation, and (2) that for function shipping in distributed CoCa the predicate does not have to be transported, but just the predicate name and arguments. A disadvantage is that all predicates must be known in advance, which means that only the predicate function input arguments may depend on runtime information. Section 7.7 discusses predicate function execution and their signatures.

7.3.2 Distributed CoCa

Intra-node job space access is completely taken care of by local CoCa. Therefore we can focus on inter-node job space access. An important design issue is location transparency: the application program should not be aware of the job location. We decide to provide a location transparency layer on top of local CoCa that is treated in the following sub-section. The mechanism for inter-node communication is explained afterwards.

7.3.2.1 Location transparency layer

The task of the location transparency layer is to provide a common interface to the language binding layer for both shared and distributed CoCa. Secondly, the location transparency layer takes care of distribution of data among the nodes of a distributed memory computer. Both subjects are discussed separately.

Common interface to the language binding layer

A common interface to the language binding layer for both shared and distributed CoCa is provided by the RPC paradigm (see section 2.5.2). As distributed CoCa is a shared nothing system because of the independency of the nodes, function shipping is the appropriate technique, according to chapter 3. The choice of RPCs implies function shipping.

The RPC mechanism of CoCa works as follows. If a component invokes a CoCa primitive, the location transparency layer determines whether the job space on the local node or on a remote node has to be accessed. In the first case, a local CoCa primitive is invoked. In the second case, the client stub on the local node (1) marshals the CoCa primitive name and arguments in a
request message, (2) sends the request to the remote node and (3) awaits a reply. The server stub on the remote node (1) receives the request, (2) unmarshals it, (3) invokes the correct local CoCa primitive, (4) marshals the result of the local CoCa primitive in a reply message and (5) sends it to the local node. The client stub on the local node (3) receives the reply, (4) unmarshals it, and (5) returns the result of the CoCa primitive to the location transparency layer, which in its turn returns it to the component that invoked the CoCa primitive.

The request/reply facility is offered by a client/server protocol that is implemented by the CoCa communication layer and that is discussed in the next sub-section. We adhere to the original synchronous RPCs to avoid complexity for the application programmer. Figure 27 shows an example of a RPC where a component residing on the local node performs a remote out of a job on the remote node. Remark that in the client stub the communication layer interfaces to the location transparency layer, whereas in the server stub the communication layer interfaces to the local CoCa layer.

![Diagram](image)

**FIGURE 27:** An example of a RPC: a remote out. The grey boxes left comprise the client stub; the grey boxes right comprise the server stub. The handling of the reply via the same route in opposite direction is not shown.

**Data distribution requirements**

The location transparency layer implements data distribution strategies that determine on which node a consumer can find a given job and on which node a producer should insert a job. We recapitulate the properties of CoCa that are important for data distribution:

1. Only one job at a time is retrieved.
2. Jobs that are local to a component are retrieved faster than jobs that are remote.
3. If it is not known where a given job resides, searches for remote jobs can be done in parallel on multiple nodes, called a parallel search, or can be done sequentially on a per node basis, called a sequential search. For the parallel search, the search time is (almost) independent of the number of nodes that is involved in the search; for the sequential search, the search time is linear in the number of involved nodes. For both, it holds that network traffic and node access contention increase linearly with the number of involved nodes. A sequential search can stop when an appropriate job is found, which reduces the number of involved nodes.
4. The consumers of a group request the same job types, use the same predicate functions, but can have different predicate function input arguments.

This has the following consequences for the CoCa data distribution:
- Remote *out* strategy for producers. Data access should be local as much as possible to decrease the communication time. We discuss two methods to obtain this: (1) store jobs of a given type only on nodes that have consumers that can consume this type of jobs, or (2) consider a consumer’s job selectivity. If the job selectivity of consumers is low, i.e. they do not specify predicates, they can consume any job of the correct type, and jobs of a given type should be evenly distributed among the consumers of a group, which can be done with a round robin strategy. If the job selectivity of consumers is high and the selectivity is known, clustering of related data on the expected consuming node might be favorable, which can be done using a range partition strategy. The definition of “related” depends on the given case.

- Remote *rd* and *in* strategy for consumers. The number of nodes involved in a search should be as small as possible to decrease network traffic and node access contention. Again, we discuss two methods to obtain this: (1) only involve nodes in a search that have jobs stored of the requested type, or (2) consider a consumer’s job selectivity. If the consumer job selectivity is low, the probability is high that a sequential search only has to access a few nodes to find an appropriate job, which reduces network traffic and node access contention with respect to a parallel search. If the consumer job selectivity is high, on a given node only jobs that satisfy a set of given predicates should be stored, which can be done using a range partition or hashing strategy. This reduces the nodes to be accessed to one, if a consumer requests a job that should satisfy one of the given predicates. If a consumer specifies a predicate different from the ones on which the storage of jobs was based, the probability is high that many nodes have to be accessed before an appropriate job is found, and a parallel search is useful to reduce the CoCa primitive execution time.

If the consumer’s job selectivity is high, producers have to know this selectivity to distribute their jobs appropriately. We give two examples where consumers consume related jobs, which allows producers to know the consumer’s job selectivity and distribute jobs accordingly:

1. Events with a given run number have to be reconstructed using the set of calibration data with the corresponding run number. A component has only one set of calibration data loaded and loading another set is time consuming. The loaded set of calibration data constitutes the component’s state and this state is determined by the previously consumed event. By sending related events, which are in this example events with the same run number, to the same node and by equipping a reconstructing component with a predicate to consume events that have the same run number as the previously consumed one, locality of data is improved.

2. Related (related as defined in section 2.5.4) subjobs might have to be recombined. The recombining component starts by retrieving an arbitrary job (which defines its state). Subsequently, by means of a predicate, only jobs can be retrieved that are related to the first one. By sending related subjobs to the same node, locality of data is obtained.

**Data distribution strategies**

The data distribution strategies round robin, range partitioning and hashing are candidates to fulfill the data distribution requirements and will be discussed here. Table 17 shows a summary of the advantages and disadvantages of the data distribution strategies for various consumer data access patterns.

1. **Range partitioning.** Range partitioning distributes jobs among nodes based on a search key, which is a function of the job content. The potential job destination nodes (i.e. all nodes with job queues of the job type under consideration) are each assigned an interval of the search key function’s range. The intervals should not intersect and together should cover the range of the search key function. We refer to the partitioning of the search key function’s range among the nodes as the range function. The search key value of a job determines the interval
Section 7.3: Job space access

...to which a job belongs and consequently a producer knows on which node to store the job. A consumer that wants to consume a specific job according to the search key knows in which interval the job falls and knows on what node it should search.

A producer using range partitioning provides a clustering of jobs with the same search key value on the same node and jobs with neighboring search key values have a large probability to be stored on the same node. If the search key is chosen such that related jobs have the same search key value, related jobs are clustered.

If the search key function results are evenly distributed among the range, equi-range partitioning produces an even job distribution among the nodes. If this is not the case, variable range partitioning can improve the distribution’s evenness, but incurs slightly more administration overhead. The next sub-section discusses search key functions more extensively.

2. **Hashing.** A hash function has a search key value as input and a node number as output. The choice of the hash function determines the data distribution. It is difficult for a non-specialist to find an appropriate hash function in terms of evenness and clustering of related data. Because for range partitioning this is much easier, we do not adopt hashing as a data distribution strategy that can be specified by the application programmer. In section 7.6, we use a hashing data distribution strategy inside CoCa to obtain recombinant data.

3. **Round robin.** Round robin distributes jobs in cyclic order among the involved nodes and therefore obtains evenness in the number of jobs distributed to each node. To avoid communication overhead, every producer node has its private round robin counter per job type, rather than one system-wide round robin counter.

In the configuration information, the configurator states the data distribution strategy for a job type. For range partitioning, the range function and search key function are also specified in the configuration information.

<table>
<thead>
<tr>
<th></th>
<th>round robin</th>
<th>range partitioning/hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>consumers without predicates</strong></td>
<td>+ (evenness)</td>
<td>- (difficult to obtain evenness)</td>
</tr>
<tr>
<td><strong>consumers with predicates</strong></td>
<td>- (multiple candidate nodes for search)</td>
<td>+ (1 candidate node for search)</td>
</tr>
<tr>
<td><strong>consumers with predicates that can benefit from clustering</strong></td>
<td>- (clustering of related data impossible)</td>
<td>+ (clustering of related data)</td>
</tr>
</tbody>
</table>

TABLE 17: Advantages and disadvantages of the data distribution strategies for different consumer data access patterns.

**How do consumers with predicates benefit from range partitioning?**

Range partitioning in databases is generally done on the tuple contents (see figure 28a), which allows one to reduce the number of candidate nodes that might contain a job with a given value. We are not interested to find a job with a given value, but to find a job satisfying a predicate. Consequently, range partitioning should be used to reduce the number of candidate nodes that might contain a job satisfying a given predicate. This means that the range partitioning of a job type has to be associated with one of the available predicate functions, to allow appropriate distribution and searching of jobs among the nodes. For each job type that uses range partitioning, the application programmer selects one predicate function out of the set of predicate functions that are used on that job type. We call this predicate function `predfunc_dd`. This selection is specified in the configuration information.

As a consumer selects a job by providing a predicate with predicate function argument, we can just as well perform the range partitioning on the predicate function input argument and not on
FIGURE 28: (a) range partitioning in databases; (b) range partitioning in CoCa.

the job value (see figure 28b). If a consumer requests a job that should satisfy the \textit{predfunc\_dd},
the input argument is used as input argument of the range function to determine the candidate
node, after which the \textit{predfunc\_dd} name and input argument are sent to this candidate node.
Not all jobs on the candidate node satisfy the predicate, as (1) jobs corresponding to multiple
predicate input argument values might reside on the same node, and as (2) the predicate might be too restrictive to allow all jobs for which the predicate input argument corresponds to the
job. To facilitate the specification of the range function in the configuration information we allow
the \textit{predfunc\_dd} to have only one input argument which has to be of type integer.

For a given job to be inserted, a producer has to determine the value of the \textit{predfunc\_dd} input
argument corresponding to the job, which is then used by the range function to determine the
job destination node. To do this, the producer needs the \textit{predfunc\_dd} inverse function, called
\textit{predfunc\_dd\_inv}, which has as output the value of the \textit{predfunc\_dd} integer input argument and
as input a job. Since the inverse function is difficult to determine automatically in all circum-
stances, we decide that it has to be provided by the application programmer as part of the applica-
tion code. The configuration information contains the name of the \textit{predfunc\_dd\_inv} to
associate it with the corresponding \textit{predfunc\_dd}. It is the application programmer’s responsi-
bility to ensure the following issues. The first three are constraints, required for correct function-
ing of CoCa. The last one is an advice and not essential for the correct functioning of CoCa, but
usually needed to obtain good performance of the parallel application:

- The set of possible jobs and integer input arguments should be part of the \textit{predfunc\_dd}
domain and \textit{predfunc\_dd} should behave as a function for the set of input values (these
should hold for any \textit{predfunc}).

- The set of possible jobs should be part of the \textit{predfunc\_dd\_inv} domain. The output of
\textit{predfunc\_dd\_inv} should be part of the range function domain. \textit{predfunc\_dd\_inv} should
behave as a function for the set of input values, otherwise a producer does not know where
to store a job.

- The output of the range function should be in the set of potential destination nodes for that
job type and should behave as a function for the set of input values (i.e. non-intersecting
intervals).

- There should be correspondence between \textit{predfunc\_dd} and \textit{predfunc\_dd\_inv} so that predi-
cates are sent to nodes that do contain jobs that satisfy these predicates. Optimally, for each
job there should be exactly one \textit{predfunc\_dd} integer input argument for which \textit{predfunc\_dd}
is true. If there are multiple input arguments, then the set of jobs that satisfies a certain pred-
dicate might be stored on multiple nodes, whereas the predicate is sent to just one node,
Section 7.3: Job space access

thereby reducing the probability to find a job. If there are zero input arguments, the job will never be selected by a consumer specifying this predicate.

We show examples of a normal predicate, a `prefunc_dd` and a `prefunc_dd_inv` function. The examples are in C and therefore appropriate for applications written in C that use the CoCa C host language binding. The other host language bindings accept an appropriately tailored syntax. Note in the example that the `prefunc_dd` can be more restrictive than just a comparison between a job data member and the integer input argument.

```c
bool prefunc(jobtype *job, argtype1 arg1, argtype2 arg2, ...) {
    return(...);
}

bool prefunc_dd(jobtype *job, int predarg) {
    return(job->member1 == predarg && job->member2 + job->member3 == 65);
}

int prefunc_dd_inv(jobtype *job) {
    return(job->member1);
}
```

Node access patterns of distributed CoCa primitives

Table 18 shows the node access patterns of the CoCa primitives as a function of the data distribution strategy specified for the job type under consideration, the predicate specified with the CoCa primitive, and whether the communication is pull or push mode. Some remarks concerning the table are:

<table>
<thead>
<tr>
<th>primitive</th>
<th>data distribution strategy and predicate</th>
<th>node access pattern (pull mode)</th>
<th>node access pattern (push mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>round robin</td>
<td>local, if failed: round robin</td>
<td>round robin</td>
</tr>
<tr>
<td>out</td>
<td>range partition</td>
<td>range function</td>
<td>range robin</td>
</tr>
<tr>
<td>rd</td>
<td>round robin + any predicate</td>
<td>parallel search</td>
<td>local, if failed: parallel search</td>
</tr>
<tr>
<td></td>
<td>range partition + predicate != prefunc_dd</td>
<td>sequential search</td>
<td>local, if failed: sequential search</td>
</tr>
<tr>
<td>in</td>
<td>round robin + any predicate</td>
<td>range function</td>
<td>range function</td>
</tr>
<tr>
<td></td>
<td>range partition + predicate != prefunc_dd</td>
<td>sequential search</td>
<td>local, if failed: sequential search</td>
</tr>
<tr>
<td>in, rd</td>
<td>range partition + predicate = prefunc_dd</td>
<td>range function</td>
<td>range function</td>
</tr>
<tr>
<td>in, rd</td>
<td>round robin + no predicate</td>
<td>sequential search</td>
<td>local, if failed: sequential search</td>
</tr>
<tr>
<td></td>
<td>range partition + no predicate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE 18: Behavior of the distributed CoCa primitives.

- The parallel search uses the job from the first node that positively replies on the request, but waits for the last result to arrive before it continues. Parallel search cannot be used for the `in`, because recovery is needed when more than one result is returned. It can be used for the `rd`, as there is no remote `release`, which implies that successful, but not used searches do not require recovery.

- CoCa is not explicitly informed whether a component uses pull or push mode. It bases it node access pattern on the following information: the job type, the distribution strategy, whether there is a job queue of the requested type locally available (the location of the job queues of the requested type), the predicate and predicate input argument.

- The round robin counter is also incremented upon a failing `out` to a node, which means that jobs are not distributed among the nodes in a strict round robin order.
• To obtain an even node access pattern, for a given job type, the starting node for a sequential search changes in a round robin fashion. There is one sequential search round robin counter per node.

• The configuration should be chosen such that parallel searches are rare, as they generate much network traffic and node access contention, and that a sequential search is almost always successful upon access of the first node to establish short communication times.

• The execution time of an access to one remote node consists of (1) the time to send the request and reply, which is constant if no job is transported, and a constant time plus the job transfer time if a job is transported, and (2) the time to execute the local CoCa primitive. The job transfer time is linear in the job size. The temporal behavior of local CoCa primitives with respect to the number of jobs present and the job size was described in section 7.2.1 about shared CoCa.

Resume

Predicates are introduced to ensure that components consume appropriate jobs. Data distribution strategies are introduced improve performance. Overhead is reduced by using one integer value as parameter to a precompiled predicate function.

7.3.2.2 CoCa communication layer

The CoCa communication layer provides a client/server protocol for inter-node communication. It takes care of synchronization and buffering of in, out and rd requests and replies. Since a node does not know in advance when another node invokes a RPC, every node should be able to receive requests at any time. This is implemented by a thread called the receive_thread that waits for requests and replies coming over the network. The advantages of a separate thread over implementing this functionality in the components are the following. It simplifies the implementation, as the scheduling of request reception is controlled via facilities offered by the OS. It allows taking advantage of the communication/computation overlap, since the receive_thread can receive requests while the components continue their computations. It ensures the absence of deadlock, as requests are always accepted, if sufficient memory is available. For hybrid nodes, multiple threads provide better processor utilization. The disadvantage of a separate thread is that it produces context switch overhead.

After the receive_thread received a request, the in, out or rd has to be executed. Two possible solutions are: (1) requests can be executed by the receive_thread itself, which means that all requests that arrive on a node are sequentially processed by one thread, or (2) each request is processed by a separate thread. The second option can be implemented by having a pool of threads, or by dynamically spawning threads when they are needed, which incurs creation overhead. The second option results in a smaller variance in the request processing time than the first option, and for hybrid nodes also the average request processing time might be smaller than for the first option. On the other hand, the multiple threads of the second option produce context switch overhead. The trade-off between option 1 and 2 depends on the number of processors per node. We provide both implementations to be able to determine their performance via tests.

Client/server protocol in detail

Since the communication layer uses functionality provided by the CoCa physical layer, this functionality is described first (see also section 7.3.2.3):

• There is a (logical) bidirectional channel between every pair of nodes. Only one communication at a time is allowed on a channel in each direction.
Section 7.3: Job space access

- Synchronous *send* and *receive* operations are used to communicate messages via a channel. Both the sender and receiver have to specify the size of the communicated message. There is an operation to test whether a message has arrived on a channel.

- A *signal* operation can be invoked by a thread to inform another node that an event has occurred. There is a *wait for signal* operation to block a thread until a signal occurs.

Jobs exist in different sizes. Since the receive operation has to know the length of the message to be received, the job is sent in two parts (see figure 29). First a message of a fixed format and size is sent, containing the request or reply type (*out_request*, *in_request*, *rd_request*, *out_reply*, *in_reply* or *rd_reply*), the request or reply parameters, and if necessary the job size, but not the job itself. This allows the *receive_thread* to prepare a slot of the required size on reception of the message, in the cases where this is necessary. We discuss the various request and reply types. An *out_request* contains the job size, and upon its reception, the *receive_thread* prepares a buffer of the required size and receives the job. It then spawns an *out_request_thread*. The *out_request_thread* invokes a local CoCa *out* to insert the job into job space, and returns an *out_reply* to the client containing an acknowledgment indicating success or failure. Upon reception of an *in_request (rd_request)*, the *receive_thread* spawns an *in_request_thread (rd_request_thread)*. The *in_request_thread (rd_request_thread)* invokes a local CoCa *in* using the job type and predicate specified in the message, to retrieve the requested job and returns an *in_reply (rd_reply)* to the client containing an acknowledgment indicating success or failure and in case of success, containing the job size. This allows the *receive_thread* at the client side to prepare an appropriate slot for the job to be received. In case of success, the *in_request_thread (rd_request_thread)* subsequently sends the requested job.

On a node, for every channel a buffer is ready to receive a message, so that the *receive_thread* can receive requests and replies from any node. One message buffer per channel is sufficient, since only one communication at a time is possible on a channel in one direction. Consequently, there can be only one RPC at a time per channel in one direction, meaning that per channel and in one direction, RPCs are performed sequentially. RPCs on different channels can be performed concurrently.

To avoid the overhead of a separate *client_thread* and to allow more concurrency, the components *themselves* perform remote requests, resulting in concurrent request execution. The communication layer design must support this. We describe the relevant features.

After having done a request, a component waits for the reply by suspending itself on a condition variable to save resources. Since we decided to implement synchronous RPCs, any component can have only one request at a time, which means that one condition variable per component is sufficient. Upon reception of a reply, the *receive_thread* (at the client side) wakes up the suspended component and informs the component where to find the reply. This is done via the *reply_buffer*, which contains one element per component. Before the component sends the request to the remote node, it reserves an element in the *reply_buffer*. When the component is woken up, this element contains a reference to the reply. The reply contains a reference to the job in case of successful *in or rd*.

The notification of a message arrival by the *receive_thread* is discussed. It has not been discussed how the *receive_thread* notices that a request or reply has arrived. It can cyclically poll the channels, but this costs much processing power. Therefore, signals are used. The *receive_thread* suspends itself and is woken up by a signal that is sent by the remote node shortly before the remote node sends the request or reply itself. The *receive_thread* then determines on which channels requests and replies have received.
Summarizing, each node contains one server stub that consists of the receive_thread and threads to execute requests. For each component, there is a client stub. The client stubs can run concurrently, but together use one receive_thread for sending and receiving messages and jobs.

7.3.2.3 CoCa physical layer

The CoCa physical layer hides the particular features of the distributed memory platform under consideration. It adapts the functionality offered by the native communication library to what has been decided to offer to the communication layer. As every distributed memory platform comes with a native message passing library, we decided to model the CoCa physical layer according to the message passing paradigm, with basic send and receive primitives that send messages between nodes rather than between processes. This functionality is low enough to guarantee a good response time.

7.4 Job space consistency

7.4.1 Shared CoCa

The job space is consistent if every job is in one of the states of the job state diagram introduced in chapter 5 and if the metadata reflects the job state of all jobs. A CoCa primitive should bring the job space from one consistent state into another consistent state (correctness constraint). During the execution of a primitive, inconsistency is allowed, although not visible to other executing components (concurrency atomicity constraint). If a CoCa primitive logically fails, it should leave the job space in the state prior to its execution (fail atomicity constraint). We assume the absence of hardware failures and do not further discuss the durability constraint. The remainder of this section is structured according to the ACID constraints. The functionality described in this section is part of the local CoCa layer.

7.4.1.1 Concurrency atomicity

The goal is to ensure concurrency atomicity and to find a trade-off between the potential concurrency and the generated overhead. We separately discuss the four types of data that were distinguished earlier.
Section 7.4: Job space consistency

Control data

In chapter 6, we decided to use Solaris synchronization primitives as control data. We concluded that (1) a mutex preceded by a while loop for empty list waiting was the preferred solution, and that (2) the number of mutex operations that is used should be kept low, as they are time consuming.

Meta data

CoCa primitives concurrently access meta data and the control data are used to ensure concurrency atomicity of the meta data. As all successful CoCa primitives change the state of the job space, they change the meta data. This means that not much concurrency between actions of different CoCa primitives that act on the meta data of one job queue is allowed, and we decide to ensure serializability by enforcing sequential access of the CoCa primitives that act on the same job queue. We decide to use mutexes rather than read/write locks.

The choice of the number of mutex primitives to use is a trade-off. One mutex per independent resource allows concurrent access to the resources; fewer mutexes result in less mutex operations, so less overhead, but also less concurrency. With respect to this trade-off, it is important to remark that since CoCa primitives use reference access to select jobs, they are fast with respect to mutex operation execution times. There are three independent resources that are handled by the meta data and that we discuss separately. In the discussion, we refer to the list exclusion and empty list waiting as introduced in chapter 6.

- Job queues: a CoCa primitive only accesses the meta data of one job queue, and hence job queues of different job types are independent, which suggests a concurrency granularity at the job queue level. Jobs in a job queue are randomly selected, which makes one mutex per job queue suitable to obtain list exclusion for the meta data. We decide to use one mutex, called excl, meaning that CoCa primitives access one job queue sequentially, but can access different job queues concurrently. Empty list waiting is obtained by preceding the mutex by a while loop testing whether the job queue contains jobs indicated by the variable njobs.

- Slot queues: getslot and returnslot only access the meta data of one slot queue, and hence slot queues of different job types are independent, which suggests a concurrency granularity at the slot queue level. We decide to use one mutex, called slot.excl, per slot queue. Consequently, a slot queue is accessed sequentially, a job queue and its corresponding slot queue can be accessed concurrently, and different slot queues can be accessed concurrently. Empty list waiting is obtained by preceding the mutex by a while loop that tests (1) whether the slot queue contains slots indicated by the variable nslots and (2) whether the total number of allocated buckets, called Nbuckets, is less than Nbucketmax.

- Not yet allocated component shared memory: the not yet allocated component shared memory is a system-wide resource, which requires global concurrency granularity. We use a global mutex shm.excl to obtain list exclusion. A while loop testing whether not yet allocated component shared memory is left, using the boolean shm.left, is used to obtain empty list waiting. Once no component shared memory is left, this situation does not change, as we use the semi-dynamic allocation strategy.

After discussing job and slot queue access above, we now consider the consequences for the CoCa primitives. A CoCa primitive has the following general design: statements that access meta data related to one of the resources described above have to be executed in a critical section, which is done by enveloping the statements by the corresponding mutex. The question is in how many atomic parts can or should each CoCa primitive be divided. One critical section allows no concurrency of different CoCa primitives, but requires only two mutex operations per CoCa primitive. Multiple smaller critical sections allow concurrency, but require more
mutex operations. Considering the execution time of a mutex operation relative to that of a CoCa primitive, we decide to use one critical section per resource per CoCa primitive. A CoCa primitive that concurrently accesses multiple resources has nested critical sections. In general, nested critical sections should be avoided, as they decrease the potential concurrency and, as discussed later, may result in deadlock.

**Job data**

Job data comprises jobs and slots, and in this discussion these are not distinguished. Job selection affects the meta data and job access does not. A CoCa primitive selects only one job copy at a time and jobs are independent, which suggests a concurrency granularity at the job level. An argument for this choice is that job selection uses reference access and is therefore fast with respect to job access (job access includes reading, copying or writing a job), depending on the application and job size.

With respect to concurrent access of multiple components to one job the following remarks can be made. The job space has been designed such that only one component writes to a job copy, or multiple components read a job copy. Consequently, concurrency atomicity is ensured and no control data have to be used to protect job access, which means that the job state, determining the job's accessibility, can be described by ordinary variables (these variables are part of the meta data).

**Catalog data**

As catalog data are read-only, no concurrency control is required.

### 7.4.1.2 Correctness and fail atomicity

In chapter 5 five job states were introduced as viewed by the application programmer. As different CoCa primitives reach jobs via different ways, not all CoCa primitives require the same job state information for their correct execution. A job is represented by job copies. As a primitive selects only one job copy, we decide to add state information on a per job copy basis. We equip every job copy with a variable `Nreaders`, indicating the number of readers of that job copy, and a boolean variable `in_job_space`, indicating whether a job copy is referenced from job space. Below is shown that these two variables are necessary and sufficient. The sum of the value of `Nreaders` of all job copies of the same job gives the number of readers of a job as seen by the application programmer. We now discuss how the CoCa primitives maintain consistency.

The concurrency control algorithms for the primitives are shown in C code and also show the mutexes that ensure concurrency atomicity. To simplify the code, we assume that primitives are given that implement the CoCa functionality in case no concurrency had been allowed, which comprises finding the correct job and managing part of the meta data. `rd_` selects a job in job space. `out_` provides job space with a reference to the job and removes the component's reference to the job; it always succeeds. `rm_` removes the job space's reference to a job; it always succeeds. `returnslot_` returns an obsolete slot to the correct slot queue; it always succeeds. `extend_slot_queue` acquires a segment in component shared memory of the length `Nbucket_text*bucket_size` and inserts `Nbucket_text` slots in the slot queue. All these primitives return a status indicating success or failure. Furthermore, `getslot_` acquires a slot from the specified slot queue and returns its reference to the caller; if it fails to acquire a slot, it returns `NULL`. 
Section 7.4: Job space consistency

rd

As a rd finds a job by searching in job space, it can only select jobs that are in state in job space or in job space and read. The actions that the rd primitive performs are the same for both states: it has to increase Nreaders and supply the reading component with a reference to the job.

```c
bool rd(jobtype** job, predtype* predicate) {
    bool status;
    while(jobqueue.njobs==0) ;
    mutex_lock(jobqueue.excl);
    status = rd_(job, predicate);
    if(status)
        (*job)-->Nreaders++;
    mutex_unlock(jobqueue.excl);
    return status;
}
```

release

A release primitive can only be invoked by a reading component and only affects the job copy that is read by that component. Consequently, it can work on jobs in states read, assigned and read and in job space and read. We go over the various cases:

- **read**: there are one or more copies of the job that are all being read, meaning that every job copy has a variable Nreaders > 0. The release decrements Nreaders, and if Nreaders is zero, it returns the job copy to the slot queue.

- **assigned and read**: there are two or more copies of a job, of which one is assigned to a component and the others are being read. The release decrements Nreaders, and if Nreaders is zero, it returns the job copy to the slot queue.

- **in job space and read**: the history of the job is important:
  1. There is one copy of the job if the most recent different job state was in job space and not assigned and read. The copy is being read and has a reference from job space pointing to it. The release decrements Nreaders, and if Nreaders is zero, it does not return the job copy to the slot queue.
  2. There are two or more copies of a job, if the most recent different job state was assigned and read and not in job space. One of the job copies has a reference from job space pointing to it and might be being read, all the other job copies are only being read. The release decrements Nreaders, and if Nreaders is zero and the release works on a job copy that is not referenced from job space, it returns the job copy to the slot queue. If Nreaders is zero and the release works on a job copy that is referenced from job space, it does not return the job copy to the slot queue.

Consequently, the release has to be able to determine whether a job copy is referenced from job space. For this reason, we introduced the boolean variable in job space. Only the in and out primitives change the value of this variable.

returnslot uses the slot exclusion mutex. To avoid invoking returnslot when the release still has the job exclusion mutex, which would mean a nesting of critical sections, the boolean variable obsolete is introduced.

```c
bool release(jobtype* job) {
    bool status;
    bool obsolete = FALSE; /* to avoid nested critical sections */
    mutex_lock(jobqueue.excl);
    status = job->Nreaders--;
    if(job->Nreaders==0 && !job->in_job_space)
```
obsolete = TRUE;
mutex_unlock(jobqueue.excl);
if(obsolete)
    returnslot(job);            /* returnslot always successful */
return status;
}

in
As an in finds a job by searching in job space, it can only select jobs that are in state in job space or in job space and read. We go over the various cases:

• in job space; the component invoking the in is provided with a reference to the job, the job space’s reference is removed and in_job_space becomes false.

• in job space and read; if the in finds a job copy with Nreaders = 0, it performs the same actions as for state in job space. If the in finds a job copy with Nreaders > 0, it makes a second job copy, removes the job space’s reference to the original job copy, provides the component invoking the in with a reference to the second job copy, and sets in_job_space to false, for both the second and the original job copy.

Copying is a relatively slow operation and cannot be allowed to monopolize the job queue. Therefore it should happen outside the critical section of the job queue mutex. To accomplish this, the in performs a rd, makes a job copy using memcpy and invokes a release.

bool in(jobtype** job, predtype* predicate) {
    bool status;
    while(jobqueue.njobs==0) ;
    mutex_lock(jobqueue.excl);
    status = rd_(job, predicate);
    if(status) {
        rm_(*job);            /* status not checked as always successful */
        (*job)->in_job_space = FALSE;
        jobqueue.njobs--;
        if((*job)->Nreaders > 0)
            (*job)->Nreaders++;
    }
    mutex_unlock(jobqueue.excl);
    if(status && (*job)->Nreaders > 0) {
        jobtype* job_copy;
        job_copy = getslot();
        if(job_copy) {
            job_copy->in_job_space = FALSE;
            job_copy->Nreaders = 0;
            memcpy(job_copy,*job);            /* make a job copy */
            release(*job);
            *job = job_copy;            /* supply component with reference to job_copy */
        }
    } else {            /* recover pre-primitive situation (fail atomicity) */
        mutex_lock(jobqueue.excl);
        (*job)->Nreaders--;
        out_(*job);           /* job queue ordering has been changed */
        (*job)->in_job_space = TRUE;
        jobqueue.njobs++;
        mutex_unlock(jobqueue.excl);
        status = FALSE;
    }
}
return status;
When getslot fails, recovery of the $rm_-$ and job state variables is required to obey the fail atomicity constraint. The $out_-$, used for the recovery, inserts the job not in the previous location in the job queue, but appends the job at the end, meaning that an unsuccessful $in_-$ has a net effect. As in practice this effect is seldom of importance and as adapting the implementation to avoid the effect increases the complexity of the algorithm, we allow this situation to occur.

$out_-$

As an $out_-$ primitive can only be invoked by a component to which the job is assigned, it works on jobs in states $assigned$ and $assigned and read$. The actions that the $out_-$ primitive performs are the same for both states: it provides the job space with a reference to the job and $in_job_space$ becomes $true$.

```c
bool out(jobtype* job) {
    bool status;
    mutex_lock(jobqueue.excl);
    status = out_(job);
    if(status) [
        job->in_job_space = TRUE;
        jobqueue.njobs++;
    ]
    mutex_unlock(jobqueue.excl);
    return status;
}
```

$create$

$create$ works on job copies that have no references pointing to them.

```c
bool create(jobtype** job) {
    *job = getslot();
    if(*job) {
        (*job)->in_job_space = FALSE;
        (*job)->Nreaders = 0;
        return TRUE;
    } else
        return FALSE;
}
```

$destroy$

$destroy$ can work on jobs in state $assigned$ or $assigned and read$ and always works on job copies that only have a reference pointing to them from the component invoking the $destroy$.

```c
bool destroy(jobtype* job) {
    return returnslot(job);
}
```

$getslot$ and $returnslot$

$getslot$ and a $returnslot$ access a slot queue sequentially, as we decided to use one mutex per slot queue. $getslot$ can proceed if (1) there are slots left in the slot queue, if (2) there is non-allocated component shared memory left, or if (3) the total number of allocated buckets is less than $Nbuckethmax$. Otherwise $getslot$ waits. If (1) is false, while (2) and (3) are true, $extend_slot_queue$ is called to allocate $Nbuckethtext$ new buckets in not yet allocated component shared memory. The global mutex $shm.excl$ is internal to $extend_slot_queue$ and therefore not
visible in the code. To avoid overhead, Nbuckettext should not be chosen too small, so that extend_slot_queue is not often called and the global mutex is not often acquired. As extend_slot_queue also accesses the slot queue, it is executed under protection of the slot_excl mutex. This results in nested critical sections. Nested critical sections reduce the potential concurrency, but as extend_slot_queue is not often called, this can be accepted.

jobtype* getslot() {  
    jobtype* slot = NULL;
    while(jobqueue.nslots==0 && !shm.left &&
        jobqueue.Nbuckets==jobqueue.Nbucketmax) ;
    mutex_lock(jobqueue.slot_excl);
    slot = getslot_();
    if(!slot && shm.left && jobqueue.Nbuckets<jobqueue.Nbucketmax) {  
        extend_slot_queue();
        slot = getslot_();
    }
    if(slot)
        jobqueue.nslots--;
    mutex_unlock(jobqueue.slot_excl);
    return slot;
}

bool returnslot(jobtype* slot) {
    mutex_lock(jobqueue.slot_excl);
    returnslot_(slot);
    jobqueue.nslots++;
    mutex_unlock(jobqueue.slot_excl);
    return SUCCESS;
}

The reason why the jobqueue.nslots++ statement is put before the mutex_unlock has been given in the description of algorithm 5, section 6.2.2.5.

Deadlock

A cycle in the wait-for graph of components indicates deadlock. A cycle in the wait-for graph is only possible if a component is blocked waiting for another resource while it possesses a resource. In the CoCa primitives, two types of blocking are possible: (1) suspension on a mutex acquisition operation while the mutex is possessed by another component, and (2) busy waiting in a while loop of which the condition is true. In CoCa the following resources can be distinguished for which a component can wait: (1) access to a job queue, (2) access to a slot queue, and (3) access to not yet allocated shared memory.

Deadlock in the CoCa primitives is only possible for the following two situations: (1) a component acquired a mutex and then enters a while loop for busy waiting, or (2) a component acquired a mutex and attempts to acquire a second mutex. Situation 1 does not occur in the code of the CoCa primitives. Situation 2, i.e. nested critical sections, only occurs in the getslot function. getslot’s for different job types share only one resource, namely not allocated component shared memory, and no deadlock is possible. getslot’s for the same job type access the resources in the same order, which prevents deadlock.

Resume

The CoCa primitives are designed such that although an efficient solution for empty list waiting is implemented, a CoCa primitive can return with a failure. If the empty list waiting was implemented by the application programmer, this would imply that the mutexes had to be accessed in the busy waiting loop, which results in a high lock access contention. The empty list waiting
introduced in the algorithm avoids this. Consequently, an application programmer still has to take the possibility of a failing CoCa primitive into account.

### 7.4.2 Distributed CoCa

Design goals are (1) to avoid a concurrency control mechanism in which components have mutexes on remote job copies, and (2) to use the concurrency control mechanism of local CoCa without essential modifications. The distributed way in which the information about the number of readers of a job is stored helps to accomplish this. We discuss the primitives separately. We only show the code of the request threads which execute at the server side. As the meta data at the client side is not accessed and only the invoking component has access to the job, no concurrency control is required at the client side.

**in**

To avoid monopolizing of the job queue, the time consuming transport of the job to the client has to be done outside the job queue mutex. No recovery is required as *rpc_reply*, in contrast to *getslot*, cannot fail.

```c
in_request_thread(predtype* predicate) {
    bool status;
    jobtype* job;
    while(jobqueue.njobs==0) ;
    mutex_lock(jobqueue.excl);
    status = rd_(&job, predicate);
    if(status) {
        rm_(job); /* status not checked as always successful */
        job->in_job_space = FALSE;
        jobqueue.njobs--;
        if(job->Nreaders > 0)
            job->Nreaders++;
    }
    mutex_unlock(jobqueue.excl);
    if(status) {
        rpc_reply(job);
        if(job->Nreaders == 0) /* job was not read by other components */
            returnslot(job);
        else
            release(job);
        else
            rpc_reply(failure);
    }
}
```

**rd**

A remote *rd* goes via the optimization described in chapter 5: the *rd_request_thread* selects the job for reading, transports the job to the client and invokes a *release*. The job copy received by the client has a *Nreaders* of 1. As mentioned in chapter 5, the job copy received by the client can reside in component private memory, since it cannot be reused after the component finished reading. However, this would require using the OS operations malloc and free, that are more general than *getslot* and *returnslot* and therefore incur more overhead. Furthermore, in chapter 6, it was mentioned that the difference between the time to access component shared memory and the time to access component private memory is negligible. Consequently, we decide that job copies reside in component shared memory.
rd_request_thread(predtype* predicate) {
    bool status;
    jobtype* job;
    while(jobqueue.njobs==0) ;
    mutex_lock(jobqueue.excl);
    status = rd_(&job, predicate);
    if(status)
        job->Nreaders++;
    mutex_unlock(jobqueue.excl);
    if(status) {
        rpc_reply(job);
        release(job);
    }
    else
        rpc_reply(failure);
}

release
The optimization that after a rd with move access the release is invoked immediately (see section 5.4.1), has as a result that a remote release primitive is not required.

out
The code of the out_request_thread is straightforward. We assume that the receive_thread already performed a getslot call and subsequently received the job to be inserted.

out_request_thread(jobtype* job) {
    bool status;
    mutex_lock(jobqueue.excl);
    status = out_(&job);
    if(status) {
        job->in_job_space = TRUE;
        jobqueue.njobs++;
    }
    mutex_unlock(jobqueue.excl);
    rpc_reply(success);
}

Other primitives
A create creates a job local to the component invoking the create. A destroy destroys jobs that are in state assigned or assigned and read, which means that the job copy that is destroyed, is local to the component invoking the destroy. getslot and returnslot only have to respectively select or return slots on the node on which they are invoked. Consequently, no remote versions of these primitives and functions are required.

7.5 The components

7.5.1 Shared CoCa

Components can be UNIX processes or POSIX threads. Threads have less context switch overhead, but multiple threads in one process cannot host components having global variables with the same name. For newly built applications, this problem can be solved by renaming variables in components, but this is impossible for existing applications where part of the source code is not available. An example is the CPLEAR test case described in chapter 8, that relies on the CERNLIB of which the source code is not available. Processes do not have this name clashing
Section 7.6: Recombination of related data

problem and are therefore chosen. This means that the job space has to reside in process shared memory, to allow it to be implicitly accessed by all components, and a component’s private variables can reside in process local memory.

Nodes on a shared memory computer are indistinguishable. Therefore, only the location of components relative to each other is of importance: whether components reside on the same or on different nodes and how many components and of which type reside on a node. We decide to use the (dynamic) load balancing mechanism that shared memory computers usually offer to do the component allocation. This means that the configuration information does not include the component/node mapping.

7.5.2 Distributed CoCa

Also for distributed CoCa the question arises whether components should be mapped to threads or processes. Since a node of a distributed memory computer usually contains less processors than a shared memory computer, the name clash problem is less likely to occur. Therefore, and also to gain experience with threads and to be able to compare the performance of a thread implementation with that of a process implementation, we decide to map components to threads. The main consequence is that a node cannot run two components of the same type. This does not mean that distributed CoCa cannot take advantage of the communication/computation overlap and of hybrid nodes, as the distributed CoCa design uses a receive_thread and request threads besides the component threads.

With respect to the mapping of components to the processors of a node, the same argument holds as given for shared CoCa and therefore the same solution is chosen: the (dynamic) load balancing system provided by the operating system is used to distribute components among the processors of a node. Consequently, the configuration information includes the component/node mapping, but not the component/processor mapping.

7.6 Recombination of related data

To perform correct recombination of related data, information retrieved from the job content and information about the job history might be needed. The basic CoCa functionality, i.e. primitives with predicates, is sufficient to allow the application programmer to ensure correct recombination. As predicates act on the job content, the job definition might have to contain fields that describe the job history, meaning that the job history is visible to the application programmer. This is a very flexible solution, as the application programmer can adapt the communication pattern to the specific needs of the application under consideration. However, it implies that the parallel nature of the application is interwoven with the application code. Furthermore, the code contains hardware platform dependent information which compromises portability, as our aim was to restrict hardware dependent information to the configuration information. A third argument is that in push mode, predicates are not of help to efficiently direct related data to the correct recombining component. We do not yet have a data distribution strategy that can support this, which means that the performance is compromised.

For these reasons, we propose a facility that provides automatic recombination of related data which is transparent to the application programmer. This facility ensures correct recombination for most parallel applications, but as we cannot foresee all communication patterns that can occur in parallel applications and as an attempt to do this might result in a large overhead, the automatic recombination feature of CoCa can be bypassed, in which case it is the application programmer’s responsibility to ensure correct recombination in the way described above. In the
remainder of this section, we focus on the automatic recombination facility. For this discussion, we do not distinguish between the in and rd.

7.6.1 Shared CoCa

Related jobs need to be recombined in case of intra-job parallelism (geometric data parallelism and task parallelism), but not for inter-job parallelism (farming or pipelining). We consider some characteristic communication patterns. Intra-job parallelism is usually characterized by a pattern that starts with a fanout and possibly finishes with a recombination, whereas in inter-job parallelism, jobs are merely transferred between components. We distinguish the following data access patterns 1, 2, 3a and 3b and restrict ourselves to components that have pattern 1 or 3a at their input side and pattern 2 or 3b at their output side:

1. Fanout: the output of a component consists of multiple related subjobs.
2. Recombination: the input of a component consists of multiple related subjobs.
3. Transfer which can be: (a) the input of a component consists of one or multiple unrelated subjobs, or (b) the output of a component consists of one or multiple unrelated subjobs.

---

**FIGURE 30: Job Identification.**

We start to discuss acyclic data flow graphs. Later, we address data flow graphs with cycles. Figure 30 shows an example of an acyclic data flow graph. A node of the directed graph is a component, an edge indicates the communication of a job. The edge label is the jobid. Every job is equipped with a jobid, such that every related job set is uniquely labeled, which is necessary to ensure correct recombination. The jobid gives information about a job's history. It is not visible to the application programmer. We explain the construction of the jobid. It consists of two building blocks:

1. A logical timestamp that, on a per component basis, labels subjobs such that it is known of which job (proper jobs, not subjobs!) a subjob is originating. The jobid can contain only one timestamp. The timestamp is assigned to a job's jobid by the component that retrieved the job. The timestamp is denoted in figure 30 by ts. The timestamp allows one to distinguish subjobs participating in a fanout performed by a given component from subjobs participating in a subsequent fanout performed by the same component. Jobs that have never been consumed have a ts of 0.

2. A postfix that labels subjobs such that it is known which component of a group performed the fanout of which the subjob was part. To be able to add the postfix to a subjob's jobid, each component has a fixed componentid that uniquely defines a component within a group of components. In figure 30, five groups A, B, C, D and E are distinguished. Group A and E have one member; group B, C and D have two members. Upon a fanout, the componentid of
the producing component is postfixed to the jobid. Upon a recombination, the latest added postfix is removed. Upon a transfer, the jobid is not changed. Fanouts can be nested, meaning that the jobid can contain multiple postfixes. The fanout/recombination pairs have to be matched. This means that a jobid behaves like a stack: the last postfix is removed or a postfix is added behind the last one. The number of postfixes indicates the level of recursion of the "related" concept and we call it a job's depth. Jobs that did not participate in a fanout, or jobs that already have been recombined, are not part of a related job set and do not have to be recombined, meaning that they participate in inter-job parallelism. These jobs have a depth of zero which is indicated with a postfix of 0. The postfixes are denoted in figure 30 by the numbers behind the dash, each number is one postfix.

The timestamp and postfixes are sufficient to uniquely label related job sets. As an example (see figure 30), the jobid 1-12 has the following history. A job has been consumed by a component with componentid 1, indicated by the 1 behind the dash. Its output was a fanout and it was the first time that it performed such a fanout, indicated by the 1 in front of the dash. The job got consumed and subsequently fanned out by a component with componentid 2. Jobs with the same jobid are related (on the lowest nested level) and jobs with different jobids are not related. So all jobs with jobid 1-12 are related. Two jobs with jobid 1-12 and 1-13 are not related on the lowest nesting level, but are related one nesting level higher, which is indicated by jobid 1-1. For example, all jobs participating between a fanout and the corresponding recombination comprise a related job set.

To correctly manage the jobids, CoCa has to be able to determine from the application code which communication statements constitute a fanout, a recombination or a transfer. We distinguish two cases:

• **Task parallelism.** Two new primitives `start_transaction` and `finish_transaction` are introduced, with which the application can indicate which set of communication statements comprise a related job set, i.e. it is a fanout or a recombination. The section between the `start_transaction` and `finish_transaction` primitives is called a transaction. The communication statements in a transaction deal with a related job set. Consequently, a component performing a fanout knows how to encode the jobid of the subjobs participating in the fanout, and a component performing a recombination knows the jobids of the subjobs it has to retrieve.

• **Geometric data parallelism.** This is similar to task parallelism with two complications:
  i. The number of related subjobs in a fanout can be variable, which is an example of conditional communication. This is problematic, as it implies that a recombining component does not know how many related subjobs to expect. As a solution, the jobid of the last CoCa primitive of the fanout contains information about the total number of subjobs in the transaction.
  ii. Subjobs have the same type and are therefore not uniquely distinguishable by CoCa. This is problematic, as with the overtake problem, described in section 2.5.4, ordered reception cannot be guaranteed. The solution is to encode information in the jobid to make a subjob unique within the transaction. In the cases where it is difficult to pass this information to the application, for example for applications written in Fortran, we decide that ordered reception is the responsibility of the application programmer. This means that the job definition contains a field allowing unique distinction of subjobs in a transaction.

**Data flow graphs with cycles**

Data flow graphs with cycles have the additional constraint with respect to acyclic graphs that related subjobs not only have to be recombined, but they also have to be consumed by the com-
ponent that performed the matching fanout. An example of a cyclic data flow graph is the mas-
ter/slave pattern. We meet the additional constraint by having the fanout and recombination in
one transaction. Geometric data parallelism fanouts with a variable number of participating
subjobs are not problematic, as this information is automatically available in the recombining
component, which means that the corresponding information in the jobid is redundant.

Conditional communication
For the present model, conditional communication has to be separated into no output on one
hand, and one or multiple output jobs on the other hand. The latter is supported by the presented
model, but the former is not, as a recombining component waits for all jobs to be received. To
circumvent this problem, a rejected subjob is sent to its destination anyway, but tagged as
invalid. The receiving component has to know how to handle invalid jobs, which is the respon-
sibility of the application programmer. This solution (1) does not save communication, (2) does
save CPU resources, because the job does not give rise to computation in the subsequent part of
the data flow graph, and (3) does decrease the latency of rejected jobs. A solution that saves
CPU resources used in parallel branches and accomplishes more latency decrease is an early
stop option, which can be implemented using broadcasting. This can be implemented in the
future, but may not be an improvement due to the high communication overhead of a broadcast.

The recombination algorithm
A component is equipped with a variable called currentid that indicates its state. currentid can
have the value “0”, indicating that it is not processing a related job set, or a certain jobid, indic-
ating that it is processing a subjob of a related job set with that jobid. The algorithm is as fol-
loows:

• If an out is the first primitive of a transaction, the section is identified as a fanout. The cur-
rentid is postfixed with the componentid and the out under consideration and subsequent
out’s get the currentid as jobid. The finish_transaction primitive removes the last postfix
of the currentid. If there are also in primitives in the transaction (they are usually after the
out’s), the in primitives request jobs that have the currentid as jobid. A situation with first
out primitives and then in primitives is a “fanout and matching recombination by the same
component”.

• If an in is the first primitive of a transaction, the section is identified as a recombination. The
first job to be retrieved can have any jobid. This jobid becomes the currentid. Subsequent in
primitives have to retrieve jobs with the currentid as jobid. The finish_transaction primitive
removes the last postfix of the currentid. It is not allowed to have out primitives in a transac-
tion identified as a recombination, since this situation has no useful meaning.

• An in outside a transaction is identified as an inter-job parallelism operation. It can retrieve a
job with any jobid. This jobid becomes the currentid.

• An out outside a transaction is identified as an inter-job parallelism operation. The job in
question gets the currentid as jobid. The currentid does not change until an in is performed
or a transaction is started.

Multi-level components
A multi-level fanout (recombination) is a fanout (recombination) in which one component han-
dles related subjobs with different depths at its output (input) side (see figure 31). To increase
the complexity of the semantics of the primitives and of the implementation not too much, we
do not allow multi-level components. This is not a fundamental limitation, as a multi-level
component can be decomposed into one-level components, although this requires more compo-
nents to implement a given communication pattern. Note that nested transactions (see section 3.3) are still allowed.

![Diagram](image)

**FIGURE 31:** Multi-level components.

### 7.6.2 Distributed CoCa

If related subjobs to be recombinated are scattered among multiple nodes, the recombination is correctly performed, but time consuming. Therefore, we optimize the distribution of jobs among the consumer nodes. We separately discuss pull and push mode. Pull mode is similar to shared CoCa, meaning that a recombinating consumer decides which jobs to consume, allowing it to retrieve the correct related subjobs. For push mode, a *producer* has to know whether the subsequent consumer will perform a recombination. In this case, the producer has to store related subjobs on the same node. This is accomplished by introducing two more data distribution strategies that can be specified in the configuration information: *recombination* for a recombination to any component, and *return* for recombination to the original fanout component. We distinguish the following cases:

- **Pull mode recombination.** A component starts by retrieving a job with any jobid, followed by parallel or sequential search for jobs with the same jobid.
- **Push mode recombination.** The distribution strategy for the involved job queue is *return* or *recombination*. In the case of *return*, the destination componentid is equal to the last postfix, which is the componentid of the fanout component. This accomplishes that jobs are returned to their original sender. In the case of *recombination*, the destination componentid is equal to the logical timestamp modulo the number of consuming components. This accomplishes a distribution as if the original fanout components had performed a round robin with each their own round robin counter. The component/node mapping is used to determine the destination node on which a component with the destination componentid resides, and subsequently, the job is sent to this node.

Figure 32a shows an example of return recombination in which components A1 and A2 each produce 3 jobs that contain 2 subjobs each. The components of type C process the subjobs. The results are of a type for which the return strategy is chosen. Therefore, the results are transported to the nodes containing components with a componentid equal to the last postfix of the results’ jobid. Figure 32b shows an example of recombination. The components A1, A2 and A3 each produce 6 jobs that contain 2 subjobs each. Components of type B process the subjobs and produce for each subjob a result with the same jobid, as the components of type B are transfer components. For the result type, the recombination strategy has been chosen. Therefore, a result is transported to the node containing a component of type C with a componentid equal to the logical timestamp of the result’s jobid modulo the total number of components of type C.
• Fanout in pull or push mode. The distribution strategy for the involved job queue is not return or recombination, but round robin or range partitioning and the destination node is selected according to the chosen strategy, which means that the jobid is ignored.

(a) push mode return recombination

(b) push mode recombination

FIGURE 32: Recombination in distributed CoCa. Figure (a) shows push mode return recombination. Figure (b) shows push mode recombination. A square box is a node, a circle a component and a rectangular box a job queue.

Resume of recombination for shared and distributed CoCa

The jobid labels jobs such that the hierarchic structure of the related job sets is encoded and that related job sets can be uniquely distinguished. This hierarchy corresponds to the nested transaction hierarchy (see section 3.3). Consequently, the jobid allows the recombination algorithm to ensure concurrency atomicity of a transaction. The algorithm ensures transaction fail atomicity in the following way: if a transaction fails, its result is an invalid job. Another transaction or parent-transaction that uses this result recognizes the invalid job and handles accordingly.

For shared CoCa and for distributed CoCa with multiple components per node, the recombination feature results in searches in a job queue. This could have been avoided by partitioning a job queue in separate queues with related subjobs. We did not do this for two reasons: (1) this partitioning incurs permanent overhead that also decreases the performance if recombination is
not required, and (2) our goal was to provide all functionality, just by having a job space and a predicate mechanism. Remark that for distributed CoCa, the return or recombination strategies do perform this partitioning, which consequently avoids searching through a job queue.

7.7 Host language bindings

The aim of this section is to integrate the CoCa primitives with the host programming languages, C++, C and Fortran. We have several requirements. The host language binding should be in agreement with the general philosophy of the programming language. The overhead incurred by the binding should be small. The binding should support the separation of application code and configuration information. The binding should not inhibit to model data such that they are readily usable by the environment.

The core of CoCa, i.e. everything of CoCa below the language binding layer, has to be able to work with three programming languages. This has a similar restrictive effect as the hardware portability requirement: for the design of the CoCa core, only functionality can be used that is common to the three programming languages. We decide to implement CoCa in C, as C has a reasonable compatibility with both Fortran and C++. This means that we cannot use the full expressive power of C++ to build the C++ binding. For example, the CoCa core does not use typed job queues, but represents the job type by means of an index, which resulted in explicit casting and less type checking. As our aim is to embed the CoCa primitives in the host language in a way that it is in agreement with the general philosophy of the host language, the C++ host language binding has to reestablish the original functionality of the host language. For example, to use C++'s typing facilities, we need runtime type information.

As the bindings are in the upper layer of CoCa, the distinction between shared and distributed CoCa is not made. Sections 7.7.2, 7.7.3 and 7.7.4 discuss the C++, C and Fortran 77 bindings, respectively. The following issues are of importance:

• Handling of the configuration information by CoCa. This is host language independent and therefore treated in a separate section.

• Choice of the job representation and specification of the job type definition. The CoCa core does not specify anything about the job format, except that a job is stored contiguously. This freedom can be used to adapt the job representation to the data representation as is used in the host language, and to model the job definition according to the requirements of the parallel application and such that data are readily usable by the environment. Consequently, the job representation is host language dependent and the job type definition is application dependent.

• Association of jobs with variables in the component code and invocation of the CoCa primitives. This is host language dependent.

• Flexibility versus runtime overhead: should a decision be made at compile time, initialization time or runtime? Should functionality be added using a pre-compiler rather than by extending CoCa? Which parameters should have a fixed value and which ones should be adjustable?

• Robustness. Which responsibility is expected from the application programmer? How much robustness should CoCa provide, again in view of the incurred overhead.
7.7.1 Configuration information

The configuration information (CI) is separated from the application code by storing it in a separate file. For shared CoCa, the CI does not contain information about the component/node mapping, as mentioned before. For shared CoCa the CI includes:

- module/component mapping
- $N\text{buckettext}$ and $N\text{bucketmax}$ for each job type

For distributed CoCa, the CI does not contain information about the mapping of components to processors of a node, as mentioned before. For distributed CoCa, the choice between pull and push mode communication is done by storing a job queue at the consumer or at the producer side. For distributed CoCa, the job type/module association determines the job queue/node mapping, since the module/component mapping and the component/node mapping are also specified. For distributed CoCa the CI includes:

- module/component mapping
- component/node mapping
- job type/module association
- distribution strategy per job type (including range specification and associated predicates)
- $N\text{buckettext}$ and $N\text{bucketmax}$ for each job type

We now discuss how the application programmer presents the CI to CoCa. In general, information can be specified at compile time, at initialization time or at runtime (i.e. dynamically). At compile time means that any change requires recompilation of the application. This is not necessary if the specification is given at initialization time. Specification at runtime means that the application can be changed without it being taken offline. Runtime specification allows also dynamic adaptation to changing conditions and requirements. This flexibility often involves more complex code and runtime overhead. A pre-compiler can decrease the runtime overhead, but decreases the flexibility, as changes of the host language and changes of CoCa require adaptation of the pre-compiler and for every host language a pre-compiler is needed.

Since during the configuration phase, the CI is often changed, specification of the CI at initialization time is advantageous, as it prevents frequent recompilation and linking of the application. After investigation, it turned out that interpretation at initialization time of the CI can be implemented without noticeable runtime overhead, and therefore this option has been chosen. We investigated as well whether runtime specification of the CI was useful and feasible. From all information that is specified in the CI, only the dynamic changing of the number of components is useful to allow better dynamic load balancing. For shared CoCa, dynamic changing of the number of components can be realized without additional runtime overhead and therefore has been implemented. A system to automatically adapt the number of components to the load has not (yet) been implemented. For distributed CoCa, dynamically changing the number of components also requires dynamic adaptation of the job queue allocation and distribution strategy, which incurs much runtime overhead and therefore has not been implemented. A second reason not to implement it, is that Meesters [Meesters95] showed that for this particular case, an application independent implementation of automatic dynamic load balancing is not feasible.

A parser interprets the information in the CI file at initialization time and performs consistency checks on the CI file. It is constructed using Flex [Paxson93] as lexical scanner generator and Bison [Donnelly92] as parser generator. Using a parser generator provides more flexibility. As parsing of the CI file is only done at CoCa initialization time, the parsing performance is not of importance. Below, an example of a CI file for distributed CoCa is shown:
Section 7.7: Host language bindings

NPEs = 9

[**** Modules ****]
module = event_generator PE = 0
module = event_reconstructor PE = 1,2
module = track_fitter PE = 3,4,5,6,7,8

Njobtypes = 4

[**** Jobtype NON_RECONSTRUCTED_EVENT ****]
with event_reconstructor
Nbuckettext = 900 Nbucketmax = 900
strategy = round_robin

[**** Jobtype RECONSTRUCTED_EVENT ****]
with event_generator
Nbuckettext = 300 Nbucketmax = 300
strategy = round_robin

[**** Jobtype TRACK ****]
with track_fitter
Nbuckettext = 20 Nbucketmax = 20
strategy = range_partition, range = (0,100,200,300,400)
predfunc_dd = track_num_pred
predfunc_dd_inv = track_num_pred_inv

[**** Jobtype FIT ****]
with event_reconstructor
Nbuckettext = 20 Nbucketmax = 20
strategy = return

7.7.2 C++ binding

Since C++ is an OO language, a natural job representation is the C++ object, which means that a job is associated with an object variable in the component. This is possible, because (1) data members of an object are stored contiguously in memory and (2) data members of an object are stored separately from the member functions of the object. We would not like to communicate the member functions, as they are not part of the object state. Consequently, a job corresponds to an object, a job type to a class, a data item to a class member. We have the following requirements. For any normal C++ object variable, it should be possible to declare it as a CoCa object variable, without changing the corresponding class definition. This provides transparency, as the application programmer does not have to worry whether he deals with a CoCa object variable or a normal C++ object variable. This translates into two sub-requirements: (1) the CoCa functionality should be hidden to the application programmer as much as possible, and (2) a CoCa object variable should have at least the functionality of a normal C++ object variable.

Adding CoCa functionality to C++ objects

Because a normal C++ object variable can be declared as global, automatic or pointer variable, this should also be possible for a CoCa object variable. As process shared memory can only reside on the process heap, it has to be dynamically allocated. This would require the declaration of a CoCa object as pointer variable. A solution to this problem is as follows (see figure 33). We have an object that we call the proxy that represents the actual job residing in process shared memory. Via the proxy, the actual job can be accessed. The proxy can be declared as global, automatic or pointer variable and therefore behaves as a normal C++ object.

In C++, adding functionality to existing object definitions can be done with inheritance or templates. An implementation of this proxy model using inheritance does not allow to hide the dynamic memory allocation of a job, as inheritance requires the explicit use of a memory allocation operator. Furthermore, inheritance has class level granularity, which means that all
objects of one type have the CoCa functionality. C++ templates work on the object level, rather than the class level, which means that for every object separately, it can be decided whether it needs CoCa functionality or not. C++ templates allow an elegant implementation of the proxy system. The proxy object is an instance of a template class. The requested job type is specified as the template argument. The proxy class definition has member functions corresponding to the CoCa primitives. The template argument is a class and the corresponding class definition does not contain CoCa related statements. This means that any (existing) class can be used to declare a CoCa object variable. Accessing the CoCa functionality defined on the proxy class, and accessing the functionality defined on the template argument, is done by overloading the dot and arrow operator. This method avoids name clashes. The dot operator invokes the CoCa primitives defined on the proxy class; the arrow operator invokes functions defined on the template argument. The following example shows how the application programmer declares a CoCa object variable and invokes functions of the proxy class and functions of the template argument. The name of the proxy class is CoCa and the CoCa object variable is of class type Track. Remark that the Track class definition does not contain CoCa related statements.

```c++
class Track {
public:
    Track();
    ~Track();
    init(const int x, const int y, const int track_num);
    bool track_fitter(CoCa<Fit>& fit);
    bool track_num_pred(int num);
private:
    int the_track_num;
    int the_x;
    int the_y;
};
CoCa<Track> track; // initate CoCa object of type Track
track->init(); // invoke function defined on class Track
track.out();  // invoke function defined on proxy class
```

Using predicates

The predicate functions are defined by the application programmer on the template argument. The predicate mechanism is provided by the CoCa template class. Since the CoCa template class cannot depend on the template argument, the signature of all predicate functions has to be the same and fixed. In accordance with section 7.3.2.1, we allow a predicate function signature with one integer input argument: `bool predfunc(int num)`.

The mechanism to specify predicates as described above does not cause overhead for primitives without predicates. Mechanisms to speed up the predicate execution, like indexes, give a permanent overhead and therefore have not been applied. The `rd` and `in` primitives are two-fold overloaded to allow usage with and without predicate:

```c++
t.rd();
```
Section 7.7: Host language bindings

t.rd(bool predfunc(int num), int num);

CoCa limitations

We discuss the limitations of objects that can be used as CoCa objects with respect to normal C++ objects. CoCa objects cannot have data members that are pointers. This is because (1) a pointer loses its validity if transported to another node, which creates difficulties for move access, and (2) for reference access, if a CoCa object is passed to another component, this component cannot access the object the pointer is pointing to, as it does not necessarily reside in component shared memory. This limitation resembles the pointer problem of RPCs. It might be removed in future versions of CoCa. Note that CoCa objects declared as pointers are possible.

Since the CoCa C++ binding is implemented with templates, all restrictions that hold for templates also hold for CoCa. Since the concept of C++ templates is relatively new and not yet fully crystallized, there are some limitations:

- It is not possible to instantiate an array of objects that is treated as one CoCa object. The reason is that templates do not allow a template instance that is an array of the template argument type. A solution is to use a class as template argument that contains an array, instead of being an array. However, for the current version of CoCa this does not work, since a CoCa object cannot contain pointers as data members and an array is implemented in C++ as a pointer variable. Note that declaring an array of CoCa objects is possible.

- If we want to have different constructors for the template argument class type, for every of these template argument class types, an equivalent constructor for the template class has to be provided. Since any class can be a CoCa object, the CoCa template class cannot depend on its argument class type. We implemented one CoCa template constructor, namely the one that invokes the default constructor of the template argument class type.

Runtime type information

For reasons mentioned before, job types in CoCa are represented by indices. A CoCa primitive invoked as a member function of a given CoCa object variable has to access the job queue corresponding to the type of the CoCa object variable. This means that a CoCa primitive has to determine the type of a CoCa object variable at runtime. Since the type of a CoCa object is a template argument, runtime type information about template argument types is required. The current C++ standard does not support runtime type information. Future releases of C++ will have support for runtime type information by means of runtime functions that return the type of an object. For portability reasons, we decide to not rely on compilers that support runtime type information and therefore we implemented our own support.

A solution is to add to every CoCa primitive an argument with the object variable type, but this requires administration by the application programmer. A better solution was found using static data members. If the CoCa template class has a static data member, for every template argument type, there is an instance of this static member that is shared by the objects of that type. This static member is used to encode type information for usage at runtime.

7.7.3 C binding

In C it is more difficult to hide the proxy structure of jobs than in C++. Therefore, we decide that the application programmer has to deal with pointers to jobs in job space. Furthermore, following the C syntax, a job is an argument of a CoCa primitive, rather than that the CoCa primitive is a member function of the job type, as is the case for C++. As job representation, it is natural to use C structs. The struct definition is part of the component code. As it is more difficult in C to automatically determine the type of a struct, we decide that the struct type has to be
explicitly mentioned as argument of a CoCa primitive, which is the responsibility of the application programmer. For specifying predicates, the same solution as for C++ can be used, which means that the predicate function definition is part of the component code. Consequently, the CI file is similar to that of C++. We show an example of component code written using the C binding:

```c
#include "local_coca.h"
#define TRACK 0
struct track {
    int the_track_num;
    int the_x;
    int the_y;
};
bool track_num_pred(int num, struct track* tr) {
    return(tr->the_track_num==num);
}
void print_track(struct track* tr) {
    printf("track value %d %d\n",tr->the_track_num,tr->the_x,tr->the_y);
}
void assign_track(struct track* tr, int track_num, int x, int y) {
    tr->the_track_num = track_num;
    tr->the_x = x;
    tr->the_y = y;
}
Job tr1;
Job tr2;
create(TRACK,&tr1);
assign_track(tr1->items,55,77,99);
out(TRACK,tr1);
rd(TRACK,track_num_pred,55,&tr2);
print_track(tr2->items);
release(TRACK,tr2);

7.7.4 Fortran 77 binding

The Fortran binding allows existing HEP programs, usually written in Fortran, to benefit from CoCa. Another solution is to convert the existing Fortran code into C code, for example with the f2c tool, but this can be very tedious. Since Fortran usage will decrease in the future, the effort put in the design of the Fortran binding has been relatively small. The Fortran binding is similar to the C binding except for the job representation, the job type definition and the predicate usage.

As job representation we can choose a Fortran construct, but this means that the job type definition would be in Fortran as well. CoCa is written in C and cannot directly understand definitions in Fortran. We choose as job representation the tuple, which consists of a sequence of data items, and which is language independent. For example, this job representation allows job type definitions according to the Zebra format, which means that data are readily usable by applications using the Zebra system. As for the C struct, the tuple type cannot be determined using runtime type information, and therefore the tuple type is an argument of a CoCa primitive, as is the case for the C binding. The tuple data members have to be defined somewhere, which is done in the CI file, although this is not a proper usage of this file, since the job type definition is not part of the configuration. We allow one predicate, namely one that compares the predicate input argument with the value of a tuple data member. Which tuple data member this is, is mentioned in the CI file. We show an example of a job type definition in the CI file for the Fortran binding:
Section 7.7: Host language bindings

[**** Jobtype TRACK ****]
with track_fitter
Nbucketext = 20 Nbucketmax = 20
strategy = range_partition, range = (0,12,38,888) on tuple member 5 [trackid]
Nmembers = 4
[trackid] col_type = int
[eventid] col_type = int
[numrun] col_type = int
[track_data]col_type = bin      col_size = 1400

CoCa is implemented in C, which caused problems building the Fortran binding. The main differences between C and Fortran are that (1) Fortran functions use copy by reference for their function arguments whereas C functions use copy by value and (2) Fortran does not have dynamic data structures. Copy by reference can be simulated in C by using pointers as function arguments. The lack of dynamic data structures implies that a Fortran program can only use job data if these data are stored in a predefined location. Consequently, move access has to be used rather than reference access, thereby causing superfluous copying of jobs. Another issue is that CoCa primitives implemented in C have to be called from a Fortran program and that a Fortran program has to be executed as a component in the CoCa framework written in C. In other words, C has to be called from Fortran and vice versa which can be done using some simple rules.
Chapter 8: Validation

8.1 Introduction

The subject of this chapter is the validation of the CoCa programming model of chapter 5 and its design presented in chapter 7. This is done by tests that assess the performance of CoCa and the effort involved with parallelizing a HEP application using CoCa. Questions are addressed concerning the usefulness of the features provided by CoCa and whether other features are required. The tests are presented in an incremental way and comprise the following. The performance in isolation of the shared CoCa primitives measured on the SC2000 and the performance in isolation of the distributed CoCa primitives measured on the CS-2 are discussed in section 8.2. To show the viability of the CoCa approach in realistic situations, we used CoCa for the parallelization of an existing HEP application, namely the CPREAD event reconstruction program written in Fortran of the CPLEAR experiment. Since this program is written in Fortran, the CoCa Fortran binding could be tested. We had two test cases. The first one in section 8.3 was event farming using CoCa to increase the event throughput. We used it to perform scalability tests on the SC2000 with shared CoCa and on the CS-2 with distributed CoCa. The second test case in section 8.4 was parallel track fitting to decrease the event latency. This test case allowed one to determine the overhead generated by CoCa when used in a realistic example. Also this test case was tried with shared CoCa on the SC2000 and distributed CoCa on the CS-2. In these tests, we did not perform configuration specific optimizations, i.e. we used the general cases.

8.2 Performance of the CoCa primitives

Shared CoCa primitives and intra-node distributed CoCa primitives

Table 19 shows the execution times of the shared CoCa primitives on the SC2000 and the execution times of intra-node distributed CoCa primitives on the CS-2, which has two processors per node. Intra-node distributed CoCa primitives on the CS-2 have the same functionality as the shared CoCa primitives on the SC2000, except that in the first case a component is a thread whereas in the second case a component is a process. The tests were performed with CoCa components that generated negligible CPU load and job queue contention. Tests were performed on an empty machine in the case of the SC2000 and on empty nodes in the case of the CS-2. In chapter 6, we saw that mutexes are relatively time consuming. Therefore, Table 19 shows the execution time of a CoCa primitive subdivided into three parts. It shows the execution time of the envelo ping mutexes and that of the body in between. We remark the following about the results in Table 19:

- The main conclusion is that the execution times of shared CoCa and intra-node distributed CoCa are very short. The measurements were reproducible with a small standard deviation.
TABLE 19: Execution times in µs of isolated shared CoCa primitives on the SC2000 and isolated intra-node distributed CoCa primitives on the CS-2. No predicates were specified.

- If a CoCa primitive is the first one invoked in a component, its execution time is longer than if it is not the first one. This difference is caused by caching effects. A release cannot be the first routine invoked in a component, as it is always invoked after a rd. We used the symbol "-" in the table to denote this.
- We do not show execution times for different job sizes, since both shared CoCa and intra-node distributed CoCa use reference access, which means that the execution times are independent of the job size. Not shown measurements confirmed this.
- We see that the mutexes are less time consuming than in chapter 6. We attribute this difference to caching effects.

**Inter-node distributed CoCa primitives**

Table 20 shows the execution times of the inter-node distributed CoCa primitives on the CS-2. In a test run, a component performed 1000 consecutive CoCa primitive calls. The tests were performed with CoCa components that generated negligible CPU load and job queue contention. As on the CS-2, a user can reserve nodes for its private use, tests have been performed on empty nodes. Because during the tests no other users were running applications, there was not much network traffic. We remark the following about the results in Table 20:

- The times in the table are the minimum observed ones. The reason for this is that the intra-test run variance was small, but the inter-test run variance was quite large. This same effect has been seen in the bare ELAN Channel measurements. We attribute the variance in the CoCa primitive execution time to the variance in the ELAN Channel execution time (a successful CoCa primitive uses three ELAN Channel primitive calls).
- ELAN\(^a\) shows the execution times of the ELAN Channel primitives consecutively executed, used in a test program. ELAN\(^b\) shows the execution times of the ELAN Channel primitives that were used by CoCa. Surprisingly enough there is a difference. In section 6.3 we observed a similar effect, where the ELAN Channel communication time was related to the length of inter-communication intervals, although that effect occurred for much larger inter-communication intervals. The fact that in this case we see the effect also for smaller inter-communication intervals may be caused by additional CPU activity caused by CoCa.
Section 8.2: Performance of the CoCa primitives

A similar effect was observed when we did not perform the CoCa primitive calls in one test run consecutively, but with a 1 ms delay inserted between the CoCa primitives. This is shown in the \textit{out}\textsuperscript{c} column of the table.

<table>
<thead>
<tr>
<th>#bytes</th>
<th>\textit{out}</th>
<th>\textit{rd}</th>
<th>\textit{in}</th>
<th>ELAN\textsuperscript{a}</th>
<th>ELAN\textsuperscript{b}</th>
<th>\textit{out}\textsuperscript{c}</th>
<th>native PVM</th>
<th>non-native PVM</th>
<th>MPI\textsuperscript{d}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>137</td>
<td>159</td>
<td>158</td>
<td>18</td>
<td>33</td>
<td>203</td>
<td>216</td>
<td>2815</td>
<td>257</td>
</tr>
<tr>
<td>10</td>
<td>137</td>
<td>159</td>
<td>158</td>
<td>19</td>
<td>34</td>
<td>207</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>150</td>
<td>169</td>
<td>161</td>
<td>23</td>
<td>40</td>
<td>208</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>173</td>
<td>186</td>
<td>184</td>
<td>41</td>
<td>59</td>
<td>238</td>
<td>298</td>
<td>3490</td>
<td>393</td>
</tr>
<tr>
<td>10000</td>
<td>446</td>
<td>412</td>
<td>422</td>
<td>250</td>
<td>276</td>
<td>489</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23040</td>
<td>790</td>
<td>742</td>
<td>732</td>
<td>575</td>
<td>601</td>
<td>816</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>2731</td>
<td>2578</td>
<td>2594</td>
<td>2432</td>
<td>2457</td>
<td>2798</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textbf{TABLE 20:} Execution times of inter-node distributed CoCa primitives on the CS-2. Figures in \textmu s.

\begin{itemize}
\item\ a. ELAN Channels used in a test program.
\item\ b. ELAN Channels used by CoCa.
\item\ c. CoCa \textit{out} primitive with 1 ms communication intervals.
\item\ d. MPICH MPI v. 1.0.9 (Argonne)
\end{itemize}

- We see that the \textit{out} is faster than the \textit{in} and \textit{rd} for small jobs and slower than the \textit{in} and \textit{rd} for large jobs. We did not expect this, as the communication overhead of the CS-2 network and the memory allocation overhead is the same for all three CoCa primitives.
- The \textit{rd} primitive reads 1000 times the same job during a test run, whereas the \textit{in} primitive retrieved 1000 different jobs. This implies that caching effects could make the \textit{rd} primitive faster than the \textit{in}, although from Table 19 it can be seen that these caching effects, if present, are small with respect to the complete execution time of an inter-node distributed CoCa primitive. We did not observe such an effect in Table 20.
- The last three columns of Table 20 contain execution times of primitives of PVM and MPI [Hauser97]. Despite the richer functionality, CoCa performs better than MPI and PVM.

Distributed CoCa has been designed such that the CoCa primitive execution time should be a constant overhead plus the time to send the job over the network. To verify whether this is indeed the case, we fitted the results of Table 20 to a straight line. The results are shown in Table 21. The fit error shown in Table 21 is the square root of the sum of the squares of the differences between fit and measured values, and this value divided by the number of measurements. If we compare the ELAN\textsuperscript{b} fitted bandwidth with the CoCa primitive fitted bandwidth, we see that for the \textit{in} and \textit{rd} the design aim was realized. For the \textit{out} some bandwidth is lost, an effect for which we did not find an explanation.

<table>
<thead>
<tr>
<th></th>
<th>ELAN\textsuperscript{a}</th>
<th>ELAN\textsuperscript{b}</th>
<th>\textit{out}</th>
<th>\textit{rd}</th>
<th>\textit{in}</th>
</tr>
</thead>
<tbody>
<tr>
<td>start-up latency (\textmu s)</td>
<td>18</td>
<td>35</td>
<td>148</td>
<td>163</td>
<td>161</td>
</tr>
<tr>
<td>bandwidth (MBytes/s)</td>
<td>41.4</td>
<td>41.2</td>
<td>38.3</td>
<td>41.2</td>
<td>41.0</td>
</tr>
<tr>
<td>fit error (\textmu s)</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

\textbf{TABLE 21:} Linear fit of distributed CoCa execution times on the CS-2.

\textbf{Decomposition of the inter-node distributed CoCa \textit{out}}

We also performed measurements that allowed a decomposition of the execution time of the inter-node distributed CoCa \textit{out} on the CS-2, shown in Table 22. The test conditions were as mentioned above. We have the following remarks:
### TABLE 22: Decomposition of a inter-node distributed CoCa *out* on the CS-2.

<table>
<thead>
<tr>
<th>client node</th>
<th>server node</th>
<th>time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>remote node choice using data distribution strategy</td>
<td>receive signal</td>
<td>10</td>
</tr>
<tr>
<td>send signal</td>
<td>receive signal</td>
<td>9</td>
</tr>
<tr>
<td>send out_request</td>
<td>receive out_request</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>getslot</td>
<td>7</td>
</tr>
<tr>
<td>send job (10000 Bytes)</td>
<td>receive job (10000 Bytes)</td>
<td>276</td>
</tr>
<tr>
<td></td>
<td>spawn out_request_thread</td>
<td>(55)</td>
</tr>
<tr>
<td></td>
<td>local CoCa <em>out</em></td>
<td>21</td>
</tr>
<tr>
<td>receive signal</td>
<td>send signal</td>
<td>9</td>
</tr>
<tr>
<td>receive out_reply</td>
<td>send out_reply</td>
<td>35</td>
</tr>
<tr>
<td>wake up component that performed remote <em>out</em></td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

- The individual actions sum up to 442 µs, which is in agreement with the 446 µs for a remote *out* shown in Table 20. The time to spawn the out_request_thread is not included in the 442 µs, since in all previous measurements, the local CoCa primitives that were executed remotely, have been executed by the receive_thread and not by a newly spawned thread.

- The presented times of the individual actions are averages, although we compare these values with the values in Table 20, which are the minimum measured ones. This is because for the execution of a given primitive, not every individual action will have its minimum execution time.

- An anomaly of the CS-2 is that a communication with a not previously used slot has a very large overhead: a 23040 Bytes communication took 50 ms instead of the expected 600 µs. This anomaly has been circumvented by pre-touching of memory: at initialization of distributed CoCa a large chunk of memory is allocated, random data are written into it and subsequently it is freed.

- Further measurements showed that if the job size is varied, only the actual job transfer time changes. This is in agreement with what has been concluded from Table 21.

### 8.3 Test case 1: CREAD event farming

This test case shows CoCa’s suitability to obtain an event throughput increase by farming of an existing HEP event reconstruction application written in Fortran. The reason to perform the farming using both shared CoCa on the SC2000 and distributed CoCa on the CS-2 is to assess the application’s portability that CoCa promised to provide: whether code changes were required to go from the SC2000 to the CS-2. The test case tests the CoCa Fortran binding, as CREAD is written in Fortran. Furthermore, it allows one to perform scalability tests, as farming can be done using many nodes, depending on the event input rate. As well, we can test whether it is useful to separate the realization of a parallel application into a coding and a configuration phase. Therefore, the remainder of the section is structured accordingly:

1. Coding phase to equip CREAD with CoCa communication capability (section 8.3.2).
2. Configuration phase for the SC2000 (section 8.3.3).
3. Tests on the SC2000 to see whether the application output is correct and scalability tests (section 8.3.4).
4. Configuration phase for the CS-2 (section 8.3.5).
5. Tests on the CS-2 to see whether the application output is correct and scalability tests (section 8.3.6).
8.3.1 The CPREAD event reconstruction program

The HEP experiment CPLEAR [Angelopoulos91] produces events at a rate between 200 Hz and 450 Hz, which are reconstructed offline using the event reconstruction program CPREAD. For our tests, we used the version of CPREAD with cputil correction set 4.713, cpread correction set 4.705 and cpgen correction set 4.702, that has been adapted for use on SUN platforms by W. Lu. We refer to this basis program as the CPREAD reference version. CPREAD is a software application consisting of about 230000 lines of Fortran code. It is built of 1051 functions and subroutines. It reconstructs events (see figure 34) produced by the CPLEAR experiment, one event at a time. Event reconstruction in CPREAD consists of a number of reconstruction tasks. Figure 34 shows the tasks and their data dependencies. Each task has some rejection power. CPREAD is a sequential program and the tasks are processed sequentially. To minimize the required processing power for CPREAD, the tasks are processed in order of rejection power, the largest rejection power first, but respecting the data dependencies. This results in the following order: 1. decode & translate, 2. track finding, 3. track fitting, 4. vertex fitting, 5. calorimeter (shower), and 6. calorimeter (cluster). Raw event data plus the accumulating reconstruction information are passed from task to task. All tasks have access to the calibration data, which are constant for a run. A run consists of the events between two consecutive recalibrations.

![Diagram of CPREAD event reconstruction program]

**FIGURE 34**: Left, an example of a CPLEAR event. Right, the data dependencies between the reconstruction tasks.

8.3.2 Coding phase: adaptations to CPREAD for event farming

Adapting a large program like CPREAD by a non-expert is tedious and time consuming. Therefore, we aimed at minimizing the changes made to CPREAD. All data in CPREAD and also its I/O are structured using the Zebra data structuring package [CN94]. The unit of CPREAD I/O is a so-called Zebra block. Each Zebra block contains 23040 bytes and contains a variable number of complete events. It is very tedious to replace the use of Zebra in CPREAD with something more convenient for use with CoCa, so we adhere to the original CPREAD I/O. Therefore, we perform farming at the Zebra block level, meaning that a job is a Zebra block. Consequently, to obtain a CPREAD farm worker, the I/O of the CPREAD reference version had to be changed to I/O established via the CoCa Fortran binding. As mentioned before, the Fortran binding performs explicit moving of data, to which we refer as the Fortran binding overhead.
As events are unrelated, recombination is not an issue. Rejected events are simply discarded by CPREAD, so no additional action by CoCa is required. To avoid unnecessary complexity, reading of calibration constants and recalibrations are accomplished independently of CoCa.

8.3.3 Configuration phase: CPREAD event farming on the SC2000

On the SC2000, shared CoCa is used. Figure 35 shows the configuration of CPREAD event farming on the SC2000. Zebra blocks with raw events are read from disk by event generator components and inserted into the raw event job queue at a specifiable rate that is fixed during a test. CPREAD farm worker components retrieve Zebra blocks with raw events from the job queue with raw events, reconstruct them, and insert the Zebra blocks with reconstructed events into the job queue with reconstructed events. Event writers retrieve Zebra blocks with reconstructed events from the reconstructed event job queue, write them to a file on disk or into a database. As database we used the Oracle7 relational database [Oracle92] from the Oracle corporation. It is more efficient to write multiple Zebra blocks to disk at a time, but this requires the blocks to be contiguously stored. As this is not the case in job space, an event writer moves Zebra blocks to a contiguous segment of memory before it writes them to disk. Writing to Oracle is done with SQL*loader, a tool meant to insert a large amount of data at high speed into Oracle. Parallel I/O is possible by having (1) multiple event generators reading from different disks, (2) multiple event writers writing to different disks, or (3) multiple event writers, each connected to a SQL*loader tool, writing into one Oracle database. Shared CoCa uses pull mode, therefore load balancing is automatically established.

![Diagram of CPREAD configuration](image)

**FIGURE 35:** Configuration on the SC2000 of CPREAD event farming using shared CoCa. Circles are components, cylinders are secondary storage and the job space resides in component shared memory. The arrows with CoCa primitives, indicating the data flow, are not part of the configuration.

CPREAD farm workers have a given set of calibration data loaded and therefore have a state. As a recalibration (a state change) is a relatively time consuming operation, it can be advantageous to have a CPREAD farm worker reconstructing only events that require the same calibration data set. This can be established by predicates on the run number in the way described in section 7.3.2.1. On the other hand, predicates cause the load balancing to be less dynamic than in a situation without. We tried both options. If predicates on the run number were used, we observed that for a test run, there was indeed only one (initial) recalibration per CPREAD farm worker. However, as there are only three calibration data sets used in the test data sample (Table 23), and as the events are ordered according to their run number, this solution resulted in
Section 8.3.3: Test case 1: CPREAD event farming

much job idle time and processor idle time, and hence a bad load balancing. It turned out to be much better not to use predicates, which resulted for one test run in three recalibrations per CPREAD farm worker, but a much better load balancing. The performance tests have been carried out with CPREAD farm workers without predicates.

8.3.4 Tests: CPREAD event farming on the SC2000

We measured the event reconstruction rate as a function of the number of CPREAD farm workers, to show CoCa's scalability, although the coarseness of the parallelism might not reveal possible CoCa communication bottlenecks. For the measurements, we used two test data samples which consisted of events produced by the CPLEAR experiment. Their characteristics are shown in Table 23. We can see that the event reconstruction time is variable. To be able to draw some conclusions, we discuss the test conditions quite extensively:

<table>
<thead>
<tr>
<th></th>
<th>sample 1</th>
<th>sample 2</th>
</tr>
</thead>
<tbody>
<tr>
<td># raw events</td>
<td>1000</td>
<td>3000</td>
</tr>
<tr>
<td># Zebra blocks with raw events</td>
<td>90</td>
<td>273</td>
</tr>
<tr>
<td># events per Zebra block with raw events</td>
<td>11.1</td>
<td>11.0</td>
</tr>
<tr>
<td># accepted reconstructed events</td>
<td>155</td>
<td>493</td>
</tr>
<tr>
<td># Zebra blocks with accepted reconstructed events</td>
<td>93</td>
<td>287</td>
</tr>
<tr>
<td># events per Zebra block with accepted reconstructed events</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>rejection power</td>
<td>0.15</td>
<td>0.16</td>
</tr>
<tr>
<td># recalibrations</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>max. Zebra block rec. time divided by min. Zebra block rec. time</td>
<td>2500</td>
<td>2500</td>
</tr>
<tr>
<td>variance in Zebra block reconstruction time</td>
<td>125%</td>
<td>76%</td>
</tr>
</tbody>
</table>

TABLE 23: Characteristics of the two test data samples used for the measurements.

- **Available CPU power.** Tests were performed on a machine without other users. Between 1% and 2% of the processing power of the machine, which is between 20% and 40% of the power of one processor, was spent on kernel activity and I/O wait time, and was not available for the tests. We only observed an effect of this lost processing power in tests where the machine is fully occupied (so for 20 or more CPREAD farm workers). For tests with less than 20 CPREAD farm workers, the automatic load balancing system of the SC2000 attempts to schedule the processes using the 1% to 2% processing power, to processors not occupied by a CPREAD farm worker.

  It was checked how much CPU power was assigned to CPREAD farm workers. For tests with up to 20 farm workers, all farm workers were assigned 100% CPU power of a processor throughout a run. For tests with more than 20 farm workers, 20 farm workers were scheduled on the CPUs and were each assigned 100% CPU power of a processor, and the non-scheduled ones were waiting to be scheduled. Consequently, all available processing power was assigned to CPREAD farm workers.

  Below, we will see that disk speed, shared memory bus speed and the amount of shared memory were sufficient for all tested configurations (except for the ones using Oracle), meaning that our application is CPU bound.

- **Required disk speed.** At the maximal obtained event reconstruction rate of 385 Hz (with 22 CPREAD farm workers), the input rate of the CoCa farm is 35 Zebra blocks per second and
the output rate is about the same. This corresponds to a total bandwidth that has to be read from and written to disk of 1.6 MBytes/s, which can easily be established by one disk.

It turned out that one event generator and one event writer reading from and writing to the same disk were sufficient for the event input and output up to at least 40 CPREAD farm workers. The CPU power used by the event generator and writer was about 0.2% of the CPU power of one processor (or about 0.01% of the total CPU power), and therefore negligible.

The results shown below are of tests where the event writer stored reconstructed events in a file on disk. We also performed tests where the event writer stored reconstructed events in an Oracle database. The maximum speed at which a SQL*loader tool could write into Oracle was 220 kBytes/s. This was not enough to cope with the complete output, so we used 4 event writers, connected to 4 SQL*loader tools that wrote to different tables in Oracle. The SQL*loader tools and Oracle used a considerable amount of CPU power. No test results with Oracle are shown.

- **Required shared memory bus speed.** Besides the 1.6 MBytes/s that is transferred between shared memory and disk, the Fortran binding causes Zebra blocks to be memory copied one time at the input and one time at the output, using another 1.6 MBytes/s. After this copying, we can assume that a Zebra block resides in the 2 MBytes cache of a processor, which means that the computations of a CPREAD farm worker do not involve the shared memory and therefore do not take bandwidth from the memory bus. The event writer copies the Zebra blocks with reconstructed events to a contiguous segment of memory to prepare them for output to disk, which takes another 0.8 MBytes/s. Hence, the total used bandwidth is about 4.0 MBytes/s, which is far less than the 625 MBytes/s sustained bandwidth offered by the SC2000 according to the manufacturer's specification. We conclude that the memory bus bandwidth is not a bottleneck and can further be neglected.

- **Required shared memory.** CPREAD is not built using shared libraries. A CPREAD farm worker occupies 13 MBytes virtual memory of which 6 MBytes are resident in real memory. 40 farm workers require 40*6 MBytes of memory, which is far less than the total shared memory of 512 MBytes. Indeed, during tests no swapping has been observed.

**Discussion of the results**

To allow comparison of the results of configurations with different numbers of farm workers, for a configuration with \( n \) farm workers, the time has been measured that it takes to reconstruct \( n \) times the 1000 event sample. This avoids that different event samples with different properties have to be used, or that if one event sample is used, a test of a configuration with many CPREAD farm workers, lasts too short to be reliable, whereas a test of a configuration with few CPREAD farm workers lasts too long to have a machine with stable resources, e.g. no other users. Consequently, during a test, many instances of the same event were present. During tests, job queues were not allowed to become empty, since this would have resulted in CPREAD farm worker idle time.

About 10 measurements for every configuration have been performed. They showed a variance of less than 2 percent. For clarity, no error bars have been drawn in the graphs. Figure 36 shows the results of event farming on the SC2000. It shows a graph with the event reconstruction rate as a function of the number of farm workers when CoCa is used, to which we refer as the CoCa farm. It also shows a second graph with the event reconstruction rate of \( n \) independent CPREAD reference applications started at the same time, to which we refer as the reference farm. Each CPREAD reference farm worker reads from a file the same event set in the same order and processes the events sequentially. This second graph allows us to investigate the scal-
Section 8.3: Test case 1: CPREAD event farming

ability of CoCa itself, as the overhead generated by properties of the SC2000 is the same for the CoCa and the reference farm.

We also plotted a CoCa scale-up graph and a reference scale-up graph. In a scale-up graph, the event reconstruction rate of one CPREAD worker (CoCa farm worker or reference farm worker) is multiplied by the number of workers and plotted against the number of farm workers. A scale-up graph assumes full linear scalability with respect to the one CPREAD farm worker case. A comparison of the scale-up graphs and the measured graphs, allows statements about the scalability as a function of the number of farm workers, although effects other than caused by CoCa might influence the scalability. We remark the following:

![CPREAD event farming on the 20-node SC2000](image)

**FIGURE 36: CPREAD event farming on the SC2000.**

1. The CoCa farm performs better than the reference farm. This was not expected, since CoCa farm workers might have idle time caused by the contention to access the single job queue, whereas the CPREAD reference farm workers obtain their jobs separately.

   This difference may be explained as follows. An advantage of the CoCa farm with respect to the reference farm is that the event generator performs pre-fetching of the jobs (job queues are never empty during a test run), and the event writer allows asynchronous writing of the results to disk, which means that a CoCa CPREAD farm worker does not become blocked on disk I/O. In contrast, a reference farm worker performs the (synchronous) I/O itself and thus becomes blocked on disk I/O. On the other hand, since we did many tests with the same event sample, the sample will reside in the OS disk cache and does not have to be read from disk, thereby avoiding waits for disk I/O. Therefore, there will not be much component idle time in the reference farm caused by disk I/O for retrieving the jobs. In other words, for the reference farm, the OS cache has a similar functionality as CoCa for the CoCa farm.

2. A good test to see whether CPREAD farm workers have idle time is to allocate more than one CPREAD farm worker per processor, which allows exploiting the computation/communication overlap. If the performance is better than in the one component per processor case, there was component idle time. Tests with the CoCa farm with between 21 and 25 CPREAD farm workers indeed show a better performance than the 20 farm worker case, which indicates that there is component idle time.
The performance of the reference farm does not improve above 20 farm workers, which means that there was no component idle time. Concluding, component idle time in the reference farm seems not to be a feasible explanation for the CoCa farm performing better than the reference farm.

3. To make statements about the scalability, we compare the CoCa farm measurements with the CoCa farm scale-up graph. We see that the CoCa farm performs better than the one-worker case. There is a super-linear speedup and therefore a very good scalability. We did not find an explanation for this effect, but we should take into account that, since the scale-up graph is an extrapolation, the measurements errors of the one CPREAD worker case are multiplied with the number of CPREAD farm workers.

The comparison of the reference farm measurements with the reference farm scale-up graph shows the expected behavior: the performance per worker becomes slightly less as the number of workers increases, which means that the results are nearly linearly scalable.

4. We see that for the CoCa farm of 30 or more farm workers the performance degrades. However, for the reference farm, the performance remains constant. A possible explanation for the degradation might be context switching overhead, as there is more than one process per processor. However, in that case we should have seen the effect also for the reference farm, so context switching overhead is not a feasible explanation. Another possible explanation might be the increased contention on the job queue. However, the contention tests of chapter 6 did not show an increased contention, although these tests were not performed on a fully loaded machine.

5. The overhead per job for a CPREAD farm worker caused by CoCa consists of: an in to obtain the job (75 μs) and a memory copy of 23040 bytes caused by the CoCa Fortran binding (4000 μs). Furthermore for 15% of the inputs there is an output and the output is about 7 times as large, which requires a memory copy of 7 * 0.15 * 23040 bytes (4000 μs) caused by the CoCa Fortran binding, plus an out (75 μs) to insert the result in the job queue. The job access contention is negligible, as we have seen in chapter 6. So the communication overhead per job is about 8 ms. The reconstruction time per job is about 650 ms. We see that the communication overhead is about 1% of the time spent in computation, which means that the parallelism is very coarse grained.

For the reference farm, the communication overhead per job consists of a memory copy of 23040 bytes from the OS disk cache to the CPREAD farm worker to obtain the job, and a memory copy to the OS disk cache of 7 * 0.15 * 23040 bytes for the output. We conclude that the communication overhead for the CoCa farm and reference farm are comparable.

6. The reconstruction results of the CoCa farm were identical to those obtained by using one single CPREAD reference version, so the parallelization was successful.

Concluding remarks. We observed two unexpected effects, a super-linear speedup and a performance of the CoCa farm that is better than that of the reference farm. We expect that both effects are caused by the OS. The most likely explanation is that retrieving data from the OS cache is less efficient than retrieving data from shared memory that has been written to by some process. [Argante95/2] describes similar measurements and results on a 8-node SC2000.

8.3.5 Configuration phase: CPREAD event farming on the CS-2

The CPREAD code used for event farming on the SC2000 could be used without modification on the CS-2, which confirmed that CoCa provides the promised portability. Since distributed CoCa uses move access, the additional copy action of a job caused by the Fortran binding is not necessary and could have been removed. As this would lead to non-portable code, we did not
Section 8.3: Test case 1: CPREAD event farming

As we want to use multiple nodes on the CS-2, we have to use distributed CoCa. The configuration looks similar to that used on the SC2000, with the difference that on the CS-2 data distribution is an issue. This is described below.

As for the SC2000 tests, also for the CS-2 the number of recalibrations performed by a farm worker can be reduced by having a CPREAD worker only reconstruct events of one run. For the CS-2 this can be done by using predicates and range partitioning on the run number. We tested this option to verify that it works correctly, but for the same reasons as mentioned for the SC2000, it results in a very uneven distribution of the events, and hence a bad load balancing. Therefore, for the performance tests we used CoCa primitives without predicates for the input. Consequently, push mode with round robin is the best choice, as this results in an even distribution and a small CPREAD farm worker idle time, as the computation/communication overlap allows that, while a CPREAD farm worker is processing a job, the receive_thread receives new jobs.

Since the CPREAD farm workers themselves perform the synchronous out operations of the results, a remote out would lead to a longer CPREAD farm worker idle time than a local CoCa out. As the event writer does not specify predicates, pull mode with round robin strategy is appropriate for the output, as this will result in sequential searches of the event writer. On the CPREAD farm worker nodes, the receive_thr handles the event writer requests, thereby obtaining computation/communication overlap. Figure 37 shows the configuration for CPREAD event farming on the CS-2 using distributed CoCa.

![Configuration diagram](image)

**FIGURE 37:** Configuration on the CS-2 of CPREAD event farming using distributed CoCa. The arrows with CoCa primitives, indicating the data flow, are not part of the configuration.

### 8.3.6 Tests: CPREAD event farming on the CS-2

Tests similar to the ones on the SC2000 have been performed using the same test data samples. The test conditions were as follows:

- **Available CPU power.** Tests with up to 28 CPREAD farm workers were possible, as 28 nodes of the CS-2 were available for tests and not more than one CPREAD farm worker per node is possible, as a component is a thread. Tests have been performed on empty nodes, and during tests there was not much network traffic, as no other users were running applications.
Below, we will see that network and disk speed are not bottlenecks for all tested configurations, meaning that our application is CPU bound.

- **Required network and disk speed.** It turned out that one event generator and one event writer offered sufficient I/O bandwidth to cope with the reconstruction rate of at least 28 farm workers.

  The maximal obtained event reconstruction rate of 560Hz (with 28 CPREAD farm workers) generates 2.3 MBytes/s network traffic which is distributed among multiple links, where each link has 50 MBytes/s bandwidth. We conclude that the CS-2 network will not suffer from congestion. 2.3 MBytes/s disk I/O can be obtained using one disk.

Oracle was not available on the CS-2 and could not be tried.

**Discussion of the results**

Figure 38 shows the results of event farming on the CS-2 (also described in [Argante95/1]). It shows the event reconstruction rate as a function of the number of CPREAD farm workers when CoCa is used. It also shows the CoCa scale-up graph and reference farm scale-up graph. Measurements of the same configuration showed a variance of about 5 percent; no error bars have been drawn for clarity.

In a distributed memory machine, nodes are more independent than in a shared memory machine. The only influences that a node has from its environment are the event generator and event writer accessing the job queues. Moreover, a job queue is not shared by multiple farm workers. Consequently, the job queue access contention is low. Therefore, we expect a linearly scalable speed-up, meaning that the graph with CoCa measurements is equal to the CoCa scale-up graph. Indeed the graph with CoCa measurements is a straight line, but its slope is not equal to the CoCa scale-up graph. However, the scale-up graph is an extrapolation, which means that measuring errors of the one CPREAD farm worker case are multiplied by the number of CPREAD farm workers. If we take this effect into account, the difference between the CoCa measurements and the CoCa scale-up graph is within the error margin.

The reference farm scale-up graph also suffers from the error amplification, but the difference between the CoCa measurements and the reference farm scale-up graph is outside the error margin. We expected the performance of the reference farm with one CPREAD farm worker to be equal to the performance of the CoCa version with one CPREAD farm worker for the following reason. The CoCa version can read the Zebra blocks from job queues that reside in memory. The CS-2 has a OS disk cache per node and a general disk cache. After several tests with the same data sample, the Zebra blocks are cached in the per node OS disk cache. So also the CPREAD reference farm reads events from memory. We expect that the observed difference is caused by properties that are specific to the CS-2. The CS-2 manual requires that applications that do not use the ELAN communication network, e.g. the CPREAD reference farm, should be started in the so-called sequential mode. Possibly, this mode allows more processor time to be available for user applications.

Since distributed CoCa does not allow to run more than one CPREAD farm worker per node, we cannot make any statements about the CPREAD farm worker idle times.

Figure 38 also shows that event farming on the CS-2 can cope with the event production rate of the CPLEAR experiment of maximally 450 Hz. Hence, online event reconstruction is feasible. As a comparison, the GPMIMD ESPRIT project attempted to build a system for online event reconstruction in the CPLEAR experiment [Heeley95]. Their system, specially built for this task, consisted of a dedicated 50 node T9000 Transputer processor farm connected with 20 C104 packet routing switches. The system was programmed on a very low abstraction level to
achieve maximal performance, and because of the unreliable and incomplete state of the native software. They obtained an event reconstruction rate of 64 Hz, which was only useful for offline event reconstruction. The low rate was due to the low CPU performance of the T9000 processor.

![Graph showing CPREAD event farming on the CS-2](image)

**FIGURE 38: CPREAD event farming on the CS-2 using distributed CoCa.**

### 8.4 Test case 2: CPREAD parallel track fitting

Track fitting is one of the tasks in the reconstruction of an event. As the fitting of the tracks of one event are independent tasks, they can be performed in parallel, and the event reconstruction latency can be decreased. This parallelism is more fine grained than event farming, which means that the communication overhead caused by CoCa has a relatively larger effect on the performance, which makes parallel track fitting a good test case for CoCa. Moreover, since tracks of one event are related and have to be correctly recombined after fanout, the recombination functionality of CoCa can be tested. A third reason to choose parallel track fitting as a test case is that Schiefer already performed the code splitting of the CPREAD program [Schiefer95].

Like for event farming, parallel track fitting tests the CoCa Fortran binding. Again, we show the application’s portability that CoCa promises to provide, by testing parallel track fitting with both shared CoCa on the SC2000 and distributed CoCa on the CS-2. Again, the realization of a parallel track fitting is separated into a coding and a configuration phase, resulting in the following section lay-out:

1. Coding phase to equip CPREAD with the CoCa communication capability (section 8.4.1).
2. Configuration phase for the SC2000 (section 8.4.2).
3. Tests on the SC2000 to see whether the application output is correct, and measurements of latency decrease and code replication (section 8.4.3).
4. Configuration phase for the CS-2 (section 8.4.4).
5. Tests on the CS-2 to see whether the application output is correct (section 8.4.5).
8.4.1 Coding phase: adaptations to CPREAD for parallel track fitting

In the CPREAD reference, data produced by a task of the reconstruction process have to be (logically) transferred to the next task. This is done using the Zebra package and goes as follows (see figure 39). If a reconstruction task finishes, it writes its results from the Fortran common blocks into Zebra banks. The next reconstruction task can access all Zebra banks containing raw data and accumulating reconstruction information, and writes the data required for its task from the Zebra banks back into common blocks. This modular structure, in which the interface between modules is solely defined by Zebra banks, offers a relatively easy separation of the reconstruction tasks into components.

![Modular architecture of CPREAD](image)

FIGURE 39: Modular architecture of CPREAD\(^a\).

a. Courtesy of Schiefer.

Our parallel track fitting is modeled as a master-slave application. The master, CPREADmain, communicates tracks to the slaves, the trackfit worker, using CoCa. The slaves communicate their results back to the master. The changes to CPREAD to obtain parallel track fitting are very simple on the conceptual level (see figure 39 and 40). The code of the reference version consists of an initialization part, followed by a loop of which each cycle performs the reconstruction of one event. The CPREADmain component and the trackfit worker component are based on the CPREAD reference. CPREADmain is a CPREAD reference version, in which the loop that fits the tracks of one event is replaced by CoCa primitives that insert the tracks into job space. These primitives have to be inside a transaction, to ensure correct recombination. The information about the number of subjobs is already available in the recombining component and does not have to be part of the job. In the transaction, CoCa \textit{out} primitives perform the fanout of the tracks (consisting of hits), one track per job, and then CoCa \textit{in} primitives perform the recombination of the fit results, one fit result per job.

A CPREAD trackfit worker version starts with the same initialization as the CPREAD reference, so that it can access the same common blocks, but the loop over the events is replaced by a loop of which the loop body performs a CoCa \textit{in} to retrieve a track, a call to the trackfit algorithm, and a CoCa \textit{out} to insert the fit result in job space. As a trackfit worker is a transfer component, these statements are not part of a transaction.

We also tested parallel track fitting without using CoCa's automatic recombination facility. In this case, recombination was performed by using predicates and adding fields to the job definition. It turned out that this was very well possible both on the SC2000 and CS-2, but as the difference between shared and distributed memory was visible in the application code, portability was lost. Also, we had to change the distribution strategy such that it could choose the destination node based on the componentid of the destination component.

Since a Zebra bank uses a pointer system that is specific to that Zebra bank, Zebra banks cannot be transferred between components, since this would destroy referential integrity. The Zebra package offers routines to copy a part of a Zebra bank into a Fortran common block and vice versa. The common block is then communicated using CoCa. These Zebra routines incur a
large overhead in CREADmain and the trackfit workers, to which we refer as the Zebra overhead. Attempts to avoid this overhead had negative results.

The CREAD program can reject a complete event based on the decision of one reconstruction task. The modular structure of CREAD already has a rejection structure that is very easy to adapt to the parallel track fitting case. The CREAD reference trackfit task returns an empty Zebra trackfit bank to indicate that a trackfit result is invalid. The code that embeds the trackfit loop rejects the event when it encounters an empty trackfit bank. Consequently, no additional functionality of CoCa is required.

The trackfit task uses the calibration data. This means that if a track is received of which the corresponding event belongs to a different run than the event corresponding to the previously fitted track, a recalibration has to be performed in the trackfit worker. The trackfit worker should recognize when it should perform a recalibration, which is done by sending the run number of the corresponding event together with the track. The trackfit worker compares the run number with the run number of its loaded calibration data and in case of a difference performs a recalibration.

### 8.4.2 Configuration phase: CREAD parallel track fitting on the SC2000

For the SC2000 it is appropriate to use shared CoCa. The configuration is identical to CREAD event farming, except that a track job queue and a fit job queue are added to job space and a number of trackfit workers are started. CREADmain inserts tracks in the track job queue and retrieves results from the fit job queue. Trackfit workers retrieve jobs from the track job queue and insert their results in the fit job queue.

To be able to determine whether a configuration of multiple CREADmains and trackfit workers provides a correct result, we need reproducibility. We decide that every CREADmain worker should receive the same data sample. We do this by adding a component identifier field to a job and have a predicate issued by a CREADmain that acts on this component field. The event generator consecutively inserts $n$ of the same events, each with another component identifier, if there are $n$ CREADmains. It seems natural to define the events of the 1000 event sample as a related subjobs, in which case the recombination functionality of CoCa could perform the required functionality. However, this is not possible because (1) it is not an example of recombination of related subjobs, (2) the CoCa recombination feature does not allow interleav-
ing of CoCa primitives of different transactions in the code of one component (see section 7.6.1).

Since trackfit workers use the calibration data, they have a state and with the help of predicates on the run number, the number of recalibrations can be decreased. We tested this option to see whether it works correctly, but for performance tests we used trackfit workers that have CoCa primitives without predicates, for the same reason as mentioned before.

8.4.3 Tests: CPREAD parallel track fitting on the SC2000

Measurement of the latency decrease

The goal of the measurements was to determine the latency decrease that parallel track fitting can establish. We used one CPREADmain with four trackfit workers. We first attempted to measure averages over multiple events, but it turned out that the variance in the reconstruction time of different events and the variance in the measurements of the same event were too large to capture subtle details. Therefore, we decided to perform detailed measurements on one arbitrarily chosen event of the 1000 event sample. To avoid any influence of transient effects, we did the measurements on an event somewhere during the processing of the 1000 event sample. The test conditions were identical to the ones described before for the SC2000. Table 24 shows the results of the measurements. To determine the latency decrease only the measurements in CPREADmain have to be considered. Note that the in primitive in CPREADmain retrieving a fit result blocks, until a result arrives and therefore includes the idle time of CPREADmain waiting for results. We have the following remarks:

<table>
<thead>
<tr>
<th></th>
<th>measured</th>
<th>measured (corrected)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(µs)</td>
<td>(%)</td>
</tr>
<tr>
<td>Zebra overhead</td>
<td>837</td>
<td>12.7%</td>
</tr>
<tr>
<td>Fortran binding overhead</td>
<td>635</td>
<td>9.6%</td>
</tr>
<tr>
<td>CoCa overhead</td>
<td>597</td>
<td>9.1%</td>
</tr>
<tr>
<td>idle time</td>
<td>4504</td>
<td>68.4%</td>
</tr>
<tr>
<td>administration overhead</td>
<td>14</td>
<td>0.2%</td>
</tr>
<tr>
<td>total</td>
<td>6587</td>
<td>100%</td>
</tr>
<tr>
<td>average trackfit time (without Zebra or Fortran binding overhead)</td>
<td>(16737/4 =) 4184</td>
<td></td>
</tr>
<tr>
<td>maximum trackfit time (without Zebra or Fortran binding overhead)</td>
<td>4775</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 24: Results of the measurements on the SC2000 of parallel track fitting of a 4-track event. The first six rows are measured inside CPREADmain, the last two in the trackfit workers.

1. We see that the CPREADmain idle time is shorter than the trackfit worker maximum trackfit time, although the CPREADmain idle time was expected to be equal to the trackfit worker maximum trackfit time plus the Fortran binding, Zebra, CoCa and administration overhead. This is caused by the optimization to interleave CoCa calls with the routines that cause the Zebra overhead: while CPREADmain is suffering from the Zebra and Fortran binding overhead of the second, third and fourth track, the first track is already being fitted by a trackfit worker. In addition, as for this particular event, the most time consuming track
Section 8.4: Test case 2: CPREAD parallel track fitting

is inserted first in job space and therefore selected first by a trackfit worker, the CPREAD-
main idle time is shorter than the trackfit worker maximum trackfit time.

This means that part of the time that CPREADmain spends on Zebra and Fortran binding
overhead would become CPREADmain idle time, if the Zebra and Fortran binding overhead
were removed. The 3rd and 4th column in the table are corrected for this difference of 4775-
4504 = 271 µs. Note that the calculation does not include the various overheads in the track-
fit worker.

2. For the total trackfit time in the sequential case, we use the sum of the trackfit times (without
any overhead included) measured in the trackfit workers, which is 16737 µs. The total time
spent by CPREADmain in the trackfit module is 6587 µs. This means that the established
speedup is 2.54. For the estimated speedup without Zebra and Fortran binding overhead, we
use the corrected values, which gives 16737/(597+4775+14) = 3.11. The theoretical speedup
is the total trackfit time divided by the maximum trackfit time without taking any overhead
into account, which gives 16737/4775 = 3.51. The theoretical speedup is not 4, because the
trackfit times are not equal, which gives a non-perfect load balancing.

Schiefer performed similar tests [Schiefer95]. Parallel track fitting was performed using the
Mon a Lisa [Schneider93] parallel programming paradigm on a T805 Transputer network.
He measured a speedup of 1.5, where the theoretical speedup was 3.5. The same application
run on a T9000 Transputer network [INMOS93], using the bare T9000 native communica-
tion library that is known for its low communication latency, yielded a speedup of 3.0, again
where the theoretical speedup was 3.5. Schiefer’s application suffers from the Zebra over-
head, but not from the Fortran binding overhead. Therefore, if Schiefer’s results are to be
compared with ours, his figures have to be compared with a figure somewhere between 2.54
and 3.11. Comparison should anyhow be done with care as external conditions like the pro-
cessed event set were not identical.

3. The reconstruction results produced by CPREAD parallel track fitting were identical to
those produced by the CPREAD reference version, so the parallelization was successful.

Measurements with multiple CPREADmains

For combined event farming and parallel track fitting, the CoCa recombination functionality
should ensure correctness. Tests showed that the results were indeed correct. For configurations
with multiple CPREADmains used with shared CoCa, the recombination feature of CoCa
causes searches in the job queues to find jobs with correct jobids. To investigate the overhead
of these searches, the latency of the same event as used for the previous test was measured for
different configurations. The results are shown in Table 25. We see that there is no significant
difference between the one master and multiple master cases. Hence, the overhead of the
recombination feature is negligible.

<table>
<thead>
<tr>
<th>configuration</th>
<th>measured latency (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 master 4 workers</td>
<td>6.9</td>
</tr>
<tr>
<td>2 masters 4 workers</td>
<td>6.8</td>
</tr>
<tr>
<td>2 masters 8 workers</td>
<td>6.9</td>
</tr>
<tr>
<td>4 masters 4 workers</td>
<td>6.4</td>
</tr>
<tr>
<td>4 master 6 workers</td>
<td>6.6</td>
</tr>
<tr>
<td>4 master 8 workers</td>
<td>6.5</td>
</tr>
</tbody>
</table>

TABLE 25: Measurements to show whether the CoCa recombination feature incurs
overhead.
8.4.4 Configuration phase: CREAD parallel track fitting on the CS-2

As was the case for CREAD event farming, the CREAD track fitting code could be used on the CS-2 without modification. Figure 41 shows the configuration of parallel track fitting on the CS-2 using distributed CoCa. For the tracks we choose push mode with round robin distribution strategy, since the trackfit workers do not specify predicates, which means that a trackfit worker can use any job that it finds locally. As a consequence, a remote out generates less network traffic (and therefore also a reduced communication latency) than a remote in, as the remote in might have to access multiple nodes to find an appropriate job. Push mode provides a second reduction of the trackfit worker idle time in comparison to pull mode, as it does not suffer from the communication latency. The computation/communication overlap allows the reception of new jobs, while a trackfit worker is processing a job.

![Diagram of CREAD parallel track fitting on the CS-2](image)

FIGURE 41: Configuration on the CS-2 of CREAD parallel track fitting using distributed CoCa. The arrows with CoCa primitives, indicating the data flow, are not part of the configuration.

For the same reason, we choose push mode for the fit job queues. The return data distribution strategy is compulsory to ensure correct results. Since maximally one CREADmain per node is possible, and because events of one CREADmain are processed sequentially, the predicates acting on the jobid that the recombination feature automatically performs on jobs in the local job queue are redundant for CREADmain to retrieve correct jobs, since all jobs that arrive in its local fit job queue are appropriate for retrieval by that CREADmain. However, since these predicates do not result in searches their overhead is negligible.

To be able to determine whether a configuration of multiple CREADmains and trackfit workers provides a correct result, we need reproducibility. This is established by having the event generator consecutively insert \( n \) of the same events in round robin mode, if there are \( n \) CREADmains. No predicates were required, as was the case for the SC2000.

The same argument as before holds for predicates on the run number to decrease the number of recalibrations.

8.4.5 Tests: CREAD parallel track fitting on the CS-2

We only performed functionality tests. For any number of CREADmains and any number of trackfit workers, the results turned out to be identical to the reconstruction results of the CREAD reference version.
Section 8.5: C++ binding

Measurement of code replication

On distributed memory machines, data parallelism results in replicated code, whereas algorithmic parallelism results in code distributed among the components, although parts of the code, like initialization code have to be replicated. The parallel track fitting application allows us to investigate code replication in practice. Except for the required code for initialization, calibration and termination, a CPREAD trackfit worker contains some unnecessary code. This is caused by the complicated structure of CPREAD, which made it difficult to remove all redundant code. Table 26 shows the results measured on the CS-2. We see that a CPREAD trackfit worker is indeed smaller than the original CPREAD code, but the CPREADmain and trackfit worker together show a fair amount of code replication. The large difference between the CPREAD executable size and the process size in virtual memory is caused by the Fortran global array variable of 4 MBytes used by Zebra. The figures for distributed CoCa are for an empty job space with no applications attached to it.

<table>
<thead>
<tr>
<th>in MBytes</th>
<th>CPREAD reference</th>
<th>distributed CoCa</th>
<th>CPREADmain + CoCa</th>
<th>trackfit worker + CoCa</th>
</tr>
</thead>
<tbody>
<tr>
<td>executable size</td>
<td>3.9</td>
<td>0.2</td>
<td>4.1</td>
<td>2.0</td>
</tr>
<tr>
<td>process in virtual memory</td>
<td>8.3</td>
<td>4.2</td>
<td>12</td>
<td>8.9</td>
</tr>
<tr>
<td>part of process resident in real mem.</td>
<td>5.0</td>
<td>4.1</td>
<td>8.4</td>
<td>5.4</td>
</tr>
</tbody>
</table>

TABLE 26: CPREAD code size comparison on the CS-2.

8.5 C++ binding

The aim of this section is to test the CoCa C++ binding. We show that a parallelization using the C++ binding results in very concise and simple application code, as not many statements are required for administration. With measurements we show that the C++ binding incurs negligible overhead with respect to the corresponding bare CoCa primitives. For the measurements we used a test program, as HEP applications written in C or C++ are not yet available in a usable form. The test program has combined event farming and parallel track fitting. Figure 42 shows its most important part, that consists of an EventGenerator, a master called EventReconstructor, and a slave called TrackFitter. With use of shared CoCa, all communications use reference access. No copying is performed, if subjobs are fanned out that were part of a previously retrieved job. To allow this, we overloaded the CoCa create primitive. The new version has a job type as argument type and the yielded job is the job that is used as the argument. This avoids the copying of jobs. As well, no copying is performed if recombined subjobs are embedded in a previously created data structure. To allow this, we added a CoCa assign function.

Table 27 shows the execution times of the CoCa primitives measured in the test program run on the SC2000 and using shared CoCa. We can see that the results are indeed similar to those presented in section 8.2.

These fast communication times should allow good speedup factors. To test this, we performed measurements of the latency decrease on the SC2000 with a fake event. To compare these measurements with those of CPREAD parallel track fitting in section 8.4.3, we chose a 4-track fake event with the trackfit time per track (i.e. the time that fitfunc spends; see figure 42) similar to that of CPREAD parallel track fitting. This means that this test is a similar to the test of section 8.4.3, but now without the Fortran binding overhead and the Zebra overhead. The tracks of the fake event had trackfit times 5720, 4984, 4982 and 5083 μs, and were distributed in this order. The sum of the trackfit times is 20769 μs. With four trackfit components, parallel trackfitting of this event took 5907 μs, which means that a speedup of 20769/5915 = 3.52 is
class Fit {
    public:
        Fit();
        void set_fitid(int id);
        bool fitid_pred(int id);
    private:
        int the_fitid;
        // fit data
    
    class Track {
        public:
            Track();
            void fitfunc(CoCa<Fit>& fit);
            bool trackid_pred(int id);
        private:
            int the_trackid;
            // track data
    
    class Event {
        public:
            Event();
            void reconstruct();
            void set_eventid(int eventid);
            void set_trackvalues();
        private:
            int eventid;
            int Ntracks;
            Track track[4];
            Fit *fit[4];
            // other event data
    
    void EventGenerator() {
        CoCa<Event> event;
        for(int i=0;i<NEVENTS;i++) {
            event.create();
            event->set_eventid(i);
            event->set_trackvalues();
            event.out();
        }
    }
    void EventReconstructor() {
        CoCa<Event> event;
        while(TRUE) {
            event.in();
            event->reconstruct();
        }
    }
    void TrackFitter() {
        CoCa<Track> track;
        CoCa<Fit> fit;
        while(TRUE) {
            track.in();
            fit.create();
            track->fitfunc(fit);
            fit.out();
        }
    }
    void Event::reconstruct() {
        CoCa<Track> tr;
        CoCa<Fit> fi;
        for(int d=0:d<Ntracks; d++) {
            tr.create(&track[d]);
            tr.out();
        }
        for(d=0:d<Ntracks; d++) {
            fi.in();
            fit[d]=fi.assign();
        }
    }

FIGURE 42: Test program for the CoCa C++ binding.

| create  | 11 |
|------------------|
| overloaded create | 13 |
| out          | 11 |
| in           | 14 |
| assign      | 7  |

TABLE 27: Execution times in μs of CoCa primitives using the C++ binding.

established. The theoretical speedup is 20769/5720 = 3.63. We see that without the Fortran binding and Zebra overhead, the overhead induced by the parallelization is very small and that a speedup close to the theoretically possible speedup is achieved.
Section 8.6: Conclusion and future tests

8.6 Conclusion and future tests

Most of the features provided by CoCa have been tested on their usefulness and we assessed which features could be added. In this section we summarize some of our observations.

1. We have shown that the CoCa primitives are fast. Our main design goal, high performance, has been achieved. Despite their richer functionality, on the CS-2, the CoCa primitives performed better than native versions of PVM and MPI. We showed that the separation in a coding and configuration phase turned out to be useful, as we managed to combine a good performance of the parallel application with hardware independence of the application code. We did not perform configuration dependent optimizations in the application code. Since changing the configuration did not require recompilation or relinking, much development time was saved for a large application as CPREAD. Our claim "readily usable by the environment" is not completely realized, as for the event farming test case event writers had to move Zebra blocks with reconstructed events to a contiguous segment of memory, before they could be efficiently written to disk. However, for future applications, this overhead can be avoided.

2. Using our CoCa system together with standard hardware and software, we obtained event farming based on the CPREAD reconstruction program. While retaining a high level of abstraction and without optimization of the given test case, the performance of this system can cope with the CPLEAR event production rate, which makes online event reconstruction feasible. We have to take into account that the coarse granularity of parallelization makes it relatively easy to keep the overhead low. On the other hand, the GPMIMD system [Heeley95], consisting of hardware and software specially designed and assembled to achieve online event reconstruction for the CPLEAR system by means of event farming, only achieved a performance that is about one tenth of ours, although the software was optimized on a very low abstraction level for the hardware and application under consideration. We should be aware that is a comparison from the perspective of the user of the system, i.e. the physicists. They are interested in the performance of the complete system, and not interested whether the quality of the software or hardware is responsible for the result.

3. Using our CoCa system together with standard hardware and software, we obtained CPREAD event reconstruction with parallel track fitting. While retaining a high level of abstraction and without optimization of the given test case, our performance was much better than that of Mona Lisa [Schiefer95] used for the same parallel application, although Mona Lisa is specially designed to parallelize applications according to the master/slave pattern. Only by programming the parallel application on a very low abstraction level and by optimizing for the hardware under consideration, Schiefer obtained a better speedup than we did, although the difference is partly explained by (Zebra and Fortran binding) overhead incurred because of the use of an existing application. As in item 2, also here we compare two systems from the perspective of the user, which means that we should interpret the result of the comparison with care.

4. As the test cases used relatively old, existing HEP applications written in Fortran, we encountered many complications during the parallelization, that resulted in a decreased level of abstraction and an increased overhead. It also resulted in very long development times of these parallel applications, but this was also caused by the non-familiarity with CPREAD. Our newly designed test program written in C++ does not suffer from these complications, which resulted in concise, relatively simple application code, without many bookkeeping statements and a low overhead of the CoCa primitives. Measurements with this test program with test conditions similar to those of CPREAD parallel track fitting, showed a speedup that was better than Schiefer’s speedup and close to the theoretically possible speedup.
5. Use of CoCa's recombination facility reduced the complexity of the application code, as less bookkeeping code was required. Furthermore, it resulted in hardware independence of the application code, so that identical code could be used for shared and distributed memory machines. No performance penalty caused by the recombination facility was measured. Tests without CoCa's recombination facility were also successful, but hardware independence and consequently portability were lost. CoCa's recombination facility does not allow multiple control flows in one component, i.e. it does not allow interleaving of CoCa statements of different transactions in the code of one component. For the tested cases, this restriction was not a problem. The test cases did not require CoCa to have functionality for global rejection of jobs, although this was a requirement for the CoCa communication model formulated in section 5.3.

Future tests

To test the multiple reader, e.g. \textit{rd} functionality of CoCa, recalibrations could be performed via CoCa. To test the general recombination feature rather than the "recombination to the same component" feature, CPREAD can be split into three parts: (1) a front part containing the translation and track recognition task, (2) parallel track fitting workers, and (3) a back part containing vertex fitting and the calorimeter calculations. To test the functionality of the C++ binding more extensively, CoCa can be tried with a HEP application written in C++, when available.
Chapter 9: Discussion and conclusions

9.1 Introduction

This chapter contains a discussion concerning CoCa in section 9.2, an evaluation of the design process in section 9.3 and conclusions in section 9.4.

9.2 Discussion of CoCa

9.2.1 Using CoCa for parallelizing HEP software

CoCa aims at running a HEP application in parallel on a parallel computer. We have shown that CoCa can be successfully applied to parallelize (existing) HEP applications. CoCa supports event level farming, a well known technique for event throughput increase in HEP, and additionally aims at event latency decrease by means of sub-event parallelism. Event latency decrease in HEP can save resources like expensive fast buffer memory.

The CoCa model is motivated by the observation that the native programming model of current parallel computers is very machine dependent. CoCa aimed at providing abstraction from the native programming model, while retaining performance. This abstraction results in hardware independence and portability of the parallel application. Portability of HEP applications is important, as their long lifetimes require the possibility of hardware updates, considering the quickly evolving hardware technology. Furthermore, hardware independent software allows HEP to postpone the choice of hardware, which is important considering the long development time of the large HEP applications.

The CoCa programming model addresses problems caused by typical features of HEP software. Consequently, much of the complexity, induced by the parallelization, could be hidden for the application programmer, which contributed to the abstraction from the parallel computer’s native programming model.

CoCa is designed as a library that can work with standard software and hardware. Nevertheless, the CoCa primitives are very fast, which results in an excellent performance of the applications parallelized using CoCa. CoCa aims at a separation of concerns. Its separation of the realization of the parallel application in a coding and configuration phase is useful, as good performance of the parallel application is combined with hardware independence of the application code. Also the location transparency provided by the CoCa primitives results in hardware independence of the parallel application.

The multiple language bindings allow use of CoCa with both existing and new HEP applications. As the test cases used relatively old HEP applications written in Fortran, we encountered many complications during the parallelization, that resulted in a decreased level of abstraction
and an increased overhead with respect to what could have been possible. It also resulted in long development times of these parallel applications, but this was also caused by the non-familiarity with CPREAD. Our newly designed test program written in C++ does not suffer from these complications, and we saw that this resulted in quickly established, concise, relatively simple application code, without many bookkeeping statements, and a low overhead of the CoCa primitives. This expressed itself in the excellent latency decrease that was obtained. For new HEP applications, the CoCa programming model should help to structure the parallel application. CoCa is intended for HEP applications, but due to its generality, we foresee that CoCa can be successfully used for applications with similar characteristics outside the HEP domain. Radar tracking is an example of such an application, as it requires track finding by means of pattern recognition and vertex calculation.

Usage of CoCa's recombination facility resulted in a reduced complexity of the application code, as less code for administration purposes was required. Furthermore, it resulted in hardware independence of the application code, as the code could be the same for shared and distributed memory machines. No performance penalty caused by the recombination facility was measured. For cases where automatic recombination is not feasible, the application programmer can obtain correct recombination by using CoCa primitives with predicates. Tests showed that this is possible, but hardware independence and consequently, portability were not fully maintained.

CoCa does not provide support for proving the correctness of the application code. It has a debug facility with which the job space can be inspected at various levels of detail. This was useful, in particular during the configuration phase. Changing the configuration does not require recompilation or relinking, which saved much development time for the large HEP applications.

CoCa design

Good performance has been a major design issue. The similarity between the CoCa job space and a parallel main-memory database, and recognition that our desired properties resemble those of database transactions, allowed us to use database techniques. The architecture of shared CoCa resembles a parallel shared everything database, whereas distributed CoCa resembles a parallel shared nothing database that uses function shipping. CoCa supports data allocation by means of data distribution strategies and supports parallel data access; both are also applied in parallel databases. Databases have a general way to express relations between data, whereas CoCa supports this in a limited way with its automatic recombination facility. Data security and durability were not the main focus, which facilitated achieving good performance.

9.2.2 Comparison of CoCa with other models for parallelization

We only consider models or systems for parallelization, not cases where parallelism has been customized to the given problem.

In the HEP context

PIAF (Parallel Interactive Analysis Facility) is a system that aims at decreasing the latency of big HEP analysis jobs. It uses the client/server master/slave model to obtain event level farming. PIAF is strongly interwoven with Zebra. PIAF can be run on multiple work stations and is also ported to the CS2 [Hakulinen95]. At start-up, the master partitions the data set into parts that are each assigned to a slave. The slaves themselves read the data to be analyzed from disk, and send the results after analysis to the master. With respect to scalability on the CS-2, PIAF initially shows a super linear speedup, as more nodes have a bigger effective cache. As the
number of nodes increases, the limiting factor becomes the inter-process communication between master and the many slaves. For a test with 25 nodes, a speedup between 8 and 19 was achieved. The reason for this bottleneck is the use of TCP/IP socket data transfer for the inter-node communication. This allows a maximum bandwidth of 4.5 MBytes/s, which is less than the 40 MBytes/s that can be achieved when ELAN Channels are used. TCP/IP is chosen for portability reasons. Event level farming on the CS-2 using CoCa showed a speedup of 26 for 28 nodes.

SAGA (Shift environment for ALEPH using GIGAswitch and ALPHAS) [Ranjar95] is another example of a system that supports event level farming. In its initial project proposal it was planned to implement sub-event parallelism after the event level farming version would have been completed. In a second progress report it was decided that sub-event parallelism would not be worth the effort to implement, as one did not believe that it was possible to improve the efficiency beyond that of event level farming. To give an impression of the granularity, the event processing time was 70 ms average. SAGA uses PVM for inter-processor communication. It is based on a master/slave model that allows dynamic load balancing. The master sends the address of an event to a slave and the slave then retrieves the event from disk. The event processing results are retrieved by one process, called the “back-end client”. Initially, a maximum speedup of 2.4 was achieved for 6 slaves on 6 nodes. With more than 6 clients the system became increasingly slower. The reason for this mediocre performance was the overhead of BOSS, a package similar to Zebra, which implied that 40% of a slave’s processor time was spent on reading and unpacking of event information. The master unpacked the first banks of every event to find the event number. This event number was subsequently sent to a slave. A slave also unpacked the first banks of every event, and checked its run number against the one sent to him by the master, to see whether it was the event that the master assigned to him. A system where the master does not send the event number, but the complete event, meaning that a slave does not read the event from disk, should avoid this overhead. Tests showed that with this improvement a nearly linear performance was established up to 6 nodes.

The GPMIMD system consists of hardware and software specially designed and built to obtain online event reconstruction for the CPLEAR experiment by means of event farming. In section 8.3.6 we compared the GPMIMD system with CoCa for the CPREAD event farming test case. Although their software was optimized on a very low abstraction level for the hardware and application under consideration, the GPMIMD system obtained an event reconstruction rate of 64 Hz, which was only useful for offline event reconstruction. CoCa on the CS-2 obtained 560 Hz, without optimizing the given test case. This means that it can cope with the event production rate of the CPLEAR experiment of maximally 450 Hz. Hence, online event reconstruction is feasible. We have to take into account that the coarse granularity of parallelization makes it relatively easy to keep the overhead low. The comparison of GPMIMD and CoCa is done from the perspective of a user (e.g. the physicists), as it compares two systems consisting of hardware and software. This means that from this comparison no statement can be made whether the result is due to the quality of the software or the hardware.

Although not specifically intended for HEP, PVM has been used in HEP. In the examples we found, PVM was applied to obtain relatively coarse grained (event level) farming. Our comparison on the CS-2 of PVM and MPI with CoCa showed that CoCa’s performance is better, despite CoCa’s richer functionality. The result of this comparison should be interpreted with care, because although CoCa is intended for use on any shared or distributed memory computer, PVM and MPI are even more general, as they are also intended for distributed computing applications. This means that their design might contain more compromises than CoCa’s design, what might explain a part of the observed performance difference.
A model in the HEP context that tackles sub-event parallelism is Mona Lisa. In section 8.4.3 we compared Mona Lisa with CoCa for the CPREAD parallel track fitting test case. Although it is specially designed for parallelization according to the master/slave pattern, Mona Lisa obtained a speedup of only 1.5 where 3.5 was theoretically possible. CoCa achieved a speedup of 2.5, without optimizing the given test case. In tests with a test program that has the same time characteristics as CPREAD parallel track fitting, but that does not suffer from the overhead increasing complications caused by the old fashioned structure of CPREAD, CoCa established a speedup of 3.52 where 3.63 was possible (see section 8.5).

Outside the HEP context

Linda resembles CoCa in that it also uses a communication model based on a job space. Linda has a large expressive power, but its generality translates into low performance [Molinari94]. CoCa has more restrictions and the resulting knowledge can be used to improve the performance. We chose the restrictions such that they do not limit CoCa’s flexibility, needed for the target applications under consideration, namely HEP applications. Defining the job types in CoCa is done at compile time for the C and C++ binding and at initialization time for the Fortran binding. This type information allows CoCa to confine the search space for appropriate data items, by logically and physically structuring the job space according to a job’s type. It also allows data distribution decisions based on the data type, which results in an improved locality of data. Furthermore, the knowledge about the job size allows cheaper memory management. Linda has a dynamic typing system which has as a consequence that to search for a tuple, the whole tuple space is candidate. Furthermore, dynamic typing in Linda causes runtime overhead, since data items of tuples have to be evaluated at runtime to find out the tuple’s type. CoCa relates job allocation and component allocation, which facilitates the configuration phase and is less error prone. Linda has no support for component allocation. CoCa is aware of relations between jobs and the CoCa recombination feature facilitates the application programmer’s work. Linda does not support this, which makes it the responsibility of the application programmer.

Some implementations of Linda use pre-compilers that attempt to guess the communicated data types to improve data locality. The use of pre-compilers, however, reduces a paradigm’s flexibility and complicates supporting many host languages. Although the CoCa primitives are implemented as a library, which means that no pre-compiler is needed, good performance could be retained.

9.2.3 Parallel Hardware

In section 2.2.1, we described models of a shared and a distributed memory computer that we used as working model to design CoCa. Although the models were constructed using the SC2000 and the CS-2 as examples, they were intended to be general models of shared and distributed memory computers, as our aim was to develop portable software. In section 2.2.2.1, we also mentioned some opinions about programming shared and distributed memory computers. We can now evaluate our experiences.

The apparent speed advantage in memory access bandwidth and latency of shared memory computers with respect to distributed memory computers was lost for a large part, because of the overhead introduced by the use of synchronization primitives. On the distributed memory computer, also bookkeeping turned out to produce an important overhead. The constant part of the (inter-node) communication latency consists of (1) a start-up latency caused by the communication hardware, (2) synchronization overhead, and (3) administration overhead. So, although literature often focuses on (1), we found that (2) and (3) are relatively large.
A reduction of the constant part of the (inter-node) communication latency would be a major improvement. This holds for both types of computers. It would allow more fine-grained parallelism. The (inter-node) communication bandwidth is not a problem, for the parallel applications we investigated. Günzinger's idea [Günzinger96] to provide hardware support for the administration part is promising, with the remark that it could be considered as special purpose, and consequently suffers from the inherent disadvantages of special purpose hardware. Especially models like CoCa and distributed shared memory models could benefit from an improvement as described above, as it would make them more competitive with respect to customized implementations that take the distributed nature into account.

During the development of CoCa on the CS-2, we experienced that the state of the (debugging) tools is not yet fully matured, which resulted in long application development times. This is another reason for the decreased popularity of distributed memory computers.

## 9.3 Evaluation of the design process

### Ph.D. result versus the project proposal

If we compare the Ph.D. result with the original project proposal [Argante94], most of the planned goals have been accomplished. Some goals were not accomplished due to external factors. Oracle related tasks received limited attention because of Oracle's unavailability on the CS-2, although availability was foreseen at the time of the project proposal. Therefore, the focus on the offline data diminished. The foreseen event level farming of CPREAD has been replaced by Zebra block level farming, as this was suggested by the Zebra structure of CPREAD. Calibration constants have not been managed via CoCa, although this could have been an interesting exercise. The Virtual Shared Memory (VSM) option was not available on the CS-2, although availability was foreseen at the time of the project proposal. This has been overcome by implementing part of this functionality in CoCa. The Ph.D. thesis, which was not planned originally, replaced the CoCa user manual. A detailed technical description of CoCa [Argante97/2] and a code level description of the adaptation of the CPREAD application [Argante97/3] have been made.

Although most of the planned features have been covered, the final product is different from that sketched in the original proposal. The motivation remained unchanged, but the emphasis changed to other areas with respect to the original proposal. We focused on a general model for HEP software parallelization, instead of a system for event reconstruction, although event reconstruction was used for the test cases. Many tasks turned out to be more time-consuming than planned. For example, designing and implementing the core of CoCa on the CS-2 took 8 months, whereas only 2 months were planned. This was caused by the preliminary state of development tools on the CS-2, which made the debugging of the CoCa implementation tedious and time consuming, and the unavailability of the VSM option on the CS-2. Writing the Ph.D. thesis took 18 months, whereas originally 2 months were planned for the user manual. Performance measurements turned out to be less straightforward and more time consuming than expected, because of the application dependent behavior of the OS facilities. On the other hand, some foreseen tasks were less time consuming than expected. For example, we expected to need database-like concurrency control algorithms, as we foresaw complicated concurrent data access patterns. However, investigation of the HEP software showed that relatively simple concurrency control algorithms that incur little overhead were sufficient. This simplicity was also suggested by the high overhead of the Solaris synchronization primitives. These changes of emphasis required a readjustment of the original planning. We performed four readjustments. Besides the readjusted plannings, about 20 progress meetings with progress reports as a result were another means to measure the progress of the project.
Software engineering

The following paragraph has been taken in an adapted form from [Williams96]. In the waterfall software engineering approach [Yourdon90] tasks are laid out and executed sequentially. There is no software product until the very end. Intermediate products are for example analysis or design diagrams. This seems to be a reasonable manufacturing approach and one that manages complexity, yet many software projects fail when done this way. Criticisms against the waterfall approach are its lack of flexibility, requirements are frozen too early, the development is slow and cannot adapt to a changing environment, and the lack of feedback makes that changes required in the system are recognized too late. Evolution is a useful analogy for discussing software development: it is rare that software is simply manufactured and used as is forever. Just as a living organism adapts to its changing environment, software must adapt to ever-changing requirements. An evolutionary software engineering approach, also called spiral model [Boehm88], can accommodate changes as they occur. It is an iterative approach of which the basic goal is to deliver useful systems in shorter cycles, even with fewer clearly defined requirements. Developers may do some analysis, some design, some prototype coding and then return to the analysis. Also analysis design loops are possible. A danger is that too much coding effort is done too early.

At the beginning of the project, the project goal was not very strictly defined and the requirements were of a very general nature. Therefore, much time was spent on stating the detailed requirements. The original attempt to follow the waterfall method slowed down the project, as five months were needed to define the requirements. Only after a relatively long time, when an implementation was ready, feedback was available and the requirements could be adapted. A problem of the HEP community is that new applications and systems are only adopted (for evaluation) if they have proven to be useful. In my case this resulted in the absence of users of CoCa, which slowed down the project, because of the lack of feedback. For research-like projects with not strictly defined end product and requirements, early prototyping is needed to have feedback by means of user tests, performance tests and comparisons with existing models. This means that the evolutionary method is more appropriate. As the project proceeded, I changed from the waterfall method to a method with more prototyping which resulted in better progress.

In chapter 6 we met an example where the iterative character of the evolutionary approach turned out to be more appropriate than the sequential character of the waterfall approach. Since we did not know in advance which features provided by the hardware and OS we would use, we did not know in advance of which features of the hardware we should test the performance. On the other hand, without the information about the hardware performance, we could not make proper design decisions.

Use of theory and standard solutions

For the CoCa design, we tried to apply standard solutions as much as possible. However, the assumptions made in formal derivations of algorithms cannot always be made to hold in practice. Also the relative overhead of some functionality on which the algorithms are based, can turn out to be very different in practice, which has as a result that the trade-offs made in the algorithms lose their relevance. An additional problem is that this trade-off is very machine dependent. This is one of the reasons why portable software with a good performance is difficult to realize. We mention some examples where this gap between theory and practice was encountered. The large overhead of the synchronization primitives offered by the Solaris OS had as a result that algorithms, that allow much concurrency by many small locking intervals, could not be used. To circumvent this large overhead, we attempted to obtain cheaper synchro-
nization by looking at algorithms designed by Lamport, but the algorithm's assumption of atomic read and write operations could not be ensured, because of the SC2000 memory model.

**Documenting a design**

The CoCa system has been built with facilities offered by the hardware and the OS, of which the properties were not exactly known in advance. This resulted in a design process with iterations and side steps that were dead ends. Parnas [Parnas86] argues that a software design should be documented in a rational way, without "stream of discovery"-like chronological, iterative descriptions, or "stream of execution"-like descriptions. This thesis describes the CoCa model in this way. An example is chapter 6, that is presented in a rational order (beforehand we limited the set of tested operating system primitives, without knowing how the design would develop), although the chronological design process was iterative. Another example is that early in this thesis the requirements are mentioned, whereas some requirements became clear late in the project. The section about the Lamport exclusion algorithm is an example of the description of a dead end.

**Portable software**

Portability of the parallel application was one of CoCa's aims. CoCa's implementation should be able to exploit the resources of the different hardware platforms and should be able to work with different host languages (we also call this portability). The best way to implement some given functionality depends much on the hardware characteristics. A slight change in the relative performance of features provided by the hardware might favor a different design. Therefore, implementing portable software compromises the performance, since the relative performance of the provided features is different for different hardware platforms, which means that one design cannot be ideal for all hardware platforms. A second reason for compromised performance is that portable software can only use features that are supported by all target hardware platforms. In the CoCa design, we applied four solutions to this portability problem: (1) we defined some functionality we expected to be provided by the hardware, (2) we introduced a physical communication layer that is thick or thin, dependent on the functionality provided by the given hardware platform, (3) for different hardware architectures, rather than different hardware platforms, a layered design was chosen with some layers specific to the given target hardware architecture (example: shared and distributed CoCa), and (4) CoCa itself cannot infer all required hardware specific information, so we introduced a configuration phase that allows the application programmer to provide configuration and hardware specific information to CoCa.

The requirement that CoCa should be able to work with different host languages, had a similar restrictive effect as the requirement that CoCa should be portable to different hardware platforms. It imposed restrictions on the functionality that could be used to realize CoCa. We decided to implement CoCa in C as this programming language is compatible with Fortran and C++. Although there is a C++ binding, we could not use the full expressive power of C++. For example, we did not use typed job queues, which resulted in explicit casting and less type checking. This had also as a result that the C++ host language binding had to reestablish the original functionality of the host language, as one of CoCa's aims is to follow the general philosophy of the host language. For example, this required runtime type information.

Summarizing, portability comes at a cost: it has a restrictive effect on the design and implementation tools and the hardware facilities that can be used for the realization of the artifact, which can lead to a reduced functionality and a performance penalty. Whether portability is worth this cost depends on the application domain. For HEP, it is worth it, due to the complex HEP applications with long life and development times. The wide introduction of networks and the inter-
net makes portability more important, which can be seen from the popularity of a product like Java.

9.4 Conclusion

CoCa consists of a small set of simple primitives that allow parallelization of software in a relatively easy way. It aims at abstracting from the native programming model of parallel platforms, while retaining performance. This abstraction improves the portability of parallel applications both for shared and distributed memory computers, and reduces the complexity for the application programmer. The abstraction could be established by limiting CoCa’s application domain to HEP software applications. For the test cases, we showed that these aims were indeed accomplished and that the performance was excellent. With CoCa we showed that in the HEP context, parallelism with as goal a latency decrease is possible and also feasible and does not require parallelism customized to the given problem.
Summary

This thesis describes the development of CoCa. CoCa has been developed during a designer’s Ph.D. project carried out at CERN, the European Laboratory for Particle Physics, in collaboration with the Eindhoven University of Technology. CoCa, which stands for Communication Capability, aims at running high energy physics applications in parallel on parallel computers. This offers a solution to meet the very demanding computing requirements of high energy physics applications. The CoCa model is motivated by the observation that the native programming model of current parallel computers is very machine dependent. CoCa aims at abstracting from the native programming model, while retaining performance. This abstraction results in hardware independence and portability of the parallel application. Portability is very important for high energy physics applications, because their long lifetimes require the possibility of hardware updates, considering the quickly evolving hardware technology. CoCa takes the problems into account that are caused by typical characteristics of high energy physics applications. Consequently, much of the complexity, induced by the parallelization, could be hidden for the application programmer, which contributed to the abstraction from the parallel computer’s native programming model. CoCa is intended for use on shared and distributed memory computers. It has been implemented on a SUN SPARCcenter 2000 shared memory computer and a Meiko CS-2 distributed memory computer. CoCa has been successfully applied for the parallelization of the CPREAD high energy physics application.

CoCa has a programming model in which multiple components run in parallel. A component performs a task and consists of several lines of code. CoCa has a communication model based on communication via a job space combined with single writer/multiple reader access to the communicated data. All not yet processed data residing in job space can be accessed by any component. A component selects appropriate data from job space by specifying predicates. Job space communication allows hiding the structure of the underlying hardware platform. By separating the realization of a parallel application into a coding and a configuration phase, CoCa can nevertheless take advantage of specific features of the underlying hardware, thereby retaining the performance. In the coding phase, the programmer of the parallel application inserts CoCa primitive statements in the application code that take care of inter-component communication. In the configuration phase, the component/processor mapping and communication patterns are specified, which allows one to improve data locality and therefore reduce communication overhead. CoCa provides support to correctly recombine related data that need to be reassembled after they have been distributed among the components.
CoCa: een model voor parallelisatie van hoge energie fysica software

Dit proefschrift beschrijft de ontwikkeling van CoCa. CoCa is ontwikkeld gedurende een promotie op proefontwerp, uitgevoerd op het CERN, het Europese Laboratorium voor Deeltjesfysica, in samenwerking met de Technische Universiteit Eindhoven. CoCa, wat staat voor Communicatie Capabilliteit, richt zich op het parallel executeren van hoge energie fysica programma's op parallele computers. Dit biedt een oplossing voor de grote behoefte aan rekenkracht van hoge energie fysica programma's. Een aanleiding voor het CoCa model vormt de observatie dat het programmeermodel van de huidige parallele computers erg machineafhankelijk is. CoCa streeft ernaar te abstraheren van het programmeermodel van parallele computers met behoud van prestatie. Dit resulteert in hardwareafhankelijkheid en portabiliteit van de parallele applicatie. Portabiliteit is erg belangrijk voor hoge energie fysica applicaties, omdat hun lange levensduur de mogelijkheid vereist om de hardware op te waarderen, de snel evoluerende hardweotechnologie in aanmerking genomen. CoCa geeft een oplossing voor problemen die worden veroorzaakt door typische eigenschappen van hoge energie fysica programma's. Hierdoor kan een groot deel van de complicaties, teweer gebracht door de parallelisatie, worden verborgen voor de applicatieprogrammeur, wat bijdraagt tot het abstraheren van het programmeermodel van parallele computers. CoCa is bedoeld voor gebruik op computers met gemeenschappelijk en gedistribueerd geheugen. Het is geïmplementeerd op een SUN SPARCcenter 2000 computer met gemeenschappelijk geheugen en een Meiko CS-2 computer met gedistribueerd geheugen. CoCa is met succes toegepast voor het parallelliseren van het CPREAD hoge energie fysica programma.

CoCa heeft een programmeermodel waarin meerdere componenten gelijktijdig executeren. Een component voert een taak uit en bestaat uit meerdere regels code. CoCa heeft een communicatiemodel gebaseerd op communicatie via een job space gecombineerd met een model waarbij één schrijvende component of meerdere lezende componenten gelijktijdig toegang tot een data item hebben. Elke component heeft toegang tot alle nog niet verwerkte data die in de job space resideren. Een component selecteert geschikte data uit de job space door predicaten te specificeren. Job space communicatie maakt het mogelijk om de structuur van het onderliggende hardwareplatform te verbergen. Door het scheiden van de ontwikkeling van de parallele applicatie in een codeerfase en een configuratiefase, is CoCa desondanks in staat om de specifieke eigenschappen van het onderliggende hardwareplatform uit te buiten, wat tot het behoud van goede prestaties leidt. In de codeerfase neemt de programmeur van de parallele applicatie aanroepen van CoCa primitieven op in de applicatiecode, die de inter-component communicatie voor hun rekening nemen. In de configuratiefase worden de plaatsing van de componenten op de processors en communicatiespatronen gespecificeerd, wat het mogelijk maakt dat elke component independently van de andere componenten werkt. CoCa heeft functionaliteit om op correcte wijze gerelateerde data te recombineeren en onnodige communicatie te vermijden. CoCa heeft functionaliteit om op correcte wijze gerelateerde data te recombineeren en onnodige communicatie te vermijden. CoCa heeft functionaliteit om op correcte wijze gerelateerde data te recombineeren en onnodige communicatie te vermijden.
Acknowledgments

I would like to thank my daily supervisors Peter van der Stok and Ian Willers. Peter van der Stok for his inexhaustible patience and enthusiasm, always ready to read a chapter of this thesis and to return it with comments. I still remember the sessions on Sundays with a view on the sun-drenched vineyards. Ian Willers for his continuing support and the many coffee breaks we had together. Thanks to Marcel Meesters who spent nine months on this project. During this time, besides doing useful work, we invented the word “bruizelen”, which turned out to be very often applicable, but whose meaning is still not fully clear to me.

I am grateful to my Ph.D. supervisor Martin Rem for his contribution. I would like to thank the people of the GPMIMD project who provided a perfect ambiance during the first two years of my Ph.D. Thanks to the people of CMS-CMC, responsible for the CMS software, and its leader Martti Pimia for my pleasant stay in this group during the last year of my Ph.D. I am thankful to Ad Aerts for comments on chapter 3 and Dave Francis for comments on chapter 4. Thanks to René Schiefer for the code splitting of CPREAD and William Lu for adapting the CPREAD code for usage on SUN Solaris platforms. Thanks to EU ESPRIT project P7255, GPMIMD-2, for the access to their Meiko CS-2 machine and the Linköping University in Sweden for the access to their SUN SPARCcenter 2000 machine.

I would like to thank my friends in Geneva with whom I spent a splendid three years, in the pub, in the mountains and in the air. Thanks to my friends in the Netherlands that always received me with a warm welcome when I returned from Geneva for a visit. I am thankful to Geneviève for her infinite patience and confidence that this Ph.D. would eventually finish. Thanks to my parents and family for their confidence and unremitting stimulation.

This work is jointly funded by the Ph.D. program at CERN and the designers Ph.D. program of the Stan Ackermans Institute for continuing education (SAI) at the Eindhoven University of Technology, the Netherlands.
# Curriculum vitae

<table>
<thead>
<tr>
<th>Erco Argante</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 januari 1968</td>
</tr>
</tbody>
</table>
| 1980 - 1986 | VWO  
Elshof College in Nijmegen |
| 1986 - 1992 | Natuurkunde  
Katholieke Universiteit Nijmegen |
| 1992 - 1994 | Ontwerpopleiding Technische Informatica  
Technische Universiteit Eindhoven |
| 1994 - 1998 | Promotie op Proefontwerp  
Technische Universiteit Eindhoven  
uitgevoerd bij het CERN in Genève |
| 1998 - | Werkzaam bij Ericsson Telecommunicatie in Rijen |
References

[AIHENP96] AIHENP’96, proceedings of the AIHENP’96 conference, Lausanne, Switzerland, September 1996.


[Huang91] J. Huang, J.A. Stankovic, K. Ramamirtham, D. Towsley, Proceedings of


[Paxson93] V. Paxson, Flexdoc - documentation for Flex, fast lexical analyzer generator, version 2.4, Lawrence Berkely Laboratory, Berkeley, California, USA, November 1993.


[SUN96] SUN Microsystems Inc., A scalable server architecture, from department to enterprise, the SPARCserver 1000E and SPARC-center 2000E, Mountain View, California, USA, February 1996.


Index

A
ACID constraints 29
action 29
algorithmic parallelism 13
algorithmic parallelism pipelining 13
API 73
application programmer's interface 73
asynchronous communication 23
atomicity 29, 30
automatic recombination 95
automatic variable 55

B
B-tree index 30
bucket 75
busy waiting 58

C
calibration constants 36, 39
calorimeter 35
cardinality 27
catalog data 76
channel communication 71
CI 102
class 29
client/server 23, 84
cluster 11
CMS 35
CoCa communication model 48
CoCa primitive 48
commit 30
communication 11
component 10
component exclusion 58
component level parallelization 10
component local memory 56
compoment shared memory 56
component waiting 58
computation/communication overlap 8
concurrency atomicity 30
concurrency control 30
concurrent 55
concurrent solutions 18
condition variable 56
conditional communication 18
configuration information 46, 102
configuration phase 46, 102
conflict serializable 32
conflicting action 32
consumer 11
control data 76
control/interactive data flow 37
correct recombination 26
correctness 29
critical section 56

data abstraction 29
data dependent 11
data distribution 31
data distribution strategy 54, 80
data flow graph 96
data parallelism 12
data parallelism pipelining 13
data shipping 32
database 27, 29
database management system 27
deadlock 92
deferred functions 29
detector 35
distributed CoCa 73
distributed memory computer 7
distributed memory system 24
distribution strategy 31
DMA transfer 71
dynamic binding 29
dynamic bucket management 75
dynamic communication 25
dynamic load balancing 15, 18, 26

efficiency 15
encapsulation 29
entity 27
entity-relationship model 27
environment 11
event 35
event reconstruction 36
evolutionary approach 136
explicit memory access 7
explicit parallelism 10
fail atomicity 29
fanout 96
farming 12
function shipping 32
genericity 29
geometric data parallelism 13
global stop option 19, 26
global variable 55
granularity of parallelism 22
group 11

H
hashing 81
HEP 35
high energy physics 35
hybrid distributed memory computer 8

I
I/O load 8
idle component 14
idle job 14
idle processor 14
imbalance 14
implicit communication 6
implicit memory access 6
implicit parallelization 10
index 30
inheretance 29
in-memory database 30
inter-job parallelism 12
interleaved 55
inter-query parallelism 30
intra-job parallelism 12
intra-query parallelism 30
isolation 30

J
job 11
job access 48, 57
job selection 48, 57
job space 47, 48
job state diagram 49
job states 48

L
language binding 73
latency 5
LHC 35
L\text{isol} 15
L\text{job} 13
load 15
load balancing 15
L\text{theor.min} 15
L\text{max} 15

M
main-memory database 30
maximum throughput 14
message passing 22
messages 55
meta data 76
migration 24, 31
module 10
Stellingen
behorende bij het proefontwerp

CoCa: a model for parallelization of high energy physics software

door

Erco Argante

7 oktober 1998

1. Het door middel van parallelisatie reduceren van de beslistijd van de level 2 trigger van het CMS experiment kan bij gelijktijdige rekenkapaciteit tot een beperking van het benodigde buffergeheugen leiden, wat een aanzienlijke kostenbesparing tot gevolg heeft [dit proefontwerp, hoofdstuk 4].

2. Het ontwerp van CoCa is zodanig gekozen dat hardware-onafhankelijkheid van parallelle hoge energie fysica software bewerkstelligd kan worden, zonder dat het CoCa model zijn algemene toepasbaarheid binnen de hoge energie fysica omgeving verliest. [dit proefontwerp, hoofdstuk 5].

3. Een belangrijk aspect voor het succes van een shared memory computer is een goede performance van de synchronisatiefaciliteiten geleverd door het operating system [dit proefontwerp, hoofdstuk 6].

4. De kwaliteit van produkten gemaakt door een multidisciplinair team wordt onder meer gewaarborgd doordat voor een gegeven deel van het produkt de gespecialiseerde discipline de inhoudelijke kwaliteit bewaakt, terwijl de andere disciplines resultaatgerichtheid nastreven.

5. Een belangrijke beslissing tijdens het maken van een ontwerp bestaat uit het bepalen op welk moment er tijd wordt besteed en hoeveel tijd er wordt besteed aan het zoeken naar gerelateerde informatie, om te vermijden het wiel opnieuw uit te vinden.

6. Veel tijd kan worden bespaard door op elk niveau in een project aan te geven voor welke van de requirements future proofness en flexibiliteit belangrijk zijn en voor welke requirements niet.

7. Parnas1 pleit voor een rationele opzet van de ontwerpdocumentatie; Unified Modeling Language, een op dit moment veel gebruikte methode om op object-georiënteerde wijze te modelleren, voldoet hier niet aan, omdat een groot deel van het ontwerp op chronologische wijze wordt gedocumenteerd.

8. De informatica is een goede proeftuin om verschillende ontwerpprocessen te testen.

9. Het is misleidend om bij een weersverwachting dagtemperatuur met jaargemiddelden te vergelijken zonder daarbij de distributie te vermelden.

10. Alhoewel per 1 januari 1998 geen werkvergunning meer nodig is om als inwoner van de EU te werken in een willekeurig land van de EU, zullen cultuur- en taalverschillen een vrije uitwisseling nog voor lange tijd in de weg staan.

---