MASTER'S THESIS

Implementation of abstract UNITY algorithms in PVM.

by

Vincent Nikkelen

Supervisors:  Drs. R.H. Mak
              Zoltán Horváth

October 20, 2000
3.4.2 Speedup ........................................... 27
3.4.3 Upper bound for the speedup ................. 28
3.4.4 Scaling of the parallel algorithm ............ 29
3.5 Remarks .............................................. 29
  3.5.1 When not to use this algorithm ............... 29
  3.5.2 A more compact solution ..................... 31
  3.5.3 Topics not treated ............................. 31

4 Parallel Elementwise Processing ................. 32
  4.1 Techniques, definitions and notations ........... 32
    4.1.1 Sequence .................................. 32
    4.1.2 Total disjoint decomposition ................. 33
    4.1.3 Elementwise processability .................. 34
       1-1 elementwise processing .................... 35
       2-1 elementwise processing .................... 36
       1-2 elementwise processing .................... 37
       k-1 elementwise processing .................... 38
  4.2 The algorithm in an informal way ............... 38
  4.3 Transformation of the algorithm ............... 40
    4.3.1 Topology .................................. 40
    4.3.2 Datatypes .................................. 41
    4.3.3 General considerations ..................... 46
    4.3.4 Total disjoint decomposition: Initializing part 47
    4.3.5 Total disjoint decomposition: Generating blocks 48
    4.3.6 Elementwise processing .................... 51
    4.3.7 Collecting the partial results ............... 55
    4.3.8 Absence of deadlock ........................ 57
  4.4 Timing ............................................. 57
    4.4.1 The input .................................. 58
    4.4.2 Elementwise processing slices ............... 59
    4.4.3 Constructing a cdd .......................... 60
    4.4.4 Finding an optimal B ....................... 61
4.4.5  Idle time ............................................. 63
4.4.6  Reducing the idle time .............................. 64
4.4.7  Model .................................................. 66
A simple model without communication ................... 67
Adding unbalanced load to the model ..................... 67
Adding communication to our model ....................... 68
4.5  Remarks ................................................. 70
4.5.1  Optimization of the algorithm ..................... 70

5  Conclusion ................................................. 71

A  Proofs .................................................... 72
A.1  Proof of: (8) $\Rightarrow \neg(6) \land \neg(7)$ .............. 72
A.2  Proof of: (13) .......................................... 72
A.3  Proof of: Elementwise processability of the union of sets ........................................................................... 74
A.4  Proof of STEP ............................................ 76

B  Corrections for [Fóth 01] ..................................... 77

C  Programs ..................................................... 78
C.1  Compiling and linking ..................................... 78
C.2  Parallel asynchronous computation of an associative function .............................................................. 79
C.2.1  Root ...................................................... 79
C.2.2  Node ...................................................... 79
C.2.3  A more compact solution ................................ 81
C.3  Elementwise Processing ................................... 82
C.3.1  Declarations ............................................ 82
C.3.2  Datatypes ............................................... 83
C.3.3  Master ................................................... 90
C.3.4  Slave .................................................... 93
C.3.5  Optimization ........................................... 98

References ......................................................... 100
1 Introduction

This thesis is written by order of the Eötvös Loránd University (ELTE) in Budapest, Hungary. It is part of the final examination for the study technical computing science at the Eindhoven technical university in Eindhoven, the Netherlands.

1.1 Objective

Transforming an abstract sequential program into a concrete one is usually nothing more than mapping a statement to one ore more statements of the concrete language. Suppose \( f \) is a function that transforms GCL statements [Dijk 01] into one or more C statements. One can define \( f('i := 3') = 'i = 3; \) and \( f('i := i + r') = 'i* = a; \) which would be a satisfiable mapping for those 2 points.

In an abstract parallel program however, these mapping functions are not that obvious. Take for example the UNITY statement \( x.i := x.i + y.j \). Transforming this into the PVM statement \( x[i]++ = y[j] \) could be correct, but only in the case where the performing task is able to write and read in the memory where variable \( x[i] \) is stored. For variable \( y[i] \) and \( i \), only reading is required. Unfortunately, this is not the case when a variable is stored in a memory that is connected to an other processor than the task is running on.

When transforming abstract parallel programs into concrete ones, a lot of factors have to be taken into account. How many processors are available, what type of topology does the network offer and can tasks share memory, are some fixed factors. The grain size of the computation, the size of communications and the location and number of the started tasks (mapping) has to be chosen during the transformation. Which choice one has to make depends on the performance one wishes to achieve and the used network.

Main goal of this thesis is to get more insight into the matter of transforming abstract parallel programs efficiently into concrete ones for a certain environment. The abstract algorithms in this thesis will be written in UNITY and implemented in PVM. As execution platform we will use the PowerXplorer.

1.2 The PowerXplorer

The PowerXplorer is a parallel computer that consist of one or more clusters each hosting 4 processing units. Those clusters are connected to each other and every cluster is also connected by a special dedicated interface to an UNIX system, also called the host. A schematic representation of the PowerXplorer and its host can be found in the left half of figure 2. At ELTE the PowerXplorer, called Shiva, consists of 4 clusters providing in total 16 processing units. More information including a photo can be found at [Shiv 01].

Via Augusta, the host at ELTE, one can control the PowerXplorer by means of allocating clusters for use, loading programs on processing units, start and stop those programs and get information about the system possibly for debugging purposes. Access to a disk-drive and the editing and compiling of a program happens on the host system as can be seen in figure 1 which shows a possible development cycle.

Note that we used the term processing units instead of processors in the first paragraphs of this section. Main reason for this distinction is that an processing unit consist of 2 processors, a Motorola 601-e PowerPC processor and a T400 transputer. For simplicity reasons we will use the term processor throughout this thesis, except in this section, to denote an processing unit. A
schematic representation of such an processing unit can be found in the right half of figure 2.

The 601-e processor is the processor that has to perform the actual execution of one or more loaded programs. Every 601-e is connected with 8MB of memory that can not be accessed by any of the other processors. In the PowerXplorer there is no other memory available that can be used by the 601-e processor.

Within the same processing unit one can find a T400 transputer with 1 MB of memory. This transputer is taking care of all the inter-processor and inter-cluster communications that need to be performed. It is connected by 4 external communication channels: 3 for communicating with the other 3 processing units on the same cluster and 1 for communicating with other connected clusters. On the PowerXplorer every processing unit can communicate with every other unit within the system. Note that the PowerXplorer does not use the host to perform inter-cluster communications.
1.3 PVM

The version of PVM used in this thesis is 3.3, also indicated as PVM3. PVM stands for Parallel Virtual Machine and is, besides a daemon (pvmd) process, a library that contains routines for initiating processes on other machines for communicating between tasks and changing the configuration of machines. Those routines can be used in both C and Fortran77. More detailed information about PVM can be found in [PVM 01]. Here we will try to make the reader familiar with the most important aspects of PVM.

1.3.1 User interface

In this section, a selection is made from the routines that PVM offers and which are used most frequently in the presented programs. A small example, consisting of the programs root.c and node.c, will be used as illustration. Root.c does nothing more than 'starting node.c' and waiting for a message that will be send by node.c.

Every program should start with including the header-file pvm3.h. This header-file defines values like PvmTaskDefault and PvmDataRaw. The command pvm_mytid(); should always be the first pvm command that is called. It returns the task-id and enrolls the calling task into PVM.

```
#include "pvm3.h"
#define MESSAGE_TAG 123

main()
{
    int myTid, otherTid, info, dummy;
    /* enroll in pvm */
    myTid = pvm_mytid();
    /* Start the other task */
    info = pvm_spawn("node", (char**)0, PvmTaskDefault, ",", 1, &otherTid);
    if ( info != 1 ) printf("Could not start all the tasks.\n");
    else
    {
        pvm_recv( otherTid, MESSAGE_TAG);
        pvm_upkint( &dummy, 1, 1);
    }
}
```

Function pvm_spawn(); starts other tasks into PVM and returns an integer, which states how many tasks have been successfully started. The first argument is the name of the task that has to be started, the fifth specifies the number of instances of the task to start and the last contains the task-id's of the started tasks after returning. In our example just one copy of the task 'node' should be started. It's task-id will be stored in the variable otherTid.

After starting the other task, task 'root' waits for a message with pvm_recv();. The first argument indicates from which task the root expects a message, the second one is a message tag. Only a message with a matching tag will be accepted. A -1 in one of the arguments indicates a wild card for that argument. Messages with a non-matching tag will be stored by the pvmd until a receive with a matching tag is executed.
Pvm_upkint(); unpacks a message into an integer specified by its address in the first argument. The second argument specifies its size (a number > 1 indicates an array) and the last argument is not important at the moment. Packages should be unpacked the same way as they were packed by the sender! Besides pvm_upkint(); there exist more functions with the name pvm_upk*(); where the * indicates the type of variable to be unpacked.

node.c

#include "pvm3.h"
#define MESSAGE_TAG 123

main()
{
    int myTid;
    myTid = pvm_mytid();
    pvm_initsend( PvmDataRaw);
    pvm_pkint( &myTid, 1, 1);
    pvm_send( pvm_parent(), MESSAGE_TAG);
    pvm_exit();
}

Routine pvm_exit(); tells the local pvmd that this task is leaving PVM. Note that leaving PVM does not mean stopping the execution of the task. A task that left PVM 'un-subscribes' itself from the pvmd and can therefore not communicate anymore with other tasks. Such a task can still perform other operations like e.g. freeing used memory.

In 'node.c', pvm_parent(); returns the task-id of the parent of the calling task. The other not yet mentioned functions are for sending messages. Routine pvm_initsend(); creates a send buffer for a new message, pvm_pkint(); packs the message and pvm_send(); sends the package to a task indicated by it's task-id. Note that the last argument of pvm_send(); can be a wild card (-1), but that the first should always be a valid task-id.

1.3.2 Asynchronous communication with buffers

The pvm_send(); routine is asynchronous. Computation of the sending task resumes as soon as the message is save on its way to the receiving task. The PVM model guarantees that if task 1 sends message A to task 2, and thereafter task 1 sends message B to task 2, message A will arrive at task 2 before message B. If both messages arrive before task 2 does a receive, then a receive with a wild card as message tag will return message A.

This means that one has to be careful with the amount of messages that one is sending. Suppose, like in the previous example, task 1 sends 2 messages in an arbitrary time slot to task 2 and task 2 is just capable of receiving 1 message in the same time slot. In this situation the buffer between both tasks will eventually be full or has to be of infinitive size. Note that designing 2 communicating tasks with a matching communication behavior is not enough to prevent this 'buffer overflow'. One has to add a kind of synchronization to the tasks or be able to proof there is enough space to contain the messages.

PVM provides us with a secure message passing system. It guarantees the following about messages:
• The system is not loosing messages.
• The system is not producing messages by itself.
• A message, once send, will always arrive at it’s destination within finite time.
• The contents of a message will not be changed by the system.

1.3.3 PowerPVM

PowerPVM is a port, an adaption of an existing program for a new environment, of PVM version 3.2.6\(^1\) in order to get a better performance. One of the largest advantages of PowerPVM compared to PVM is the improved communication latency and throughput.

The developers of PowerPVM designed and implemented it on top of an asynchronous communication kernel. The communication kernel itself is implemented on top of PARIX\(^2\), the operating system for the PowerXplorer. The advantage of this layered design, which can be seen in figure 3, is that it fills the functionality gap between PVM and PARIX. One can see the communication layer as a part of PowerPVM but also as a separate layer. More information about PARIX and PowerPVM can be found in [Pars 01].

![Figure 3: Layers of the PowerXplorer.](image)

**Differences in functionality**

In figure 4 the PVM functions are listed that are different in PowerPVM compared to PVM. Functions labeled with an asterisk '*' are not implemented in PowerPVM, mostly because they are not needed or can not be implemented on the PowerXplorer. E.g. the function `pvm_addhost()` is such a function that can not be implemented because it is not possible to dynamically add or remove clusters from a configuration. Not implemented functions return the parameter `PvmNotImpl`.

Functions labeled with a sharp symbol '#' are implemented for compatibility reasons. PVM daemon task id's are returned by those functions but are not valid for sending or receiving messages.

It is not possible to choose a processor where to start a task on in PVM because the PowerXplorer is presented as a homogenous system. Therefore the `pvm_spawn()` function in PowerPVM behaves different from the standard implementation. Instead of being able to choose an architecture to

\(^{1}\)There is no functional difference between PVM version 3.2 and 3.3 that is used in this thesis.

\(^{2}\)PARallel uniX
\begin{table}
\begin{tabular}{|c|c|c|}
\hline
pvm_addhosts() & pvm_notify() & pvm_start_pvmd() \\
\hline
pvm_config() & pvm_recv() & pvm_tasks() \\
\hline
pvm_delhosts() & pvm_sendsig() & pvm_tickle() \\
\hline
pvm_kill() & pvm_spawn() & pvm_tidtohost() \\
\hline
\end{tabular}
\caption{Differences between PVM and PowerPVM.}
\end{table}

start a task on, the user can specify a processor where he wants to start a task. When no specified processor is entered, tasks will be started on the available processors in a round-robin way.

\textbf{Implementation notes}

In PowerPVM a parallel machine with \( n \) processors is seen as a network with \( n \) hosts, all of the same architecture. In order to keep compatible with PVM every processor will be named \texttt{PvmNodeXX} where \( XX \) denotes the processor (node) number. The processors on the cluster in use with the lowest id will be numbered from 0 to 3, the next cluster from 4 to 7 and so on.

On every processor a pvmd is loaded which needs 190000 + \( n \times 1600 \) bytes of memory with \( n \) the number of total allocated processors. The additional memory needed by the pvmd depends on the communication behavior of the loaded application(s) e.g. the size of messages and the number of messages send to the node that are not yet received by one of the tasks running on it.

When sending a message in PVM one first has to create a send buffer with \texttt{pvm_initsend()}. The send buffer is used to store messages or parts of messages. With \texttt{pvm_pk()}; one can pack data into such a send buffer. Packing is nothing more than that the pvmd allocates memory to contain the data to be send and copies it there. This means the programmer can re-use the variables containing the data that he just packed. With the execution of \texttt{pvm_send()}; the pvmd knows the message is complete and can be send.

For the pvmd we can make a distinction between intra-processor and inter-processor communications. In the first case the pvmd will offer the message to the receiving task when it is requested. In the latter the data will be transferred to the pvmd that is serving the task the message is for.

Memory once allocated by the pvmd for communication purposes will not be freed until termination of the pvmd. Memory that is already allocated but not in use anymore will be re-used. This means one has to be careful when measuring the performance when communications are concerned. Allocating memory is much more expensive than reusing it.

\subsection{1.4 UNITY}

The basic problem in programming is the management of complexity, therefore UNITY was designed to avoid concerns about e.g. the architecture a program has to run on. The first concern of UNITY is to design a solution for a problem and not the implementation in a certain language on a particular architecture.

In [Chan 01] a computational model and proof system is introduced called UNITY, after the way K.M. Chandy and J. Misra like to view their theory: Unbounded Nondeterministic Iterative Transformations. They use the phrase ‘a UNITY program’ for ‘a program in unbounded nondeterministic iterative transformation notation’: A UNITY program consists of a declaration of variables, a specification of their initial values and a set of multiple-assignment statements. A program execution starts from any state satisfying the initial condition and goes on forever; in
each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection selection is constrained by the following 'fairness' rule: Every statement is selected infinitely often.

UNITY is a state-transition system that does not describe when, where and how assignments are to be executed. It also does not specify how an implementation may halt a program execution, termination is regarded as a feature of an implementation. The only thing UNITY describes is what should be done in the sense that it specifies the initial state and the state transformations.

Besides the initial state there is another special state called fixed point. A state is called fixed point if and only if execution of any statement of the program, in this state, leaves the state unchanged. To characterize the fixed points of a program one can use a predicate called FP.

1.4.1 Program notation

We will not describe the complete structure of a UNITY program. We will leave out the parts that are not necessary for reading and understanding this thesis. For a complete description we refer to [Chan 01]. A UNITY program consists of 4 sections as can be seen in this part of its grammar:

\[
\text{program} \rightarrow \text{Program} \\
\text{declare} \rightarrow \text{declare-section} \\
\text{always} \rightarrow \text{always-section} \\
\text{initially} \rightarrow \text{initially-section} \\
\text{assign} \rightarrow \text{assign-section} \\
\text{end}
\]

The declare-section names the variables used in the program and their types. In the always-section certain variables are defined as functions of others. The initially-section is used to define initial values of some of the variables; uninitialized variables have arbitrary initial values. The assign-section contains a set of assignment statements.

We will introduce the notation of the assign-section in an informal way using the examples given in [Chan 01]:Assigning to a single variable is well understood. UNITY allows a number of variables to be assigned simultaneously in a multiple assignment, as in

\[
x, y, z := 0, 1, 2
\]

Such an assignment can also be written as a set of assignment-statements separated by ||, as in

\[
x, y := 0, 1 || z := 2
\]

or

\[
x := 0 || y := 1 || z := 2
\]

The variables to be assigned and the values to be assigned to them may be described using quantification rather than enumeration. Thus

\[
< || i : 0 \leq i \leq N :: A[i] := B[i] >
\]

denotes the assignment

\[
\]
The assignment part of a conditional assignment-statement will not be executed when the boolean expression evaluates to false, as in

\[
sum, i := sum + A[i], i + 1 \quad \text{if } i < N
\]

where \(A[i]\) is added into \(sum\) and \(i\) incremented, provided \(i\) is less than \(N\). In the following 2 examples an identity matrix is assigned to the 2-dimensional array \(U[0..N,0..N]\)

\[
< \|| i, j : 0 \leq i \leq N \land 0 \leq j \leq N \land i \neq j :: U[i,j] := 0 > \\
|| < \|| i : 0 \leq i \leq N :: U[i,i] := 1 > 
\]

or

\[
< \|| i : 0 \leq i \leq N :: U[i,i] := 1 \\
|| < \|| j : 0 \leq j \leq N \land i \neq j :: U[i,j] := 0 > 
\]

The assign-section specifies the statements of the program. The symbol \(||\) acts as a separator between the statements. The set of statements of a program is fixed at all times; statements are not created or deleted during program execution. Also the number of statements should be finite. An example of an assign-section is e.g.

\[
| x, ch := x + 1, true \quad \text{if } ch = false \\
| y, ch := y + 1, false \quad \text{if } ch = true 
\]

where \(x\) and \(y\) will be increased in an alternating way. The previous example of the identity matrix can also be written as

\[
< \|| i, j : 0 \leq i \leq N \land 0 \leq j \leq N \land i \neq j :: U[i,j] := 0 > \\
|| < \|| i : 0 \leq i \leq N :: U[i,i] := 1 > 
\]

or

\[
< \|| i : 0 \leq i \leq N :: U[i,i] := 1 \\
|| < \|| j : 0 \leq j \leq N \land i \neq j :: U[i,j] := 0 > 
\]

The first example has two statements, one for assignment to the off-diagonal elements and the other for assignment to the diagonal elements. The last example has \((N + 1)\) statement lists, each list corresponding to one value of \(i\). Each list, in turn, has two statements: one for assigning to a diagonal element and the other for assigning to all off-diagonal elements in that row.

1.5 Notation

The term \textit{cluster} will be used to denote a single PowerXplorer, which is a parallel computer. With \textbf{PowerXplorer} we will mean from now on the complete set of 4 clusters. Every cluster contains 4 \textit{processing-units} which we will call \textit{processors}. More about the PowerXplorer and it’s processors can be found in section 1.2 on page 4.

We will not make a distinction between PVM and PowerPVM which is an adapted version of PVM for the PowerXplorer. More about PowerPVM can be found in section 1.3.3 on page 8.
To avoid confusion regarding the word task we will use it only to identify a program running, or able to run, on a processor. Note that a processor can host more than 1 task.

An guarded skip will always be written between square brackets. Note that if b then skip is not equivalent to [if b then skip]. The latter is computationally equivalent to while not(b) do skip (busy waiting) but is assumed to consume less processor time when implemented.

For the mathematical associative operation max, we will use an uparrow. The ↑ can be used as an infix or a prefix operator. Which way is meant will be clear from the context. In the same way we will use ↓ instead of min.
2 Timing

In order to make efficient implementations, we need to know more about the performance of the PowerXplorer. First we will explain how we did the measurements and why we did them this way. Note that all measurements in this thesis will follow these guidelines. In section 2.2 we will investigate aspects of the computation performance followed by a section about communication performance.

2.1 How to measure

It is very important to know exactly what one is measuring and that, when more than one sample is taken, the state of the system in which the measurements are taken is always the same. We will explain, by using an abstract example, how the data in this thesis is collected and why we did it this way.

Note that we are using the PARIX function `TimeNow()`. With `TimeNow()` we are measuring the time on the processor a task is running on. We could also use other functions provided by the operating system, like `time()`, but then we are measuring the system time and not the local time on the processor used by our task\(^3\). When we talk about time needed for a certain operation we mean the difference of the local time before and after completion of the operation. Note that the time on every processor may be different and therefore one should not compare time-stamps from different processors.

Although it is possible to translate the value of `TimeNow()` into seconds\(^4\) we decided not to do this. We are only interested in the proportions of the results.

```c
int startTime, endTime, totalTime = 0;

while ( i = 0; i < nrMeasurements+2; i++ )
{
    receive_message_other_tasks_are_ready_to_start;

    startTime = TimeNow();
    /* The code to measure */
    endTime = TimeNow();

    /* Do not use first 2 results */
    if (i > 1) totalTime += endTime - startTime;
}
printf("Time needed: \%d\n", totalTime/nrMeasurements);
send_message_other_tasks_can_end;
```

We can, in our case, describe the state of the system by the state of the tasks, the state of the PVMD and the location of the tasks.

**Location of tasks:** It is important to always map the tasks in the same way on the processors. Suppose we have 2 processors and 4 identical tasks. It is obvious that when we map all 4

\(^3\)A drawback of `TimeNow()` is that the timer resets every 71 minutes without a warning.

\(^4\)This is done by dividing the value by the constant `CLOCK.TICK`, which is defined by the system.
tasks on the same processor we will get an other performance then when we divide the tasks equally over the 2 processors.

Also the location of the used processors should be considered when performing communications. The PowerXplorer has a non-uniform communication cost model. The cost of communication depends on the location of the tasks involved. We distinguish, roughly in order of decreasing cost, inter-cluster communication (between tasks that reside on processors in different clusters), inter-processor communication (between tasks that reside on different processors within the same cluster), and intra-processor communication (between tasks that reside on the same processor). See also section 1.2 on page 4 for more information about the PowerXplorer.

**State of the tasks:** It is important that the other tasks are ready to participate in the calculation. Therefore we let them send a message to be sure they are in the state we would like them to be in. This is mostly a state where a message from an other task is expected in order to prevent tasks from starting their calculation to early.

![Figure 5: A possible trace](image)

**State of the system:** We decided to measure everything in one run, excluding the first 2 results. Reason for this is that the first results differ much from the others, specially when communications are involved with lots of data.

PVM needs to allocate memory for communications, see section 1.3.3 on page 9, which is an expensive operation. Because PVM keeps this memory allocated, even when it is not needed at the moment, future runs don’t need this expensive allocation. But why do we exclude also the second measurement? Remember we are working on a parallel system. Therefore 2 communications that occurred in the first measurement after each other (left side of figure 6) could be now executed in parallel (right side of figure 6) which needs more memory. Test we did showed that the deviation in the results after 2 runs is neglectable.  

![Figure 6: 2 possible communication traces](image)

The last statement in our abstract program is very important. We do not want tasks to exit before all the results are collected. An exiting tasks cause extra work for the PVM daemon which could interfere with our task.

---

5 All measurements we did in this thesis were checked for results with a large deviation and removed when found. The few results we removed did not indicate that our barrier of 2 measurements is to low.
2.2 Computation performance

In this section we would like to examine how the computational performance of the PowerXplorer behaves under different circumstances. We will try to determine if there is a big scheduling overhead when running more tasks on the same processor and if a task blocked by a `pvm_receiv()` will consume processor time.

To answer the above questions we used a small program that calculated a certain function several times, always with the same arguments. No communications were made by this program and no interaction with any input/output device was made. After completion of the computation, the program did not terminate but entered an idle state in order not to consume extra processor time. This ensured us that always the same computation was measured.

The first row in figure 7 contains the number of tasks running on the same processor. The middle row is the time needed to complete the computations on all tasks and the last row shows the factor compared to the situation where just one task is started. It is obvious that there is no scheduling overhead, we even measure a small speedup!

<table>
<thead>
<tr>
<th>Nr tasks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>6727121</td>
<td>13453045</td>
<td>20175876</td>
<td>26898697</td>
<td>33621517</td>
<td>60512807</td>
</tr>
<tr>
<td>Factor</td>
<td>1</td>
<td>1.9998</td>
<td>2.9992</td>
<td>3.9985</td>
<td>4.9979</td>
<td>8.9953</td>
</tr>
</tbody>
</table>

Figure 7: Scheduling overhead

To check how much processor time an idle task consumes, we started a few idle tasks on the same processor and 1 calculating task. The results can be found in figure 8 and show clearly that an idle task is not consuming any processor time.

<table>
<thead>
<tr>
<th>Nr idle tasks</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>6727121</td>
<td>6727120</td>
<td>6727121</td>
<td>6727121</td>
<td>6727121</td>
</tr>
</tbody>
</table>

Figure 8: How idle is a blocking `pvm_receiv()`

2.3 Communication performance

First of all, we would like to point out that there are different ways one can send a message in PVM. When performance is concerned one should keep the following rules in mind:

- Broadcasting your message (`pvm_mcast()`) is more efficient than sending it to every task separately. An other advantage of broadcasting is that less memory is used for the communication. See section 1.3.3 on page 9 for more information.
- Try to minimize the number of messages. Twice sending an integer is more expensive than sending once 2 integers.
- Because packing a variable is quite expensive, one should try to send arrays instead of separate variables.

\[\text{Except a `pvm_receiv()` to know when to start and a `pvm_send()` to signal that it is ready with the computation.}\]
Because of the physical structure of the PowerXplorer, it is important to make a distinction between inter-cluster, inter-processor and intra-processor communications. Intra-processor communications don’t need to use an physical communication channel, inter-processor communications can use an internal communication channel while inter-cluster communications need to use an external channel. In this section, we will try to determine how these different kinds of communications are related to each other in terms of performance.

For measuring we started two tasks, A and B, on the PowerXplorer. Task A was sending a message to task B, B received this message, unpacked it and returned it back to A. This experiment is also known as the ping-pong experiment. Note that when we are talking about a ping-pong communication in this section, we mean one run of the ping-pong algorithm which are in fact two communications.

### Ping-pong A

```c
for (i=0; i<nrCommunications; i++)
{
    pvm_initsend( PvmDataRaw);
    pvm_pkint( &package, packageSize, 1);
    pvm_send( tid[B], MSG_TAG);

    pvm_recv( tid[B], MSG_TAG);
    pvm_upkint( &package, packageSize, 1);
}
```

Note that we include the initialization of the send-buffer (`pvm_initsend()`), the packing of a package (`pvm_pkint()`) and the unpacking of a package (`pvm_upkint()`) in the measurement. This is done because we would like to measure the total costs of a communication and not only the time needed to transfer a package from one to another task.

Our first goal is to determine if the successive execution of communications is influencing our results. Figure 9 shows us that this is not the case for intra-processor communications with a package containing 1 integer. Other experiments with larger packages, inter-processor and inter-cluster communications show the same results. Therefore we can safely restrict our experiments to the situation where we perform 1000 ping-pong communications. This amount is large enough to eliminate variations in the results and doesn’t occupy the system unnecessarily.

<table>
<thead>
<tr>
<th>nrCommunications</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>399</td>
<td>804</td>
<td>1203</td>
<td>3996</td>
<td>3997</td>
<td>399755</td>
</tr>
<tr>
<td>Factor</td>
<td>1</td>
<td>2.015</td>
<td>3.015</td>
<td>10.02</td>
<td>100.2</td>
<td>1002</td>
</tr>
</tbody>
</table>

**Figure 9:** intra-processor communication with a package of 1 integer

Now, let us investigate what happens when we change the size of the transmitted packages. In figure 10 one can see the results for the 3 possible communication types. All the time-data is divided by 1000 in order to get an graph representing 1 ping-pong communication. It is obvious that the time needed for the communications depends linear on the number of integers transmitted.

In [Fost 01, §3.3.1] the cost of sending a message between two tasks is represented by two parameters: the message startup time \( t_s \), which is the time required to initiate the communication, and the transfer time per (typically four-byte) word \( t_w \), which is determined by the physical bandwidth of the communication channel linking the source and destination processors. In our case we will
Figure 10: Costs of a ping-pong communication.

use $t_i$, the time per integer\(^7\), instead of $t_w$. The time $T_{\text{msg}}$ needed for a communication of size $L$ can be expressed with the simple communication cost model: $T_{\text{msg}} = t_s + t_i L$. As can be seen in figure 10 this model can be used to represent the communication costs on the PowerXplorer. The actual values for this model can be found in figure 11. Note that those values represent a normal communication and not a ping-pong communication.

<table>
<thead>
<tr>
<th></th>
<th>$t_s$</th>
<th>$t_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>intra-processor</td>
<td>199.2</td>
<td>0.62</td>
</tr>
<tr>
<td>inter-processor</td>
<td>178.4</td>
<td>4.12</td>
</tr>
<tr>
<td>inter-cluster</td>
<td>197.3</td>
<td>4.22</td>
</tr>
</tbody>
</table>

Figure 11: Values for $t_s$ and $t_i$

From figure 10 and 11 we can conclude that intra-processor communication is the most efficient, specially when packages get large. The difference between inter-processor and inter-cluster communication is rather small. Therefore no real gain can be expected when one tries to optimize the mapping of tasks on the available processors in this way.

The same data is plotted in figure 12, but now the x-axis only ranges from 1 to 10. It is clear that in the case of intra-processor communication the costs are more expensive for small packages. This can be explained by the fact that communication has to be handled completely by the processor. After sending a message, the processor has to 'receive' it, 'process' it and forward it to the receiving task. After the receiving task accepted the message the processor has to 'clean up' the old message. Note that this extra work is a constant factor. See section 1.3.3 on page 9 for more information.

\(^7\)The size of an integer in words depends on the system. We are not interested in the actual size in words.
Figure 12: Costs of a ping-pong communication.
3 Parallel asynchronous computation of an associative function

This section will cover the implementation of an abstract parallel program for calculating an associative function. For an arbitrary associative function $f$ holds that $f(f(a, b), c)$ is equal to $f(a, f(b, c))$. Examples of such functions are the addition and multiplication of the real numbers. Also the multiplication of matrices is associative providing that they have the right dimensions to allow multiplication. Note that the subtraction and the division of real numbers in not associative.

We will present the abstract program from [Horv 01, p 89] and transform it into a PVM program. For the derivation we refer to the original document. A short introduction, the program and some important invariants will be presented in this paper. After implementing the program, the mapping of tasks on actual processors and the performance will be discussed. We will end the section with some general remarks.

3.1 The abstract program

Let $H$ be a set and let $\circ : H \times H \rightarrow H$ denote an arbitrary associative binary operator over $H$. An arbitrary $a \in H$ will be denoted by $\ll a \gg$. In general a finite sequence $l \in H^*$ will be denoted by $\ll l_1, \ldots, l_l \gg$. Let $a \in H$ and $l, m \in H^*$, then the function $f : H^* \rightarrow H$ describes the single or multiple application of the associative operator $\circ$ and is defined by:

$$
\begin{align*}
    f(\ll \gg) &= e_f \\
    f(\ll a \gg) &= a \\
    f(\ll a, e_f \gg) &= f(\ll e_f, a \gg) = a \\
    f(\ll l++ m \gg) &= f(\ll f(l), f(m) \gg), \text{ if } |l| \geq 1 \lor |m| \geq 1
\end{align*}
$$

Observe that for a finite sequence $l \in H^*$ of at least three:

$$
f(\ll l_{i1}, \ldots, l_{i1} \gg) = f(\ll l_{i1}, f(\ll l_{i1-1}, \ldots, l_{i1} \gg) \gg) = f(\ll f(\ll l_{i1}, \ldots, l_{2} \gg), l_{i1} \gg)
$$

Now we introduce function $G : [1 \ldots |l|] \rightarrow H$ with the following specification:

$$
G(i) = f(\ll l_{i}, \ldots, l_{i} \gg)
$$

In the abstract program array $g$ is introduced to store the values of $G$. The program should reach a fixed point where the following holds:

$$
FP \Rightarrow \forall (i : i \in [1 \ldots |l|] : g(i) = f(\ll l_{i}, \ldots, l_{i} \gg) \ (1)
$$

For the calculation of $f$ for subsequences $f(\ll l_{i}, \ldots, l_{i} \gg)$ indexed by the elements of an arbitrary $[u \ldots v] \subseteq [|l| \ldots 1]$ interval we introduce function $h : [1 \ldots |l|] \times \mathbb{N}_0 \rightarrow H$ where $h(i,k)$ denotes the value of $f$ for the sequence of which the first element is $l_i$ and its length is $2^k$:

$$
h(i, k) := \begin{cases} 
    f(\ll l_i, \ldots, l_k \gg) & \text{if } i - 2^k + 1 \leq 1 \\
    f(\ll l_i, \ldots, l_{i-2^k+1} \gg) & \text{if } i - 2^k + 1 \geq 1
\end{cases} \ (2)
$$

It is clear that $h(i, \lfloor \log(i) \rfloor) = f(\ll l_{i}, \ldots, l_{i} \gg) = G(i)$. In the abstract program array $gs$ is introduced to store the known values of function $h$. 

19
\[ FP \Rightarrow \forall (i : i \in [1 \ldots n] : k(i) = [\log(i)]) \land g(i) = gs(i, [\log(i)]) \]  
\[ \forall (i : i \in [1 \ldots n] : k(i) \leq [\log(i)] \land \forall (k : k \leq k(i) : gs(i, k) = h(i, k)) \]  
\[ \forall (i : i \in [1 \ldots n] : t(i) = 2^{k(i)}) \]  

The program below is a solution for the problem specified by (3), (4) and (5) and the demand that the program will reach its fixed point FP. Notation \( i = \llbracket i \rrbracket \) is used for the abbreviation of \( n \) statements. Each statement is instantiated from the general form by substituting the dummy variable \( i \) by a concrete value.

**Abstract Program**

\[
\begin{align*}
  s_0 & : \quad i = \llbracket 1 \ldots n \rrbracket \quad g(i, 0), t(i), k(i) := f(\llbracket l_i \rrbracket), 1, 0 \\
  S : \{ & \quad i = \llbracket 1 \ldots n \rrbracket \\
  & \quad gs(i, k(i) + 1), t(i), k(i) := \\
  & \quad \begin{cases} 
  f(\llbracket gs(i, k(i)), gs((i - t(i)), k(i)) \rrbracket), 2 * t(i), k(i) + 1 \\
  \text{if} \quad (i - 2 * t(i) + 1 \geq 1) \land \\
  \quad (k(i - t(i)) \geq k(i)) \\
  f(\llbracket gs(i, k(i)), gs((i - t(i)), k(i - t(i))) \rrbracket), 2 * t(i), k(i) + 1 \\
  \text{if} \quad (i - 2 * t(i) + 1 < 1) \land \\
  \quad (k(i - t(i)) = [\log(i - t(i))]) \land \\
  \quad (i - t(i) \geq 1) \\
  \end{cases} \\
  i = \llbracket 1 \ldots n \rrbracket \quad g(i) := gs(i, k(i)) \quad \text{if} \quad (k(i) = [\log(i)])
\end{align*}
\]

3.2 Transformation of the program

The abstract program can be implemented using any number of tasks ranging \([1 \ldots n]\). At this moment, we choose to design a solution that uses precisely \( n \) tasks, expecting the transformation to fewer tasks an easy process. Further, we map every variable indexed by \( i \) to the \( i \)th task.

Fortunately every variable, used on the left side of an assignment, refers to a local variable. The only variables accessed outside the local memory are \( gs \) and \( k \), both indexed by \( i - t(i) \).

3.2.1 The conditional assignments

The program contains three boolean expressions (guards):

\[ (i - 2 * t(i) + 1 \geq 1) \land (k(i - t(i)) \geq k(i)) \]  
\[ (i - 2 * t(i) + 1 < 1) \land (i - t(i) \geq 1) \land (k(i - t(i)) = [\log(i - t(i))]) \]  
\[ k(i) = [\log(i)] \]  

We observe that \( k(i) \) is a non-decreasing value, initialized by 0. We can prove that once selection (8) is true, the program gets stuck in it's local FP. When (8) is true for every task \( i \) the program reached it's FP and we can halt the execution of the program.

\[ k(i) = [\log(i)] \Rightarrow t(i) = 2^{k(i)} \]
\[ g(i) = 2^{[\log(i)]} \]

\[ \geq i \]

Because only local variable \( g(i) \) changes after \( k(i) = [\log(i)] \) and \((8) \Rightarrow \neg(6) \land \neg(7)\) the program reaches its local FP. By using the first conjunct of \((6) \land (7)\) and the fact that \( t(i) \) is non-descending, it is clear that \((7) \Rightarrow \neg(6)\) and \((6)\) will never become true again.

Selection \((7)\) can only once evaluate to true. In order to evaluate \(2\) times to true, \((i - t(i) \geq 1)(t(i) := 2 \times t(i)) = (i - 2 \times t(i) \geq 1)\) should be valid, but this is not the case because of the first conjunct in the boolean expression.

The boolean expressions in \((6) \land (7)\) depend on \(k(i - t(i))\). Unfortunately this is not a local variable and the value should be communicated every time the guard is evaluated. Therefore we will strengthen the guard [Feij 01, p 21][Misr 01] and replace the occurrence of the expressions where \(k(i - t(i))\) occurs by fresh thought boolean variables \(b_i\) and \(c_i\) and demand:

\[ b_i \implies (k(i - t(i)) \geq k(i)) \]  
\[ c_i \implies (k(i - t(i)) = [\log(i - t(i))]) \]

Note that strengthening a guard, while maintaining the correctness of the annotation, can affect progress. Choosing false for \(b_i\) and \(c_i\) is correct when the annotation is concerned, but will cause deadlock when we look at progress. Therefore we should check if progress is not hampered by our choice.

We will use these observations to rewrite the boolean expressions from the abstract program into PVM:

---

**Implementation of the boolean expressions**

```java
while (i-2*t(i)+1 >= 1)
{
    /* b.i */
    while not(b.i);
    ...

    /* i-2*t(i)+1 < 1 */
    if (i-t(i) >= 1)
    {
        /* c.i */
        while not(c.i);
        ...
    }
    /* k(i) = Ceil(log(i)) */
    ...
}
```

---

3.2.2 Adding communications

We will postpone the choice for \(b_i\) and \(c_i\) and hope that this choice will be more easy in a later stadium. For now, we concentrate on the variables that are not available local: \(k(i - t(i))\), \(gs(i - t(i), k(i))\) and \(gs(i - t(i), k(i - t(i)))\).

---

\(^{8}\)The proof can be found on page 72, section A.1.
The only way we can know the values of these variables is by receiving information from task \( i - t(i) \); this is the only task which has access to those values. There are a few possible solutions for communicating these values:

1. Add an extra task to the program that acts as a buffer. Every time the value of a 'global variable' changes, the owner will send the value to this extra task. Other tasks will send requests for the value of variables they have no access to and the extra task will return the value to them.

2. Communicate directly between the task that calculated and the one that needs the value:
   (a) Request the values from the task which owns them.
   (b) Send the values to a buffer on the task that will need the values in the future.

The first solution costs an extra task and generates more communications than strictly necessary. E.g. for big \( n \), the amount of communications needed for sending the calculated values of \( gs \) to the buffer task is of order \( n \log(n) \). Furthermore at least \( n \log(n) \) requests have to be made and the same number of answers will be returned.

Solution 2(a) demands from every task a constant monitoring of the communication channels used for requesting values. This can get quite time consuming and a buffer is needed to store requests that can not be handled immediately. An other major disadvantage is that every task has to stay alive until it is sure that no requests will be made in the future\(^9\) by any other task.

The last one, solution 2(b), needs only half the amount of communications compared to the previous one. Because values will be sent directly to the task that needs them, no extra request communication has to be made. The buffer needed in every task to store received values can be calculated and will be of length \( \lceil \log(i) \rceil \) for task \( i \).\(^{10}\) The disadvantage of this solution is that one has to calculate the id's of the receiving tasks. Fortunately, we can show that the value of \( gs(i, k(i)) \), for \( k(i) < \lceil \log(i) \rceil \), is needed at task \( i + 2^{k(i)} \) and \( gs(i, \lceil \log(i) \rceil) \) has to be send to \( i + 2^{\lceil \log(i) \rceil + 1}, i + 2^{\lceil \log(i) \rceil + 2}, \ldots, i + 2^{\lceil \log(n-1) \rceil} \).

It is clear that option 2(b) is the one that would be the most efficient in our case. But which values do we need to send and at which moment?

We would like to have
\[
gs(i - t(i), k(i)) = h(i - t(i), k(i)) \implies k(i - t(i)) \geq k(i) \quad (11)
\]
because this we can use to establish (9) and (10). Performing a read operation on \( gs(i - t(i), k(i)) \) to establish \( b_i \) and one on \( gs(i - t(i), k(i - t(i))) \) to establish \( c_i \) will be sufficient. But to be able to do this, we need to establish (11).

Note that, by contraposition, we can replace formula (11) by the equivalent formula
\[
k(i - t(i)) < k(i) \implies gs(i - t(i), k(i)) \neq h(i - t(i), k(i)) \quad (12)
\]
Fortunately, we can make (12) invariant by adding
\[
\forall(i : i \in [1 \ldots n]: \forall(k : k \leq \lceil \log(i) \rceil : gs(i, k) \neq h(i, k)))
\]
as a precondition of the program. Our only concern left will be dealing with the matching sending operations.

---

\(^9\)In our case, the number of requests can be calculated, but our task has to stay alive until the last request has been made and with that it occupies memory that could be used by other tasks.

\(^{10}\)When tasks will be blocked after sending a message until it is received by the receiving task, no buffer is needed. The order in which messages will be send becomes important now; a wrong order can cause deadlock.
Executing the communications after all the local values of gs are calculated will hamper progress; all tasks together will behave like a sequential program. Therefore we will advance the communication of a variable gs(i,j), for arbitrary j, to the earliest possibility where gs(i,j) = h(i,j). There is no danger for deadlock. Task 1 will always end\(^\text{11}\) and one can show that after ending task i, task i + 1 will end also.

### 3.2.3 The implementation

The implementation into PVM is straightforward now. Note that we decided to use exactly n tasks which we will denote by nrTasks in the implementation and that i denotes the id of a task. The presented code does not include root.c or implementation details with relation to the implementation of communication in PVM. To improve readability we deviated a bit from the complete code like it is presented in section C.2, page 79.

#### Implementation of node.c

```c
/* S0 */
gs[0] = value; t = 1; k = 0;
if (1<nrTasks) send.gstrans.to.task.1;

/* S */
while (i-2*t+1 >= 1)
{
    /* b.i */
    value = receive.value.from.task.(i-t-1);
    /* b.i */
    gs[k+1] = fFunction(gs[k], value);
    t *= 2; k++;
    /* gs[k] == h(i,k) */

    /* Is our result needed on any other task? */
    if (i+t<=nrTasks) send.gs[k].to.task.(i+t-1);
}

/* i-2*t+1 < 1 */
if (1-t >= 1)
{
    /* c.i */
    value = receive.value.from.task.(i-t-1);
    /* c.i */
    gs[k+1] = fFunction(gs[k], value);
    t *= 2; k++;
    /* gs[k] = h(i,k) */

    /* Is our result needed on any other task? */
    if (i+t<=nrTasks) send.gs[k].to.task.(i+t-1);
}

/* k == Ceil(log(i)) */
t *= 2;
while (t+i <= nrTasks)
{
    /* Sending our result to tasks that need it */
    send.gs[k].to.task.(t+i-1);
    t *= 2;
}

11Remember we are dealing with a system that handles sending in a non blocking way.
```
3.3 Mapping of tasks on processors

It is not always possible to use the same number of processors as there are tasks. In those situations one has to make some sort of mapping from tasks to actual processors. Of course this can be done in many different ways, but not all of them are efficient. To find a suitable mapping for our program, we will first look at the following topics:

1. Communications needed at each task.
2. Calculation time needed for each task.
3. Memory needed by each task.

3.3.1 Communications needed at each task

In most parallel systems, communications between tasks hosted on the same processor can be performed much faster than communications between tasks on different processors. Therefore it is worth to try to find a mapping that tries to maximize the number of intra-processor communications. In this case it is clear that tasks communicate the most with tasks that are nearby. Figure 13 is a graph where the vertices represent tasks and the edges a communication between 2 tasks.

![Graph showing communications between tasks](image)

Figure 13: Communications between tasks

In figure 14 the function \( \lceil \log(i) \rceil + \lceil \log(n - i) + 1 \rceil \) is plotted which represents the amount of communications every task has to perform. The picture shows clearly that the amount of communications made by the first and the last tasks is less than by the other tasks. For big \( n \), the number is quite constant for the middle range of tasks and not much performance can be gained here. In figure 14 also the number of receiving communications every task has to perform is plotted. The plot of the number of sending communications can be obtained by mirroring the previous one at \( x = \frac{1}{2} n \).
3.3.2 Calculation time

Suppose that every application of $f$ can be calculated in exactly one time slot. Task $i$ will need $\lceil \log(i) \rceil$ time slots to calculate $gs[i]$. This is clearly a non-decreasing function in $i$ and has the same shape as the plot for the number of receiving communications in figure 14. For big $i$, the number of time slots needed is quite constant, so just a small performance gain can be obtained here. In situations where the calculation of function $f$ demands lots of processor time, one can better try to balance the load over the available processors.

3.3.3 Memory needed

In every task $i$, memory is allocated for array $gs$. This array is used to store the values of function $h$ and has size $\lceil \log(i) \rceil + 1$. Fortunately, this array is not needed for the final result; it can be replaced by just one variable like is done in the appendix, section C.2.3. As a result of this, every task needs the same amount of memory and no gain can be obtained in this manner. One still has to bare in mind that there is still memory needed for buffering the communications and that the amount is equal to the size of $gs - 1$. We will not consider this memory anymore in this section.

3.3.4 The 'best' mapping

Unfortunately, there is no 'best' mapping for this problem. Which mapping gives the best results depends heavy on the complexity of the function $f$, the costs of a communication between tasks on the same processor and the costs of a communication between tasks on different processors.

A possible way of mapping tasks on processors is the block distribution, also called a linear mapping. When $P$ is the number of available processors and $N$ the number of tasks then every task $t$, with $1 \leq t \leq N$, will be mapped on the processors with the following formula: $t \mapsto (\lfloor (t - 1) \div Q \rfloor) + 1$, with $Q = \lceil N/P \rceil$. Figure 15 shows such a mapping. Because, in the case of the associative function, this mapping maximizes the number of intra-processor communications, it is the best mapping when $f$ is easy to compute and the non intra-processor communication costs are high.
For a more complex function \( f \) one should not try to maximize the number of intra-processor communications but to balance the load. For the associative function load-balancing can be done by using a grid distribution, also called a cyclic mapping. Analogous to the block distribution there exists a function for the mapping of tasks on processors for the grid distribution: \( t \mapsto ((t - 1) \mod P) + 1 \). An example of such a mapping can be found in figure 16. Note that in this figure \( P \), the number of processors, is a divider of \( N \), the number of tasks.

### 3.4 Timing

Maybe the most important question one can ask after implementing a parallel program is it’s speedup. Before we can determine this, we need to choose a mapping. We will restrict us to the two mappings presented in section 3.3, the block and the grid distribution, from which we will take the most efficient one.

Before we proceed we need to explain the terms msg.size \( x \) and weight of calculation. Msg.size \( x \), with \( x \in \mathbb{N} \), denotes the size of the used communications. By adjusting the size of communications we can simulate a system with more expensive communication costs. The weight of calculation is a natural number which states how often a certain function is calculated, every time with the same arguments. A low weight represents an easy to calculate function \( f \), a high weight a complex function \( f \).

#### 3.4.1 Grid vs. block distribution

Figure 17 shows us that for a complex \( f \) a grid distribution is indeed the most efficient mapping. In contrast with what we expected in section 3.3.4 we see that also for a simple function \( f \) the grid distribution is the most efficient mapping. Apparently, the costs of a communication on the PowerXplorer are not expensive enough to let us choose for a block distribution.
3.4.2 Speedup

The speedup $S(P)$, with $P$ the number of processors in use, can be expressed formally by $S(P) = T_{seq}/T(P)$ where $T_{seq}$ represents the running time of an sequential implementation of the algorithm and $T(P)$ the running time of the parallel implementation on $P$ processors. In other words $S(P)$ is the factor by which execution time is reduced on $P$ processors. The speedup for our implementation using a grid distribution can be found in figure 18.

In figure 18 we can see that increasing the weight of $f$ has a positive influence on the speedup and that the increase of the speedup is the largest for small weights. This behavior can be explained when we take a look at the extra costs concerning the pvmd and communications. These costs are constant and the influence they have on the total computation time decreases with growing weight. Note that the speedup for 1 processor is stable from weight $\approx 250$, the speedup for 4
processors from weight ≈ 1000 and the speedup for 8 processors from weight ≈ 2000. These values are in proportion to 1 : 4 : 8, the number of used processors.

3.4.3 Upper bound for the speedup

But how good are those results? Could we get a much better speedup when we used a different machine where the costs of communications is cheaper? In order to answer those questions, we will try to find an upper bound for the speedup on an imaginary system with no communication costs.

First let us find out how many times our parallel implementation has to calculate \( f \). Let \( T \) the number of tasks and \( i \) an arbitrary task with \( 1 \leq i \leq T \). We know that task \( i \) has to calculate function \( f \) \([\log(i)]\) times. So the total number of times function \( f \) has to be calculated is\(^{12}\):

\[
\sum_{i=1}^{T} [\log(i)] = \sum_{i=1}^{[\log(T)]} T - 2^{(i-1)} = [\log(T)] \cdot T - \sum_{i=1}^{[\log(T)]} 2^{(i-1)}
\]

\[
= [\log(T)] \cdot T - 2^{[\log(T)]} + 1
\]

(13)

(14)

Now let us try to find some upper bounds for the case where 16 tasks are used to calculate an associative function. A sequential program needs to calculate \( f \) 15 times. When executing our implementation on just 1 processor, this processor has to calculate \( f \) \([\log(16)]\) \(\times 16 - 2^{[\log(16)]} + 1 = 49\) times. Therefore our speedup when using just 1 processor can be maximal 15/49 ≈ 0.3.

When using more processors we should look at the processor that has to make the most calculations. For a grid distribution this is processor number \( p_{\text{max}}(P,n) \) with \( p_{\text{max}} \):

\[
p_{\text{max}}(P,n) := \begin{cases} 
    P & \text{if } (n \mod P) = 0 \\
    n \mod P & \text{otherwise}
\end{cases}
\]

The number of calculations processor \( p_{\text{max}}(P,n) \) has to make in a grid distribution is:

\[
\sum_{i=1}^{[n/P]} [\log(p_{\text{max}}(P,n) + P \ast (i-1))] \]

(15)

Now we will apply this to the case where we have 16 tasks running on 4 processors. The maximum number of times a processor has to calculate \( f \) is:

\[
\sum_{i=1}^{[16/4]} [\log(4 \ast i)] = 4 + 4 + 3 + 2 = 13
\]

Therefore the upper bound of the speedup for 4 processors is 15/13 ≈ 1.15. The same way one can determine the upper bound for 8 and 16 processors which is 15/7 ≈ 2.14 and 15/4 = 3.75. In figure 19 one can find more upper bounds for different number of tasks.

In almost the same way one can calculate an upper bound for the block distribution. For 4, 8 and 16 processors the speedup is 15/16 ≈ 0.94, 15/8 = 1.875 and 15/4 = 3.75.

\(^{12}\)The proof of step (13) can be found on page 72, subsection A.2

28
3.4.4 Scaling of the parallel algorithm

In the previous sections we could see that the overhead that occurs when we go from a sequential algorithm to our parallel algorithm is enormous. Therefore we will take a look at the speedup with $T(1)$ as the basis instead of $T_{seq}$. The equation for the speedup becomes now: $S(P) = T(1)/T(P)$. For our calculation we will use the same imaginary system and formulae as introduced in section 3.4.3.

In figure 20 one can see the speedup for 16, 100 and 1000 tasks on a different number of processors. It is clear that one should not use more than 16 processors when only executing 16 tasks. One can also see that the program scales almost perfectly when executing 1000 tasks on 100 processors. Investing in extra processors does pay in this situation.

But now let us take a look at the situation where we run 100 tasks. Using exactly 100 processors we achieve a speedup of $573/7 \approx 81.9$, which is quite good. In spite of this one should be careful investing in extra processors. There is no difference in speedup when using between 50 and 67 processors. Increasing the number of processors from 50 to 91 results in the increase of the speedup with only 8. On the other hand adding the last 9 processors will increase the speedup with almost 30. Note that the last processor is actually superfluous.

A better look at the situation where we had 1000 tasks reveals that one has to add 244 to 500 already existing processors in order to gain some speedup. Our conclusion that investing in extra processors pays, only holds until about 250 pieces.

3.5 Remarks

3.5.1 When not to use this algorithm

Unfortunately, not every associative function can be calculated more efficient with this parallel program. Let function $f$ be a associative function that has to be calculated $n$ times over the same input. In stead of $f(x, x, x)$ we write $f^3(x)$. There exists a sequential algorithm that can perform the calculation of $f^n(x)$ in the same order of time, $\Theta(\log n)$.
In the following algorithm we will use multiplication as associative operator. Note that \( f^n(x) = x^n \).
It is easy to adjust this program for other associative functions.

**Sequential algorithm**

```c
int y = 1, z = x, k = n;

/\* y * z * k = z * n *\/
while ( k != 0 ) {
    if ( k % 2 == 0 ) {
        z *= z;
        k = k / 2;
    }
    else {
        y *= z;
        k--;
    }
}
/\* y = z * n *\/
```

It is obvious that the order of this algorithm is \( \Theta(\log n) \). Note that in the best case, with \( n \) a power of 2, it needs only \( \log(n) + 1 \) steps. In the worse case, where \( n = 2^i - 1 \) with \( i \in \mathbb{N} \), \( 2 \cdot [\log(n)] + 1 \) steps are needed.

A parallel algorithm exists, containing only 2 tasks, that will calculate \( f^n(x) \) in exactly \( [\log(n)] \) steps. An optimized version\(^\text{13}\) of the previous example is used that is able to compute \( f^n \), with \( n \) a power of 2, in \( \log(n) \) steps.

We rewrite \( n \) to \( a + b \) where \( a = 2^{[\log n]} \) and \( b = n - 2^{[\log n]} \). We can now rewrite \( f^n \) into \( f(f^a, f^b) \). It is obvious from the previous example that we can calculate \( f^a \) in \( \log(a) \) using just one task. During this calculation, the values \( f^1, f^2, f^4, \ldots, f^a \) will also be calculated. A second task can construct \( f^b \), using the theory that every number can be written as the sum of a finite number of

\(^{13}\text{Recursive doubling}\)
different 2-powers. At the end just one extra application of $f$ has to be performed to construct $f^n$. In the case where $n$ is a power of 2 this last step can be omitted. Clearly this computation can be done by 2 tasks in $\lceil \log(n) \rceil$ steps.

### 3.5.2 A more compact solution

When we are looking at the code of `node.c` in appendix C.2, page 79, we see that the code in the first two selection statements is exactly the same. The only difference is the selection itself ((6) and (7)). The first place they appeared was in the definition of function $h$ on page 19. It is possible to rewrite the definition in such a way that we lose the selection:

\[ h(i, k) ::= f(\ll a_i, \ldots, a_{(i - 2^i + 1)}) \gg \]

We can use this to rewrite the abstract program for $S$ into:

\[
S : \begin{array}{l}
\forall i=[1..n] \quad \text{gs}(i, k(i)), t(i), k(i) := \\
\quad f(\ll \text{gs}(i, k(i)), \text{gs}(i - t(i), k(i) \downarrow k(i - t(i)) \gg), 2 \cdot t(i), k(i) + 1 \\
\quad \text{if } (k(i - t(i)) \geq k(i)) \lor ((i - t(i) \geq 1) \land (k(i - t(i)) = \lceil \log(i) \rceil))
\end{array}
\]

Analogous to what we did before, we have to strengthen the guard and add the communications. The result of this can be found in the appendix, section C.2.3 page 81. Note that this program is equivalent to the technique called **recursive doubling**.

### 3.5.3 Topics not treated

In section 3.3.4 on page 25 we stated that there is no 'best' mapping for our algorithm. Later we concluded in section 3.4.1 that in the specific case of the PowerXplorer a grid distribution is more efficient than a block distribution. One could proof with use of formula (15) on page 28 that for e.g. 16 tasks on 4 or 8 processors there is no better mapping possible than the grid distribution when we do not consider communication costs. It could be interesting to investigate this also for other situations like 16 tasks on 6 processors.

All our measurements in section 3.4 were done with a function $f$ that needed a constant time, independent from it's arguments, to compute a result. We variated the weight of $f$, but only between different runs of our application and not between tasks participating in the same run. In most situations one will use a function $f$ that behaves less constant like e.g. the multiplication of matrices and the multiplication of integers. Note that multiplying by 2 or a power of 2 is less expensive than multiplying by e.g. 7. This can influence the amount of idle time, a topic we did not discuss, the choice for a mapping and the speedup.
4 Parallel Elementwise Processing

This section covers the implementation of the abstract program from [Fóth 01]. The original paper presents an abstract parallel algorithm for computing the values of elementwise processable functions.

A function is elementwise processable when the output can be obtained by processing each single data item from the input one after the other. In general the domain and the image of such functions consist of ordered sequences or sets. Some examples of operations which are mostly solved by elementwise processable functions are the merging of ordered sequences and the computation of the union of sets.

The presented parallel algorithm divides the input into blocks which can be processed independently from each other. A master process will dynamically distribute the blocks among the available processors where the blocks are processed with a sequential program. When the last block is processed, the solution will be computed from the partial results on the available processors.

We will number the abstract programs in this section the same way as is done in [Fóth 01].

4.1 Techniques, definitions and notations

In section 4.1.1 we will introduce some functions that will be used regarding to the sequence type. The next section will cover the total disjoint decomposition and how we will produce such a decomposition. Section 4.1.3 will explain formal what we regard as an elementwise processable function and introduces a sequential algorithm.

4.1.1 Sequence

The definition of sequence which is used comes from [WRMP 01, p 442] and differs from the definition in e.g. [Sips 01, p 6]. In the latter repetition of objects in a sequence is allowed, in the former not. Because the correctness of the abstract algorithm depends on the absence of repetition we will adopt the definition from [WRMP 01]. A sequence with base type \( T \) will be denoted by \( \text{seq}(T) \). Note that in a sequence the order of the elements is important; the sequence \( \langle 3, 5, 7 \rangle \) is different from \( \langle 5, 3, 7 \rangle \).

Let \( T = \text{seq}(T_0) \), \( t : T \) and \( t = \langle a_1, \ldots, a_n \rangle \) with \( n \in \mathbb{N} \). The following functions are defined and used in this thesis on the sequence type:

<table>
<thead>
<tr>
<th>dom</th>
<th>length</th>
<th>( T \rightarrow \mathbb{N} )</th>
<th>( \text{dom}(t) := n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>lov</td>
<td>head</td>
<td>( T \rightarrow T_0 )</td>
<td>( \text{lov}(t) := a_1 ), if ( \text{dom}(t) &gt; 0 )</td>
</tr>
<tr>
<td>loext</td>
<td>cons</td>
<td>( T \times T_0 \rightarrow T )</td>
<td>( \text{loext}(t, t_0) := \langle t_0, a_1, \ldots, a_n \rangle )</td>
</tr>
<tr>
<td>hiext</td>
<td>snoc</td>
<td>( T \times T_0 \rightarrow T )</td>
<td>( \text{hiext}(t, t_0) := \langle a_1, \ldots, a_n, t_0 \rangle )</td>
</tr>
<tr>
<td>lorem</td>
<td>tail</td>
<td>( T \rightarrow T )</td>
<td>( \text{lorem}(\langle a_1, a_2, \ldots, a_n \rangle) := \langle a_2, \ldots, a_n \rangle ), if ( \text{dom}(t) &gt; 0 )</td>
</tr>
</tbody>
</table>

The first column is the name we will use for those functions, the second is the more widely used. Our choice for the former is based on our wish not to deviate to much from the notation used in [Fóth 01] for the sake of convenience when one is studying both documents.

Furthermore we will write \( [x] \), with \( x \) of the sequence type, to denote the set of the elements from \( x \). For example, \( \{1, 2, 3\} \) = \{1, 2, 3\}. Note that because we do not allow repetition in sequences, the number of elements in \( x \) is equal to the number of elements in \( [x] \).
4.1.2 Total disjoint decomposition

As stated before we are aiming for a master-slave implementation. Therefore we are looking for a way to divide the complete input into parts, which we will call blocks, that can be dynamically distributed to the slaves. After processing those blocks the master should be able to merge them in a cost-effective way.

In this section we will only be concerned about the way we will divide the input into blocks, in other words how we will construct a total disjoint decomposition. Those blocks that form a total disjoint decomposition can be processed independently from each other and that merging them can be done cost-effective will be shown in section 4.1.3.

Let $H$ denote a set with a complete ordering $\langle \rangle$ among its elements and $S = \text{seq}(H)$. Let $X = S^k$ (a direct product with $k$ components), where $k \in \mathbb{N}$. With $x \in X$ and $r \in \mathbb{N}$ we call $x^{(1)}, x^{(2)}, \ldots, x^{(r)}$ a total disjoint decomposition of $x$ if $r$ for all different blocks ($i \neq j \in [1..r]$) and for all components $u, v \in [1..k]$ formulae (16) and (17) hold:

$$ [x_u^{(i)}] \cap [x_v^{(j)}] = \emptyset \quad (16) $$

$$ \exists i \in \text{Perm}(1, 2, \ldots, r) : \forall u \in [1..k] : x_u = x_u^{(i(1))} + + x_u^{(i(2))} + + \ldots + + x_u^{(i(r))} \quad (17) $$

Formula (16) expresses disjointness of the decomposition and formula (17) the completeness. Such a total disjoint decomposition of $x$ can be abbreviated by $\text{cdd}(x, (x^{(1)}, \ldots, x^{(r)}))$. The elements $x^{(1)}, \ldots, x^{(r)}$ are the actual blocks that will be distributed among the slaves.

Readers who have access to [Fótth 01] will notice that (16) differs from the formula presented there. For the sake of completeness we will repeat the original formula:

$$ \forall s \in \text{range}(x_u^{(i)}) : \forall t \in \text{range}(x_v^{(j)}) : x_u^{(i)}(s) \neq x_v^{(j)}(t) $$

In figure 21 one can see a possible total disjoint decomposition $(\text{cdd}(x, (x^{(1)}, x^{(2)}, x^{(3)})))$ with $x_i \in \text{seq}(\mathbb{N})$, $x = (x_1, x_2, x_3, x_4)$ and e.g. $x^{(2)} = (<< 3, 4, 5 >>, << 3, 5 >>, << 4 >>, << >>)$. With the identity-mapping for permutation $i$ in formula (16) we can construct e.g. $x_2 = x_2^{(1)} + + x_2^{(2)} + + x_2^{(3)} = << 1 >> + + << 3, 5 >> + + << 7 >>$. Note that every block includes no element that also exists in any other block (disjointness, formula (16)) and every element from $x$ and no other elements are in the decomposition (completeness, formula (17)).

![Figure 21: Total disjoint decomposition](image)

33
The 'bag-size' $N$ of $x$ is equal to the number of elements in $x$ and with the 'set-size' $M$ we denote the number of distinct elements in $x$. More formal:

$$N = \sum_{i=1}^{k} \text{dom}(x_i)$$

$$M = \#(\bigcup_{i=1}^{k} [x_i])$$

For our example in figure (21) $N = 19$ and $M = 12$. It is clear that $M$ is always at most $N$.

Suppose we have access to $p$ processors and need to process a certain input $x$. We could divide the input into exactly $p$ blocks, one block for each processor. In order to balance the work-load on the processors every block should have a bag-size of $N/p$. To determine the bag-size of a block we need to compute the union of the elements which is in fact an elementwise processable function.

The other option would be taking blocks of size $M/p$. Unfortunately this way we may get blocks with a huge difference in set-size. See for example figure 21 where we produced blocks with a balanced bag-size but with set-sizes of 23 and 7.

In order to balance the workload on every processor we will divide the input into blocks and distribute them among the processors dynamically. Every processor that is ready with processing a block will receive a new one until there are no unprocessed blocks available. One can show that this takes at most $M/p + B$, with $B$ the bag-size of the largest block, steps for $p$ processors to process $x$ [Föth 01, §5.2].

It is a task of the programmer to choose a value for $B$. A small value will result in more blocks, more costs for cutting the blocks and more communication cost distributing the blocks among the processors. A big value for $B$ could result in an unbalanced load over the processors. In section 4.4.4 on page 61 we will choose a value for $B$ suited for our implementation, the PowerXplorer and the presented input.

Our implementation will use the algorithm presented in [Föth 01, §5.1] which will produce blocks with a bag-size between $B$ and $B/8$. In this algorithm 2 sets are introduced; a set ($b$) for storing blocks with a bag-size bigger than $B$ and a set ($b$) for the other blocks. Initial there is only 1 block containing the whole input. Every step a block larger than $B$ is taken from $b$, cut in 2 and put back in the appropriate set until there are no blocks left in $b$. Where to cut a block can be found in section 4.3.5 on page 48.

### 4.1.3 Elementwise processability

We will start this section with the definition of the elementwise processable function, followed by some abstract sequential programs and examples in order to illustrate the use of the elementwise processable function. The last program presented will be the one from [Föth 01] that will be used in the rest of this thesis.

Let $X$ be the same as in section 4.1.2, $P = p(H)$ and $Y = P^l$, where $l \in \mathbb{N}$. Then we call $F : X \rightarrow Y$ an elementwise processable function iff for all $x, (x^{(1)}, \ldots, x^{(r)}) \in X$, such that $\text{card}(x, (x^{(1)}, \ldots, x^{(r)}))$ and for all $i \in [1..l]$ formulæ (18) and (19) hold:

$$F_i(x^{(1)}) \cup F_i(x^{(2)}) \cup \cdots \cup F_i(x^{(r)}) = F_i(x) \quad (18)$$

$$F_i(x^{(1)}) \cap F_i(x^{(2)}) \cap \cdots \cap F_i(x^{(r)}) = \emptyset \quad (19)$$

Formula (18) shows us that we can process blocks separately and formula (19) expresses that an elementwise processable function should be one-to-one\(^{15}\).

\(^{15}\)When one weakens (19) into $F(\emptyset) = \emptyset$ one gets the definition of the weak elementwise processing. The class of
In figure 22 we made a different total disjoint decomposition \( cdd(x, (x^{(x^1)}, \ldots, x^{(x^n)})) \) from the same data as used in figure 21 on page 33. For simplicity reasons we left out the elements ranging from \( \{7\} \) to \( \{13\} \). Note that this decomposition is constructed in such a way that the set-size of every block is exactly 1. With \( F \) an elementwise processable function we can calculate \( F_i(x) \) the following way:

\[
F_i(x) = F_i(x^{(x^1)}) \cup \cdots \cup F_i(x^{(x^n)}) = F_i(\{1\}, \{1\}, \emptyset, \{1\}) \cup \cdots \cup F_i(\{5\}, \{5\}, \emptyset, \emptyset)
\]

The next 4 programs depend on the fact that the set-size of the blocks is exactly 1 and will use (18) to calculate \( F \). We will introduce function \( elem() \) to select an arbitrary element from a sequence. Note that one can replace a sequence \( x \) in the algorithms by \( [x] \) in order to get the algorithm of the elementwise processing on sets instead of sequences\(^{16}\).

1-1 elementwise processing

In the 1-1 elementwise processing is the dimension of \( X \) and \( Y \) both equal to 1. This means that the initial value of variable \( x \) will be \( x_1 \). Or, like in our example from figure 22, \( x = [1, 2, 3, 4, 5] \).

Now let us look at the declare section of the program. The type declarations of the variables happens in (20). Invariant (21) shows that the elements we still have to process \( (x) \) and the elements we already processed \( (x') \) should form a total disjoint decomposition of the original input \( (x) \). The results of processed elements will be stored in \( y \) (22) and fixed point condition (23) is reached when all elements from \( x \) are processed.

A closer look at the algorithm reveals that boolean variable \( ch \) selects one of the 2 assignment statements in an alternating way. Note that an initialization of \( ch \) with \( true \) leads to the execution of the second assignment statement with an uninitialized \( e \). The algorithm itself is straight forward. An element \( e \) is chosen from \( x \), processed \( (F(e)) \) and added to \( y \) \( (y := y \cup F(e)) \) until all elements from \( x \) are processed.

An example of a 1-1 elementwise processable function is the identity function \( i \). To verify this one has to show that (18) and (19) hold. We will not do this for this trivial case. Later on one can find an example of such a proof.

\(^{16}x \neq \emptyset \rightarrow [x] \neq \emptyset \rightarrow [x] \neq \emptyset\)
1-1 elementwise processing

\[ \text{var: } x, x': X, y: Y, e: H, ch: BOOL \]

\[ \text{inv: } \quad \text{cdd}(x', (x, x' - x)) \]

\[ \text{inv: } \quad y = F(x' - x) \]

\[ FP \Rightarrow \quad x = \ll \gg \]

\[ s_0 : \quad y, ch := \emptyset, \text{false} \]
\[ \{ \begin{array}{l}
\quad e, ch := \text{lorem}(x), \text{true} \quad \text{if } x \neq \ll \gg \land \neg ch\\
\quad (y, ch := y \cup F(e), \text{false} \mid x := \text{lorem}(x)) \quad \text{if} \ ch
\end{array} \} \]

Now we will give an example of a function \( G \) that is not 1-1 elementwise processable. Let \( a_0, a_1, \ldots, a_k \) be integers, * the well known multiplication operator and \( G \) be defined as follows:

\[ G(\ll a_0 \gg) = \{ a_0 * a_0 \} \]

\[ G(\ll a_0, a_1, \ldots, a_k \gg) = G(\ll a_0 \gg) \cup G(\ll a_1, \ldots, a_k \gg) \]

then \( G \) is not a 1-1 processable function. Let \( a = \ll -1,1,2 \gg \) with \( \text{cdd}(a, (\ll -1 \gg, \ll 1,2 \gg) \)
then (19) does not hold:

\[ G(\ll -1 \gg) \cap G(\ll 1,2 \gg) = \{1\} \cap \{1,4\} \]
\[ = \{1\} \]
\[ \neq \emptyset \]

Note that in the definition of (18) and (19) it is required that they hold for every possible total disjoint decomposition. When we would have taken \( \text{cdd}(a, (\ll -1 \gg, \ll 1 \gg, \ll 2 \gg) \) we would falsely have concluded that (19) holds.

2-1 elementwise processing

In the case of a 2-1 elementwise processable function, \( F \) takes 2 arguments and the input \( x \) contains 2 sequences. The algorithm contains of 4 assignment statements; 1 for selecting an arbitrary element from \( x \), the other 3 for processing this element. For the more general k-1 elementwise processing case there are \( 2^k \) assignment statements needed. Note that we do not need to process \( F(0,0) \).

One of the most used examples of a 2-1 elementwise processable function is the union of sets. To proof that this function is elementwise processable one should show that (18) and (19) hold for every possible total disjoint decomposition. This proof can be found in section A.3 on page 74.

Two other examples of 2-1 elementwise processable functions are the intersection and the difference of 2 sets. We will use the intersection of 2 sets to show why disjointness of a total decomposition is required for elementwise processing.
2-1 elementwise processing

\[ \text{var: } x, x': X, y : Y, e : H, ch : BOOL \]
\[ \text{inv: } \quad \text{cdd}(x', (x, x' - x)) \]
\[ \text{inv: } \quad y = F(x' - x) \]
\[ FP \Rightarrow \quad x = (\ll, \ll) \]

\( s_0 : y, ch := \emptyset, false \)
\{ \)
\[ (e, ch := \min\{\text{loav}(x_1), \text{loav}(x_2)\}, true), \text{if } x \neq (\ll, \ll) \land \neg ch \]
\[ (y, ch := (y \cup F(\{e\}, \emptyset), false \parallel x_1 := \text{lorem}(x_1)), \text{if } e = \text{loval}(x_1) \land e \neq \text{loav}(x_2) \land ch \]
\[ (y, ch := (y \cup F(\emptyset, \{e\}), false \parallel x_2 := \text{lorem}(x_2) e), \text{if } e \neq (x_1) \land e = \text{loav}(x_2) \land ch \]
\[ (y, ch := (y \cup F(\{e\}, \{e\}), false \parallel x_1 := \text{lorem}(x_1) \parallel x_2 := \text{lorem}(x_2)), \text{if } e = \text{loval}(x_1) \land e = \text{loav}(x_2) \land ch \]
\}\n
Let \( A, B \) be sequences and \( F(A, B) = [A] \cap [B] \), the conjunction of sets, and let \( a = (\ll, \ll, \ll, \ll) \). Then there is a total decomposition \( cd(a, (\ll, 1, 2 \ll), (\ll, \ll, \ll)) \). Suppose this is a valid cdd and apply the rules for the elementwise processing:

\[ F(\ll, 1, 2 \ll, \ll, \ll) = F(\ll, 1, 2 \ll, \ll, \ll) \cup F(\ll, \ll, \ll, \ll) \]
\[ = (\{1, 2\} \cap \emptyset) \cup (\emptyset \cap \{1, 2\}) \]
\[ = \emptyset \cup \emptyset \]
\[ = \emptyset \]

In case we would have taken the total decomposition \( cd(a, (\ll, 1, 2 \ll, \ll, \ll), (\ll, \ll, \ll, 2 \ll)) \) the outcome would be \( \{1\} \) which is different from our previous outcome. From this small example it is obvious that when dropping the requirement of disjointness the outcome of an elementwise processable function could depend on the way the total decomposition was constructed.

1-2 elementwise processing

1-2 elementwise processing

\[ \text{var: } x, x' : X, y : Y, e : H, ch : BOOL \]
\[ \text{inv: } \quad \text{cdd}(x', (x, x' - x)) \]
\[ \text{inv: } \quad y = F(x' - x) \]
\[ FP \Rightarrow \quad x = \ll \ll \]

\( s_0 : y, ch := (\emptyset, \emptyset), false \)
\{ \)
\[ e, ch := \text{loav}(x), true), \text{if } x \neq \ll \land \neg ch \]
\[ (y_1, y_2, ch := y_1 \cup F_1(e), y_2 \cup F_2(e), false \parallel x := \text{lorem}(e)), \text{if } ch \]
\}\n
37
k-l elementwise processing

\[
\begin{align*}
\text{var} & : x, x' : X, y : Y, e : H, ch : BO\text{OL} \\
\text{inv} & : \quad \text{odd}(x', (x, x' - x)) \\
\text{inv} & : \quad y = F(x' - x) \\
FP & \Rightarrow \quad x = (\ll, \ldots, \ll) \
\end{align*}
\]

(24) (25) (26) (27)

\[s_0 : y, ch := (\emptyset, \ldots, \emptyset), \ false\]

\[
\begin{cases}
  & e, ch := \min \{ \text{loev}(x_i) \mid i = 1 \ldots k \land x_i \neq \ll \}, \ true \\
  \text{if } x \neq (\ll, \ldots, \ll) \land \lnot \ ch \\
  & (y, ch := (y_i \cup F_i(sl(x, e)))_{i=1}^k, false \ || \\
  \ll_{i=1}^k(x_i := \text{lorem}(x_i), \text{if } x_i \neq \ll \land \text{loev}(x_i) = e) \\
  \text{if } ch
\end{cases}
\]

where \( \forall j = 1 \ldots k \):

\[sl(x, e)(j) := \begin{cases} \\
  \ll e \gg, \text{if } x_j \neq \ll \land \text{loev}(x_j) = e \\
  \ll \gg, \text{otherwise}
\end{cases}\]

An example of a 1-2 elementwise processable function is the function \( F \) that makes 2 copies of the input. \( F \) is defined as follows:

\[
\begin{align*}
F_1(\ll e_1, \ldots, e_n \gg) &= \{e_1, \ldots, e_n\} \\
F_2(\ll e_1, \ldots, e_n \gg) &= \{e_1, \ldots, e_n\}
\end{align*}
\]

k-l elementwise processing

The algorithm for the k-l elementwise processing comes from [Fóth 01] and differs a bit in notation from the previous ones. Selection of an element \( e \) is not arbitrary anymore, but happens in such a way that \( e \) is the smallest element in \( x \). Remember that we are dealing with ordered sequences, so one only has to inspect the first element in every sequence to determine the smallest. That this can be done by only inspecting the first element of every sequence is because [Fóth 01] requires that the input consists of ordered sequences.

4.2 The algorithm in an informal way

By using a small example we would like to present the parallel algorithm in an informal way. In section 4.3 on page 40 one can find the formal representation of the algorithm.

Recall that the parallel algorithm divides the input into blocks which can be processed independent from each other. Then a master process will dynamic distribute the blocks among the available slaves. Those slaves process the blocks with a sequential algorithm and will return the obtained results to the master after the last block has been processed.

Suppose our input has a dimension of 3. This means the input contains 3 sequences with each sequence hosted on one of the 3 slaves. In this example slave 1 hosts \( \ll 1, 2, 3, 4, 5, 6 \gg \), slave 2 hosts \( \ll 2, 4, 6, 8, 10, 12 \gg \) and slave 3 hosts \( \ll 3, 6, 9, 12, 15, 18 \gg \). In section 4.4.1 on page 58 we call this particular type of input \textit{STEP}.
The first task of the slaves is to inform the master about the sequence they host. This is done by sending the tuple \((s, d, f)\) to the master with \(s\) the starting index of the sequence (in our example always 0), \(d\) the length of the sequence and \(f\) the median value. For slave 1 this tuple, called blockpart, is equal to \((0, 6, 3)\). Three of those blockparts describe a block (blockdescriptor). The blockdescriptor in our example after the slaves communicated their blockparts is \(((0, 6, 3), (0, 6, 6), (0, 6, 9))\) and describes the complete input.

If the master receives a block with a bag-size larger than a certain constant value \(B\) it has to cut that block in half. Those new generated blocks should be cut in half also when their bag-size exceeds \(B\). This cutting of blocks will continue until no blocks with a bag-size larger than \(B\) is left.

The master has no information about the elements in a block and therefore the actual cutting has to be done by the slaves. First the master calculates a so called cut-value and hopes this value won’t be far from the median of the complete block. Then the master will send this value together with the starting index and the length of the blockparts to the slaves.

The slaves will use this cut-value to cut their blockpart in such a way that elements at most the cut-value will form a new lower-blockpart and elements greater an upper-blockpart. Those new blockparts should be send to the master which then can construct 2 new blockdescriptors. Figure 23 represents our example where the first cut is made. The cut-value is 6 and the new blockdescriptors the master will produce are \(((0, 6, 3), (0, 3, 4), (0, 2, 3))\) and \(((0, 0, 0), (3, 3, 10), (2, 4, 12))\).

![Figure 23: Cutting a block](image)

Suppose that in our example \(B = 10\), then there is still 1 block left to cut because the first constructed lower-block has a bag-size of 11. The cut-value for this block will be 3 and the master has to send \((0, 6, 3)\) to slave 1, \((0, 3, 3)\) to slave 2 and \((0, 2, 3)\) to slave 3. After performing the cuts the new blockdescriptors will be \(((0, 3, 1), (0, 1, 2), (0, 1, 3))\) and \(((3, 3, 4), (1, 2, 4), (1, 1, 6))\).

After cutting the blocks they should be processed by the slaves. The master sends the blockdescriptors to the slaves that are idle. Every slave requests data it needs from the other slaves, waits for this data to arrive and performs the elementwise processing on the block. When ready processing a block the slave notifies the master that it is ready to receive a new block to process. After all blocks have been processed, the slaves send their results to the master.

In figure 24 one can see what happens when the master sends block \(((3, 3, 4), (1, 2, 4), (1, 1, 6))\) to slave 1 for elementwise processing. Slave 1 first requests the data described by \((1, 2)\) from slave 2 and \((1, 1)\) from slave 3. Slave 2 responds on this by sending the sequence \(\langle 4, 6 \rangle\) and slave 3 by sending \(\langle 6 \rangle\). Note that it is likely that slave 2 and 3 are also busy processing a block at the same time.
4.3 Transformation of the algorithm

In section 4.3.1 we will introduce 2 possible topology's and decide which one we will use for our concrete program. The next section will cover the types needed in the program and in section 4.3.4 till 4.3.7 the actual program is transformed.

In [Fóth 01] one can find the derivation of the complete program. We will repeat here just the most important parts that are needed to understand the concept of the program and the transformation. Instead of presenting the complete abstract code in the beginning, like is done in the case of the associative function, we will use the same partition of the program like is done in [Fóth 01]. Although this approach works fine in this case, it is always a good idea to first read and understand the derivation completely before proceeding with the transformation.

With our implementation we will restrict ourselves to the situation where the number of slaves is equal to the dimension of the input \( x \). We need to inform the reader that changing the implementation to a situation where an other amount of slaves is used is not trivial. This is caused mainly because we will use this property in our communication model.

4.3.1 Topology

As stated before, we are aiming for a master-slave implementation of our program. Once the master task is started it will spawn a number of slaves needed for the calculation. In most master-slave implementations the slaves only communicate with their master, but in our case the slaves will have to communicate among each other also.

The left side of figure 25 (A) represents a topology where slaves directly communicate among each other, in the right side (B) an extra task is spawn to handle the communications with other slaves via their extra communication-task. Data needed by other slaves is stored in this extra communication-task so that it can serve requests without the need to contact it’s parent first.

Before deciding which topology to implement we will make a small overview of the possible consequences:

**Communication:** It is clear that topology (B) causes more communications within the PowerExplorer than topology (A). Suppose we have in total \( n+1 \) slaves, then in situation (A) there will be \( n \) requests for data and \( n \) answers returning the requested data, in total \( 2n \) communications. Situation (B) needs \( n+1 \) extra communications; the request from the slave to the communication-task and \( n \) messages send by the communication-task to the slave containing the requested data.
The total number of communications in situation (B) can be reduced from $3n + 1$ to $2n + 2$ when the communication-task first collects all the data before sending it to the slave in one message. Although the number of communications can be reduced this way, the total amount of data being transported is still doubled compared to topology (A).

Note that when the slave and the communication-slave are hosted on the same processor, those extra communications will be intra-processor communications and will not occupy the external channels.

**Memory:** It is obvious that topology (B) uses more memory because of the extra program code that needs to be stored.

When e.g. a communication-task receives data from other communication-tasks this data is stored twice in memory. After the communication-task passed this data on to the slave there are 3 copies in memory. The copy stored on the communication-task can be deleted now, but we have no guarantee that this is done before the slave stored it’s copy. Therefore extra memory, at the size of a block, is needed in topology (B) compared to topology (A).

**Idle time:** In both topology’s there is idle time when a slave has to wait for the needed data to arrive. A disadvantage of topology (A) is that a slave is not always able to immediately answer a request for data which results in more idle time.

**Our decision to implement topology (A) is made with the following in mind:**

- We would like to measure the performance of the system for large input sequences. Therefore memory usage should be as limited as possible.
- Idle time is a topic we did not cover so far. It could be interesting to see how much idle time influences performance and what we can do about it.

### 4.3.2 Datatypes

A disadvantage of our approach is that we have no clear view which implementation of our datatypes will be the most efficient. Therefore the choice for abstract datatypes is obvious. One can change the implementation of abstract datatypes without the need to change the code which
uses these datatypes. During the process of transforming the abstract program, the implementation of the abstract datatypes indeed changed a few times. Of course not all datatypes are implemented as abstract ones. We restricted ourselves to the most complicated ones: $\text{SEQH}$, $\text{BVS}$, $X$ and $Y$.

In the remaining part of this section we will describe all the (abstract) datatypes. This is done by its name, a short description of the type, the functions for this type and the implementation of the type. The complete implementation of the datatypes can be found in section C.3.2 on page C.3.2. Recall that $K$ is fixed and expresses the dimension of the input.

**H** denotes a set with a complete ordering $< \! < \!$ among its elements. Elements to be processed will be of type $H$.

```c
int pvm.pkH(H *h, int nitems, int stride)
int pvm.upkH(H *h, int nitems, int stride)
```

The functions `pvm.pkH()` and `pvm.upkH()` will be used for communications of type $H$. They will behave the same as the standard `pvm.pk*()` and `pvm.upk*()` function in PVM. See also section 1.3.1 on page 6.

```c
#define MAX_H 2147483647
typedef int H;
```

In this implementation we chose `int` as type for $H$. Integers fulfill all the requirements and are a standard type in C. In order to use an other type, one has to write functions for the ordering of the new type (overloading of e.g. $'<'$ and $'>'$).

The constant `MAX_H` is used the same way as the well known `MAX_INT` and the value depends on the used system.

**SEQH** is of type seq($H$). Note that in [Fäh 01] a capital $S$ is used to denote this type. We choose an other identifier because $s$ is also used in the description of a blockpart.

```c
void SEQH.init(SEQH **seqH, int size)
void SEQH.free(SEQH **seqH)
void SEQH.fill(SEQH **seqH)
int SEQH.dom(SEQH seqH)
  ret : dom(seqH)
H SEQH.val(SEQH seqH, int i)
  ret : seqH[i]
H SEQH.low(SEQH seqH)
  ret : low(seqH)
int SEQH.search(SEQH seqH, int s, int d, H h)
  pre : d > 0
  pre : seqH[s - 1] = -∞
  pre : seqH[s + d] = ∞
  ret : seqH[ret] ≤ k < seqH[ret + 1]
int pvm.pkSEQH(SEQH **seqH, int startPosition, int size)
int pvm.upkSEQH(SEQH **seqH)
```

With `SEQH.init()` one can initialize a variable of the `SEQH` type. The argument `size` is an upper bound for the maximum length of the sequence. `SEQH.free()` un-initializes a variable of type `SEQH`.

Function `SEQH.fill()` is only used to provide initial values. The implementation could be changed in such a way that the initial values will be read from a data storage or even be communicated by an other task. `SEQH.dom()` returns the number of elements that are in
the sequence and function \texttt{SEQH.val()} returns the value of the element with the matching index.

\texttt{SEQH.loc()} returns the first element from sequence \texttt{seqH}. Because sequences of type \texttt{H} are ordered this element is also the smallest element in the sequence.

The function \texttt{SEQH.search()} searches a sequence for the element \texttt{h}, starting on position \texttt{s} up to position \texttt{s + d}. As a precondition for \texttt{SEQH.search()} we have that the block we need to search in is not empty and that the first element is at most \texttt{h}.

The functions \texttt{pvm.pkSEQH()} and \texttt{pvm.upkSEQH()} will be used for communications of type \texttt{SEQH}. When receiving a sequence packed with \texttt{pvm.pkSEQH()}, one does not know the size of the sequence. Therefore \texttt{pvm.upkSEQH()} takes an uninitialized variable of type \texttt{seqH} as argument.

```c
typedef struct Tseqh {
    int size;
    int lob;
    H *seqh;
} SEQH;
```

We implement \texttt{SEQH} with an array of fixed length which has to be known before calling \texttt{SEQH.init()}, the initializing function for the \texttt{SEQH} type. The main reason is that \texttt{pvm.pkSEQH()} and \texttt{pvm.upkSEQH()} can be implemented more efficient. Note that a communication of this type occurs rather frequent.

We can use this implementation because variables of this type do not increase in size, only elements at the head of the sequence will be removed. Integer \texttt{size} is used to store the size of the array and integer \texttt{lob} is used to point to the first item of the sequence. The size of a sequence (\texttt{SEQH.dom}) is equal to \texttt{size} – \texttt{lob}.

\texttt{BLOCKPART = (s:int, d:N, f:H)} is used to store blockparts. A blockpart is denoted by the index of the first element (\texttt{s}), the size of the blockpart (\texttt{d}) and the median value (\texttt{f}) of the blockpart. In \cite{Föth 01} this type is called \texttt{BLOCK}.

```c
int pvm.pkBLOCKPART( BLOCKPART *blockpart, int nitems, int stride)
int pvm.upkBLOCKPART( BLOCKPART *blockpart, int nitems, int stride)
```

The functions \texttt{pvm.pkBLOCKPART()} and \texttt{pvm.upkBLOCKPART()} will be used for communications of type \texttt{SEQH}. When receiving a sequence

```c
typedef struct TBlockpart {
    int s;
    int d;
    H f;
} BLOCKPART;
```

An alternative implementation could be \texttt{typedef int BLOCKPART[3]} which is more efficient to communicate than an structure\textsuperscript{17}. Although in our implementation \texttt{s}, \texttt{d} and \texttt{f} are of the same type, we decided not to use this alternative implementation because of readability and the fact that our program needs to be changed when the type \texttt{H} changes.

\texttt{BV = vector \{1 \ldots K\} of BLOCKPART, descriptor of a whole block (blockdescriptor). Because BV describes a block variables of this type will sometimes be called a block.}

\textsuperscript{17}In PVM, all elements of a structure have to be packed separately.
typedef BLOCKPART BV[K];

A more efficient implementation in terms of communication costs would be the following:

typedef struct Tbv
{
    int s[K];
    int d[K];
    H f[K];
} BV;

We chose not to use this implementation because of readability. Analogous to the previous BLOCKPART implementation we could also use the alternative typedef int BV[3*K] implementation.

**BVS** = seq(BV), the description of a total disjoint decomposition

void BVS.init(BVS *bvs)
void BVS.free(BVS *bvs)
void BVS.insert(BVS bvs, BV bv)
  pre : dom(bvs)  \neq  0
  post : bvs =lov(bvs)
void BVS.append(BVS bvs, BV bv)
  post : bvs = bvs' \Rightarrow
void BVS.rename(BVS bvs)
  pre : bvs' =« bvs_{1}', \ldots, bvs_{n}' \rangle
  post : bvs = « bvs_{1}', \ldots, bvs_{dom(bvs')}' \rangle
void BVS.dom(BVS bvs)

The functions *BVS.init()* and *BVS.free()* initialize and destroy a sequence of type **BVS**. *BVS.insert()* provides the first element of the sequence, *BVS.append()* adds an element to the beginning of the sequence and *BVS.rename()* removes the first element of the sequence. With function *BVS.dom()* one can determine the number of elements in the sequence.

typedef struct TBvs
{
    BV seq;
    int dom;
    struct TBvs * next;
} TBvs;

typedef TBvs *TBVS;

![Figure 26: type BVS](image)

**BVS** is implemented as a linked list with integer **dom** indicating the number of elements in the sequence from that point until the last element. A schematic representation of this type can be found in figure 26.
BVSP = BVS^K, for storing the actually processed block (as a sequence of one length) on each processor.

typedef BVS BVSP[K];

Although we declared this type, we do not use it in the implementation. The reason why it is mentioned here is that we will encounter this type in some of the presented abstract algorithms.

X = SEQH^K, a direct product with K components. Note that in [Fóth 01] this is denoted as 3^K.

void X.init(X *x)
void X.free(X *x)
bool X.dom(X x)
ret : \sum_{i=1}^{K} dom(x_i)
H X.lov(X x)
pre : x \neq \emptyset
ret : \{ lov(x_i) \mid i = 1..K \land x_i \neq \emptyset \}
bool X.lorem( X *x, int i, H h)
pre : x' = x, h' = h, 0 <= i < k
pre : \forall (g : g \in x : h <= g)
ret : true \implies h + x_i = x_i
ret : false \implies x = x' \land h \notin [x_i]
void X.moveSeqH(X *x, SEQH *seqH, int i)
void X.copySeqH(X *x, SEQH seqH, int i, int startPosition, int size)

The functions X.init and X.free are used for allocating and releasing memory for a variable of type X. Function X.dom() returns the number of elements in x, X.lov() determines the smallest element and X.lorem() checks if this smallest element is an element of x_i (x.rows[i]) and removes it when possible.

For moving and copying sequences into a variable of type X we designed X.moveSeqH() and X.copySeqH(). The first will move an entire sequence to x_i, the latter copies a segment to x_i. In both cases x_i should be empty.

typedef struct Tx
  {
    int bagSize;
    SEQH rows[K];
  } X;

The type definition of X is straight forward, except the extra integer named size. This integer is used to denote the total number of elements in X.

Y = vector [1..L] of p(H)

void Y.init(Y *y)
void Y.free(Y *y)
void Y.addH(Y *y, int i, H h)
pre : 0 <= i < L \land h \notin y_i
post : y_i = y_i \cup h
int pvm.pkY(Y *y)
int pvm.unpkY_and_add(Y *y)

The functions Y.init and Y.free are used for allocating and releasing memory for a variable of type Y. To add an element of type H to y_i one can use the Y.addH() function.
One can pack an entire variable of type $Y$ with \texttt{pym\_pkY()}. Unpacking is done with \texttt{pym\_upkY\_andAdd()} which merges the received variable of type $Y$ with the variable given as an argument.

\begin{verbatim}
typedef struct Tp
  { 
    H h;
    struct Tp * next;
  } P;
typedef struct Tpl 
  { 
    int size;
    P *set;
  } PL;
typedef PL Y[L];
\end{verbatim}

Type $Y$ is implemented as a linked list. The first element of such a list contains an integer named \texttt{size} to denote the number of elements in the list.

\[
\text{YV} = (\text{vector } [1 \ldots L] \text{ of } p(H) )^K, \text{ for collecting the partial results in.}
\]

\[
\text{typedef } Y \text{ YV[K];}
\]

Although we declared this type, we do not use it in the implementation. The reason why it is mentioned here is that we will encounter it in some of the presented abstract algorithms.

### 4.3.3 General considerations

In the previous sections we explained the theory behind sequences, total disjoint decompositions and elementwise processable functions. Section 4.3.1 was dedicated to our choice of a topology followed by the description of the datatypes we designed in order to make it more easy to implement the abstract program.

Instead of one complete abstract program, small separate programs are presented in [Fóth 01]. Therefore we will not translate the complete program at once, but part by part. In order to 'glue' those parts together we sometimes needed to write some extra code. Because this code was in all cases trivial we will not mention it. The same counts for starting the slaves, initializing and de-initializing\footnote{e.g. releasing allocated memory} them.

The slaves will use the function \texttt{nextMessage()} to determine the message tag of the next incoming message. Depending on the message tag, \texttt{nextMessage()} will return the message tag or receive the message itself, react on it when needed and wait for the following message. In the following sections we will introduce the basic form of this function and extend the functionality whenever needed.

Our main reason for introducing \texttt{nextMessage()} was simplifying our task to avoid deadlock. We believe we succeeded in this, but generated the same time an idle time problem as we expected in section 4.3.1. An unexpected advantage is that switching to topology B will be a relative easy job.

Rests us to mention that every slave is hosting one sequence from the input $x^{19}$. To be exact, slave $i$ is hosting sequence $x_i$.

\footnote{Remember we have exactly $K$ slaves, with $K$ the dimension of $X$.}
4.3.4 Total disjoint decomposition: Initializing part

When we take a close look at abstract program 6.1 we see that integer \( j \) will select the first assignment statement \( K \) times, followed by a continuous selection of the second assignment statement.

**Abstract 6.1**

\[
\begin{align*}
\text{var :} & \quad x : X \quad N, B, j : \mathbb{N} \quad bb, blk : BVS \\
\text{inv :} & \quad j \in [0..K] \land N = \sum_{i=1}^{j} \text{dom}(x_i) \\
\text{inv :} & \quad \forall (i : i \in [1..j]) : \text{blk}[i] = (\text{low}(x_i), \text{dom}(x_i), x_i(\text{low}(x_i) + \frac{\text{dom}(x_i) - 1}{2})) \\
\text{FP} \Rightarrow & \quad j = K \land B = \text{opt}(N) \land \text{CDD}(x', bb) \land bb = \ll \text{blk} \gg
\end{align*}
\]

\( s_0 : \ j, N := 0, 0 \)

\[
\begin{align*}
\{} & \quad N, j, \text{blk}[j + 1] := \quad \\
\{} & \quad N + \text{dom}(x_{j+1}), j + 1, \quad \\
\{} & \quad (\text{low}(x_{j+1}), \text{dom}(x_{j+1}), x_{j+1}(\text{low}(x_{j+1}) + \frac{\text{dom}(x_{j+1}) - 1}{2})) \quad \\
\{} & \quad \textbf{if} \ j < K \quad \\
\{} & \quad B, bb := \text{opt}(N), \ll \text{blk} \gg \quad \textbf{if} \ j = K \quad \\
\{}
\end{align*}
\]

Because the master needs to cut the blocks, variable \( B \), \( N \) and \( bb \) will be managed by the master. Variable \( x_i \), denoted by \( \text{seq}H \) in the translation, is managed by slave \( i \).

For the master a function \( \text{ReadBlockVector}() \) is designed which will read \( K \) blockparts, which form a blockdescriptor, from the \( K \) slaves, stores them in variable \( \text{blk} \) and returns the bag-size of the complete block. \( \text{ReadBlockVector}() \) processes the blockparts in the order it receives them from the slaves and not, like in abstract 6.1, in a specific order. Note that this order is caused by variable \( j \).

Instead of putting \( \ll \text{blk} \gg \) immediately into \( bb \) like is done in the abstract program, we check if we need to cut this block. If this is not the case, we put it in \( b \), a sequence of blocks with a set-size \( \leq B \) that do not need to be cut anymore. Of course we let the program end once the fixed point is reached.

**Master: 6.1**

\[
\begin{align*}
\text{int} & \quad B, N; \\
\text{BV} & \quad \text{blk}; \\
\text{BVS} & \quad bb, b; \\
N & = \text{ReadBlockVector}(\text{blk, MSG,BLOCKPART}) ; \\
B & = \text{Opt}(N) ; \\
/ * \text{Do we need to cut this block?} \ */ \\
\text{if} & \quad (N > B) \quad \text{BVS}.\text{joext}(\&bb, \text{blk}) ; \quad \text{else} \quad \text{BVS}.\text{joext}(\&b, \text{blk});
\end{align*}
\]

The implementation of the slave is trivial. All we need to do is constructing a blockpart which represents our part of the input and send this to the master. Note that we replaced \( \text{low}(x_i) \) by 0 and could do the same for \( \text{blockpart.s} \) in the calculation of \( \text{blockpart.f} \).
4.3.5 Total disjoint decomposition: Generating blocks

Abstract program 6.2 takes the first block from bb (lov(bb)), determines the position where this block needs to be cut (CutValue()), cuts this block in two parts (Cut()) and adds the new created blocks to the appropriate sequences (Update()) until there are no blocks left in bb. as a result of this, there will be no blocks left with a set-size larger than B.

Abstract 6.2

\[
\begin{align*}
\text{var} & : \quad x : X \quad h : H \quad b, bb : BVS \quad ub, lb : BV \\
\text{inv} & : \quad CDD(x', b \circ bb) \\
FP & \Rightarrow \quad bb = \text{nil}
\end{align*}
\]

\[\begin{align*}
\text{while } \text{dom}(bb) \neq 0 \text{ loop} \\
& \quad h := \text{CutValue}(lov(bb)) \\
& \quad ub, lb := \text{Cut}(x, lov(bb), h) \\
& \quad bb, b := \text{Update}(bb, b, ub, lb)
\end{align*}\]

The algorithm for \text{CutValue()} can be found in [Főth 01, §5.1.1] and is an easy to implement sequential algorithm. In abstract 6.3 the algorithm for \text{Cut()} is given. \text{Update()} is an abbreviation of the following code:

\[
\text{Update}(bb, b, ub, lb) = \begin{cases} 
\text{loext(loext(lorem(bb), ub), lb), b} & \text{if bagsize(ub) > B \land bagsize(lb) > B} \\
\text{loext(lorem(bb), ub), loext(b, lb)} & \text{if bagsize(ub) > B \land bagsize(lb) \leq B} \\
\text{loext(lorem(bb), lb), loext(b, ub)} & \text{if bagsize(ub) \leq B \land bagsize(lb) > B} \\
\text{lorem(bb), loext(loext(b, lb), ub)} & \text{if bagsize(ub) \leq B \land bagsize(lb) \leq B}
\end{cases}
\]

Master: 6.2

int B, h, bagSizeLb, bagSizeUb, tids[K+1];
BV blk, lb, ub;
BVS bb;

while ( BVS.dom( bb ) != 0 )
{
    /* CutValue */
    BVS lov( bb, blk);
h = CutValue(blk);

/* Cut */
SendCutInfo( blk, h, &tids[1]);
bagSizeLb = ReadBlockVector( lb, MSGLOWER_BLK);
bagSizeUb = ReadBlockVector( ub, MSGUPPER_BLK);

/* Update */
BVSelJeremy( &kbb);
if (bagSizeLb>B) BVSelext( &kbb, lb); else BVSelext( &kbb, lb);
if (bagSizeUb>B) BVSelext( &kbb, ub); else BVSelext( &kbb, ub);
}

The actual cutting of a block has to be done on the slaves because there the blockparts are stored. First, every slave has to determine the position where to cut his blockpart and then perform the cut and construct the new parts. Abstract 6.3 describes this algorithm.

Abstract 6.3

\[
\text{var : } x : X \quad h : H \quad ub, lb, bv : BV \quad v : \mathbb{N}
\]

\[
\begin{align*}
\{ &\sum_{i=1}^{k} v_i := \text{Search}(x_i(bv[i].s \ldots bv[i].s + bv[i].d - 1), h) \\
&\sum_{i=1}^{k} lb[i], ub[i] := \\
&\quad (bv[i].s, v_i - bv[i].s + 1, x_i(bv[i].s + v_i)/2), \\
&\quad (v_i + 1, bv[i].s + bv[i].d - v_i - 1, x_i((v_i + 1 + bv[i].s + bv[i].d)/2))
\}
\]

Because the slaves do not know when the master is finished cutting blocks, viz. \( bb \) is empty, they need to be warned. They will be notified that this is the case when a message arrives with an other tag as \( MSG.CUT.BLK \). For now it is sufficient to know that \( nextMessage() \) only returns the message tag from the next arrived message.

The first task of the slaves is to unpack the arrived message. This message contains the blockpart to cut and the value needed to find the position where to cut that blockpart. For efficiency reasons we will use a variable of type \( BLOCKPART \) to store this information with the mean-value position (blockpart.f) used to store the cut-value.

Before a blockpart can be cut one has to find the actual position where to cut a block. This position, stored in \( j \), can be found by using a binary search. More about the binary search, including a derivation, can be found in [Kald 01, §6.1]. In case an empty blockpart is received\(^{20}\) \( j \) will be given the value \(-1\). With \( j = -1 \) both guards evaluate to false\(^{21}\) and two empty blockparts will be generated.

Slave: 6.3

\begin{verbatim}
int j, myId, myParent;
BLOCKPART blockpart, emptyBlockpart = { 0, 0, 0};
SEQH seqH;

while ( nextMessage(seqH, myId) == MSG.CUT.BLK )
{
pvm_upkB \( BLOCKPART( \& \) blockpart, 1, 1);
\end{verbatim}

\(^{20}\) An empty blockpart does not mean the whole block is empty. Figure 21 on page 33 contains a block called \( x^{(0)} \) which contains an empty blockpart \( (x^{(0)}) \).

\(^{21}\) This is only the case because we let the starting index of an empty blockpart \( (blockpart.s) \) be equal to 0.
if (blockpart.d > 0)
    j = SEQH_search(seqH, blockpart.s, blockpart.d, blockpart.f);
else
    j = -1;

/* Construct lowerBlockpart */
if ( (j - blockpart.s + 1) > 0 )
{
    newBlockpart.s = blockpart.s;
    newBlockpart.d = j - blockpart.s + 1;
    newBlockpart.f = SEQH_val(seqH, (blockpart.s + j + 1)/2);
    SendBlockpart(myId, myParent, MSGLOWER_BLK, newBlockpart);
}
else
    /* lowerBlockpart is empty */
    SendBlockpart(myId, myParent, MSGLOWER_BLK, emptyBlockpart);

/* Construct upperBlockpart */
if ( (j + 1) > 0 )
{
    newBlockpart.s = j + 1;
    newBlockpart.d = blockpart.s + blockpart.d - j - 1;
    newBlockpart.f = SEQH_val(seqH, (j + blockpart.s + blockpart.d + 1) / 2);
    SendBlockpart(myId, myParent, MSGUPPER_BLK, newBlockpart);
}
else
    /* upperBlockpart is empty */
    SendBlockpart(myId, myParent, MSGUPPER_BLK, emptyBlockpart);

Now let us take a look at function nextMessage(). With the static integer nrblocks we count how many blocks there are in total. We start with 1 block initially and every cut an extra block is generated. Later on will be clear why we want to know the total amount of blocks. Note that with this implementation the master has the obligation to keep offering empty blockparts from blocks that need to be cut to the slaves.

With PVM functions pvm_recv() and pvm_bufinfo() we are able to gather information about the next message to be received, but do not read it. This still has to be done outside the nextMessage function. Note that our choice for pvm_recv makes nextMessage a function that blocks progress until a message can be received.

nextMessage v.0
int nextMessage(SEQH seqH, int myId)
{
    static int nrBlocks = 1;    /* We start with an uncut block */
    int bufid, tid, tmp;

    bufid = pvm_recv(-1, -1);
    pvm_bufinfo(bufid, &tmp, &msgTag, &tid);

    switch (msgTag)
    {
        case MSG.CUT_BLK:
            /* We need to cut a block, so there will be 1 more block */
            nrBlocks++;
            break;
    }

    return msgTag;
} /* nextMessage */
4.3.6 Elementwise processing

In section 4.1.3 on page 38 we already presented the abstract sequential algorithm for the k-l elementwise processing.

As we have seen before, boolean variable $ch$ is used to select one of the two assignment statements in an alternating way. The first assignment statement determines the minimum element ($e$) from $x$. The second assignment statement applies $F$ to this element, stores the result in $y$, and removes $e$ from $x$. This will go on until the fixed point is reached, viz. $x = (\langle \rangle, \ldots, \langle \rangle)$.

We replaced the function $sl()$ from the abstract algorithm by a boolean array with dimension $K$. This array, also called $sl$, will have the following property $\forall i \in [1 \ldots K] :$

$$sl[i] = TRUE \iff sl(x,e)(i) = \langle e \rangle$$

This allows us to write the abstract $F_i(sl(x,e))$ into $F(e,i,sl)$; Note that we do not initialize $y$ ($y := (\emptyset, \ldots, \emptyset)$) in our implementation. This will be done before the first call of our function $EP()$. This way we can add the results of different processed blocks together in the same variable ($y$).

3.1

```c
void EP( Y *y, X *x)
{ /* Program 3.1 */
  int i;
  bool sl[K];
  
  while ( X.dom(*x) != 0 )
  {
    e = X.locv(*x);
    
    for ( i=0; i<K; i++ )
      if ( X.lem( x, i, e ) )
        /* e in z\'_i */
        sl[i] = TRUE;
      else
        /* e not in z\'_i */
        sl[i] = FALSE;
    
    for ( i=0; i<L; i++ )
    {
      fe = F( e, i, sl );
      Y.addH( y, i, fe );
    }
  } /* EP */
```

In abstract algorithm 6.4 fixed point condition (32) is different from the original fixed point in [Fóth 01]. Because in [Fóth 01] $K$ is equal to $L$, one can substitute one for the another. In our case, variable $i$ should range over $[1..L]$ and not over $[1..K]$.

Although not explicit stated in abstract algorithm 6.4, it is clear that variables $b_u$ and $y_u$ should be local to slave $i$. Variable $b$, which contains the description of the blocks that need to be processed, is hosted on the master and $x$ is scattered among the slaves, more precise $x_i$ is hosted on slave $i$.  

51
Abstract 6.4

\[
\begin{align*}
\text{var} & : b : BVS \quad bv : BVSP \quad x : X \quad yv : YV \\
FP & \Rightarrow \forall (i : i \in [1..L] : \bigcup_{j=1}^{K} yv_j[i] = F(x'_i))) \\
FP & \Rightarrow \forall (i : i \in [1..K] : bv_i = b = \langle \rangle) 
\end{align*}
\] (32) (33)

\( s_0 : \bigcup_{i=1}^{K} bv_i, yv_i := \langle \rangle, \langle \rangle \)
\[
\begin{align*}
\quad & \bigcup_{i=1}^{K} bv_i, b := \text{hiext}(bv_i, lov(b)), lorem(b) \quad \text{if } bv_i = \langle \rangle \land b \neq \langle \rangle \\
\quad & \bigcup_{i=1}^{K} yv_i, bv_i := EP(yv_i, x, lov(bv_i)), lorem(bv_i) \quad \text{if } bv_i \neq \langle \rangle 
\end{align*}
\]

When writing code for the master, we only have to consider those parts that access variables hosted by the master. In this case this is only the first assignment statement\(^{22}\) which assigns a block to slave \(i\) when there is a block to be processed \((b \neq \langle \rangle)\) and when slave \(i\) has no block to process at the moment \((bv_i = \langle \rangle)\).

Our implementation contains a repetition which ends when there are no blocks anymore in \(b\) \((b = \langle \rangle)\). Note that this does not imply that fixed point (33) is reached; we do not know if all \(bv_i\) variables are equal to \(\langle \rangle\), viz. the slaves could still be busy processing blocks.

An invariant of the repetition is that \(nr\text{ProcessingTasks}\) is equal to the total number of slaves that received at least one block to process. When a slave is ready processing it's block it will notify the master with a message. The master can check with function \(\text{whichTaskIsIdle}()\) which slave is able to process a new block. Note that once we emptied \(b\) we do not receive any idle messages from the slaves anymore. This means there will be \(nr\text{ProcessingTasks}\) messages that we still have to read. This will be done in section 4.3.7.

Master: 6.4

```c
int nrProcessingSlaves, idleId, tids[K+1];
BV blk;
BVS b;

nrProcessingSlaves = 0;
while ( BVS.dom( b ) != 0 )
{
    if ( nrProcessingSlaves < K )
    {
        idleId = tids[nrProcessingSlaves+1];
        nrProcessingSlaves++;
    }
    else
        idleId = whichTaskIsIdle();

    BVS.lov( b, blk);
    BVS lorem( &bh);
    SendBlock( idleId, MSG_EP, &bh[0], K );
}
```

First let us take a look at the communication behavior between slave \(i\) and the master this far.

\(^{22}\)This are actually \(K\) assignment statements, the \(\square\) operator is actually used for the quantified assignment-list. See section 1.4 on page 9 for more about UNITY.
An outgoing message will be preceded by an uparrow (↑), an incoming message with a down arrow (↓). From now on we will assume every slave will get at least 1 message to process.\footnote{In section C.3.4 on page 93 one can find the code where we took care of the situation where fewer blocks need to be processed. It involved an extra if statement and a small change to the function nextMessage().}

**Master**:

\[ \uparrow (msg\_tids) \downarrow (msg\_block) (\uparrow (msg\_cut\_blk) \downarrow (msg\_lower\_blk) \downarrow (msg\_upper\_blk)) \downarrow (msg\_ep) \uparrow \]

**Slave i**:

\[ \downarrow (msg\_tids) \uparrow (msg\_block) (\downarrow (msg\_cut\_blk) \uparrow (msg\_lower\_blk) \uparrow (msg\_upper\_blk)) \uparrow \]

It is clear that those 2 communication behaviors are matching and therefore there is no danger for deadlock. Note that until now there was no communication between slaves.

We constructed the implementation of the abstract algorithm 6.3 for the slaves in such a way that after receiving a message tag different from MSG.CUT.BLK the repetition will be ended. Because this message has message tag MSG.EP we can use a do - while repetition which will be exited when a message with an other tag as MSG.EP will be received. Within this loop the slave has to unpack the message, request data from the other slaves, collect this data, process it and finally notify the master that it is ready to receive a new block.

### Slave: 6.4

```c
int myId, myParent, tids[K];
SEQH seqH;
BV bv;
X x;
Y yv;

do
{
    pvm.upkBLOCPART( bv, K, 1);

    /* request needed data */
    RequestData( tids, bv, myId);

    /* collect needed data */
    CollectData( &x, bv, seqH, myId);

    /* yv = BP( yv, x, bv ) */
    BP( &yv, &x);

    pvm.initSend(PvmDataDefault);
    pvm.send( myParent, MSG.IDLE);
}
/* while ( nextMessage(seqH, myId) == MSG.EP ); */
/* nextMessage == START_TRANSFER */
```

Note that `pvm.upkBLOCPART()` has K as second argument. This means K blockparts will be received at once which form a block descriptor.

Implementation of the function `RequestData()` is trivial. The only remark we have to make is that we will always request data, even if the requested blockpart is empty. Later it will become clear why we made this decision.

The function `CollectData()` is very simple. We just use `nextMessage()` and receive the next K - 1 messages that contains an answer on our request. Because we could also receive requests from
other slaves at this point we will design nextMessage in such a way that it answers those requests by itself and wait for the next message to receive.

Let us take a closer look at the changes we made in nextMessage v.1. The repetition we introduced will filter out all messages with the MSG_REQUEST tag and not return the function until an other message is received. The new static integer nrCollect will count how many answers we received on our request for data. The other static integer nrBlocks will be used by a slave to count how many blocks he still needs to process or provide data for. Note that at this moment we use that data for empty blockparts will be requested and that the number of slaves is equal to the dimension of X.

The switch statement contains now all possible message tags we may encounter and those tags are:

MSG_REQUEST : This tag indicates that an other message is requesting data. We need to answer this request and do that with the function AnswerRequest(). Because the requesting slave is processing one of the blocks we need to decrease nrBlocks.

MSG.COLLECT : A message arrived with the data we requested. We need to increase nrCollect with 1. When we received K-1 messages with data, we know that all our requests are answered. Therefore we are able to process a block and we can decrease nrBlocks.

MSG.CUT.BLK : A block needs to be cut which means we have to increase nrBlocks.

MSG.EP : The master sends us a block that needs to be processed. In case the dimension of the input is 1 (or when we are the only slave) we should decrease nrBlocks now, because we do not need to request data from other slaves.

```c
nextMessage v.1

int nextMessage(SEQH seqH, int myId)
{
    static int nrBlocks = 1;    /* We start with an uncut block */
    static int nrCollect = 0;    /* we didn't collect messages yet */
    int bufId, tid, tmp;

    do
    {
        bufId = pvm_recv(-1, -1);
        pvm_bufinfo( bufId, &tmp, &msgTag, &tid);
        switch (msgTag)
        {
            case MSG_REQUEST:
                /* An other task is processing a block */
                AnswerRequest(tid, seqH, myId);
                nrBlocks--;
                break;
            case MSG.COLLECT:
                /* Data is arriving that we requested */
                nrCollect++;
                if (nrCollect == K-1)
                    { /* All needed information is collected */
                        nrBlocks--;
                        nrCollect = 0;
                    }
                break;
            case MSG.CUT.BLK:
                /* We need to cut a block, so there will be 1 more block */
                nrBlocks++;
                break;
            default:
                break;
        }
    } while (nrBlocks > 0);

    return;
}```
case MSG_EP:
    if (K==1)
    {
        /* No need to request data, so decrease nrBlocks */
        nrBlocks--; 
    }
    break;
} while ( msgTag == MSG_REQUEST );

return msgTag;
} /* nextMessage */

4.3.7 Collecting the partial results

Before we start implementing the collecting of the partial results, we look back at the imple- 
mentation of of Slave 6.4. The loop in this implementation continues until a message with tag 
START_TRANSFER arrives. This is the sign that all blocks are processed and that the slave can 
start sending the results to the master.

We could let the master send all the slaves a message with message tag START_TRANSFER when 
all blocks are processed. This should not be done after the last block is assigned to a slave because 
the other slaves might still need to answer requests. The right moment to send this message is 
when all slaves are ready processing their block. This means that all slaves and the master have 
to wait with collecting the partial results until all slaves are ready. In the worst case, K – 1 slaves 
and the master have to wait until a block of size B is processed.

Fortunately we can avoid this extra idle time and the K messages the master has to send. Re- 
member that in nextMessage() we used the integer nrBlocks to count for how many blocks a slave 
still work has to do. When all work is done, viz. nrBlocks = 0, we let nextMessage() fake a 
message from the master in order to notice the slave it can start sending the partial results.

nextMessage v.2

int nextMessage(SEQH seqH, int myId)
{
    static int nrBlocks = 1; /* We start with an uncut block */
    static int nrCollect = 0; /* we didn’t collect messages yet */
    int bufid, tid, tmp;

    if (nrBlocks > 0)
    do
    {
        bufid = pm_recv( -1, -1);
        pmbufinfo( bufid, &tmp, &msgTag, &tid);
        switch (msgTag)
        {
            /* no changes made in this part of the code */
        }
    } while ( (msgTag == MSG_REQUEST ) && (nrBlocks > 0));

    if (nrBlocks == 0)
        return START_TRANSFER;
    else
        return msgTag;
} /* nextMessage */
Abstract algorithm 6.5 starts with emptying variable $y$, which will be used to store the result. Then the value of every $yv_i[j]$ is added to $y_j$. The fixed point is reached when every $yv_i[j]$ is empty.

The abstract algorithm we present here is different from the original in [Fóth 01]. Because in [Fóth 01] $K$ is equal to $L$ one can substitute one for the other, in our implementation we can not do this.

**Abstract 6.5**

\begin{align*}
\text{var :} & \quad y : Y & yv : Y V \\
\text{inv :} & \quad \forall (j : j \in [1..L]) : \bigcup_{i=1}^{K} yv_i[j] \cup y_j = \bigcup_{i=1}^{K} yv_i[j] \quad (34)
\text{FP }\Rightarrow & \quad \forall (i, j : i \in [1..k], j \in [1..L] : yv_i[j] = \emptyset)
\text{s}_0 : & \quad \{ \begin{array}{l}
L_{i=1}^{L} y_j := \emptyset \\
K_{i=1}^{k} \quad yv_i[j] := y_j \cup yv_i[j], \emptyset \end{array} \}
\end{align*}

Figure 11 on page 17 showed us that communicating a big message at once is cheaper that small parts one by one. Therefore we will rewrite the statement-list from the abstract program into

\begin{align*}
\{ \bigcup_{i=1}^{K} y, yv_i := y \cup yv_i, \emptyset \}
\end{align*}

This has the consequence that we can rewrite invariant (34) into

\begin{align*}
\bigcup_{i=1}^{K} yv_i \cup y = \bigcup_{i=1}^{K} yv_i'
\end{align*}

Note that in the implementation of the master we only expect a result from $nrProcessingSlaves$ slaves. First the message with the $MSG.IDLE$ tag is received, something we still had to do. With $pvm.upkY.andAdd()$ a result is received and stored in $y$.

**Master: 6.5**

```c
int nrProcessingSlave;
Y y;
Y.init( &y);

while ( nrProcessingSlaves > 0 )
{
    pvm.recv( -1, MSG.IDLE ); /* Clean-up the MSG.IDLE's */
    pvm.recv( -1, MSG.TRANSFER );
    pvm.upkY.andAdd( &y);
    nrProcessingSlaves--;}
Y.free( &y);
```

56
int myParent;
Y yv;
pvm_initsend( PvmDataDefault);
pvm_pkY( &yv);
pvm_send( myParent, MSG_TRANSFERT);

4.3.8 Absence of deadlock

We will now show in an informal way that there is no danger for deadlock in our implementation. Recall that the function nextMessage() is the only way slaves receive messages and therefore deadlock can only occur within this function. The function nextMessage() is constructed in such a way that a possible blocked slave is always able to answer requests for data from other slaves.

In implementation Slave: 6.4 on page 53 there are 2 points where a slave needs to receive a message: in function CollectData() and in the guard of the loop.

Function CollectData() is proceeded by a request to all other slaves for data. The only way there could occur deadlock at this point is when one of the other slaves is not answering this request. A slave a request is send to can be in one of the following states:

The slave is stuck in function nextMessage(): Because the function nextMessage() is constructed in such a way that a request for data can always be answered this is not a concern.

The slave is requesting data or processing a block: In both function RequestData() and EP() there is no danger for deadlock. Because after execution of those functions there is a call to nextMessage() (in CollectData() there are even K − 1 calls), slaves will be able to answer requests for data.

A slave left the loop in implementation Slave: 6.4: There is only 1 way for a slave to leave this loop; receiving a message with a tag that is different from MSG_EP. There is only one message that can be received at this moment: START TRANSFER.

Recall that this is in fact a message faked by nextMessage() when variable nrBlocks reaches 0. Suppose there were v blocks constructed initially. When nrBlocks = 0 then a slave received exactly v messages with a MSG_EP or a MSG_REQUEST tag. The fact that there arrives an other request for data means that all slaves together try to process at least v + 1 blocks.

Because there are only v blocks this situation can not occur. Therefore a slave can not leave the loop of Slave: 6.4 before a request for data arrives.

4.4 Timing

There are a lot of parameters involved when doing performance measurements on our implementation of the parallel elementwise processable function. Take for example the input where not only the set-size and the bag-size matters but also the type of the input.

We decided to restrict ourselves to 3 different type of inputs, all with a bag-size of 30000 and a dimension (K) of 3. Each slave and the master will be hosted on an individual processor. This means that we need 4 processors which form exactly 1 cluster of the PowerXplorer.
The actual elementwise processable function will be simulated by the function we already used in section 3.4 for our measurements on the associative function. Recall that the weight \((W)\) is a natural number which states how often a certain function is calculated, each time with the same arguments. A low weight represents an easy to calculate function, a high weight a complex function.

First we will specify the 3 types of inputs that are going to be used followed by some measurements on the costs of the actual elementwise processable function. Section 4.4.3 analyses the construction of a total disjoint decomposition for different values of \(B\).

In section 4.4.4 we will try to find an optimal value for \(B\) using measurements. Also some analyses will be done of the speedup reached for different values of \(B\), followed by an analyses of the idle-time. An optimized algorithm will be presented in section 4.4.6 together with some analyses. In the last section we will present a model that can help finding a \(B\) for which our implementation will be the most efficient.

### 4.4.1 The input

Because we would like to see how performance depends on the input we decided to design 3 different kinds of input. Recall that the input is of type \(X\), a direct product of \(K\) sequences. although these sequences may have distinct set-sizes, we will only consider measurements on inputs for which all set-sizes are equal because this facilitates the analyses of the results.

The 3 input types we will use are called \(SAME\), \(STEP\), and \(DISJ\). In the following explanation, every type contains sequences with set-size \(n\). This means that the bag-size of every type is \(K \times n\). In our case where \(K = 3\) the bag-size is therefore \(3n\).

**SAME**

As the name already indicates, all sequences in this input type are the same. Therefore the total input has a set-size of \(n\).

\[
\begin{array}{c|c}
\hline
x_1 & \langle 1,2,3,4,5,6,\ldots,n \rangle \\
\hline
x_2 & \langle 1,2,3,4,5,6,\ldots,n \rangle \\
\hline
x_3 & \langle 1,2,3,4,5,6,\ldots,n \rangle \\
\hline
\end{array}
\]

**STEP**

In the \(STEP\) input-type, a sequence \(x_i\) with length \(n\) is equal to \(\langle i,2i,\ldots,ni \rangle\). This way we can create an input with elements that exist only once (e.g. 1), twice (e.g. 2) and three times (e.g. 6) in the same input.

The set-size of the \(STEP\) input type can be calculated with the following formula\(^{24}\)

\[
3n - n \text{ div } 2 - 2 \times (n \text{ div } 3) + n \text{ div } 6
\]

In case \(K = 3\) one can say that \(M_{\text{step}} = \frac{3}{3}N_{\text{step}}\).

\[
\begin{array}{c|c}
\hline
x_1 & \langle 1,2,3,4,5,6,\ldots,n \rangle \\
\hline
x_2 & \langle 2,4,6,8,10,12,\ldots,2n \rangle \\
\hline
x_3 & \langle 3,6,9,12,15,18,\ldots,3n \rangle \\
\hline
\end{array}
\]

**DISJ**

In the \(DISJ\) input type all sequences are disjoint. No element exist in more than 1 sequence at the same time. Sequence \(x_i\) will be equal to \(\langle (i-1)n+1,(i-1)n+2,\ldots,in \rangle\). The set-size of input from the \(DISJ\) type will be \(3n\) which is equal to the bag-size, viz. \(M_{\text{disj}} = N_{\text{disj}}\).

\(^{24}\)A proof of this formula can be found in section A.4 on page 76.
\[
\begin{array}{c|c}
 x_1 & \langle 1, 2, 3, 4, 5, 6, \ldots, n \rangle \\
 x_2 & \langle n + 1, n + 2, n + 3, n + 4, n + 5, n + 6, \ldots, 2n \rangle \\
 x_3 & \langle 2n + 1, 2n + 2, 2n + 3, 2n + 4, 2n + 5, 2n + 6, \ldots, 3n \rangle \\
\end{array}
\]

Note that when we let \( x_j \) be equal to \( \langle i, i + K, i + 2K, \ldots, i + nK \rangle \) we also have disjoint sequences with a total set-size of \( 3n \) but when producing a \( \text{cdd} \) different blocks will be generated.

In figure 27 one can find an example of both possible input-types. Suppose the first block should be processed by slave 1. In the first situation slave 1 hosts \( \langle 1, 2, \ldots, 7 \rangle \) and therefore no data has to be communicated. Using the second input-type slave 1 needs to receive the sequences \( \langle 2, 5 \rangle \) and \( \langle 3, 6 \rangle \) from the other slaves.

\[
\begin{array}{c|c|c|c|c|c|c|c}
 x_1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 x_2 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
 x_3 & 15 & 16 & 17 & 18 & 19 & 20 & 21 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
 x_1 & 1 & 4 & 7 & 10 & 13 & 16 & 19 \\
 x_2 & 2 & 5 & 8 & 11 & 14 & 17 & 20 \\
 x_3 & 3 & 6 & 9 & 12 & 15 & 18 & 21 \\
\end{array}
\]

Figure 27: Cdd of different input-types with disjoint elements.

In our measurements on the elementwise processable function we will only use the here introduced input types with a total bag-size of 30,000.

### 4.4.2 Elementwise processing slices

First we need to introduce the term slice. A slice is the smallest non-empty block one can construct from a certain input. The set-size of a slice \( (M_{\text{slice}}) \) is always 1 and for the bag-size of a slice holds \( N_{\text{slice}} \in [1 \ldots K] \). Figure 22 on page 35 shows 5 of those slices. The number of slices in an input is the same as the set-size of an input.

The results presented in figure 28 show the costs for calculating the results for an elementwise processable function with a weight \( W \) on different input in a sequential way. The results obtained this way, presented in figure 28, corresponds to what we expected: the time needed to process a certain input depends linear on the weight of the used elementwise processable function.

It is also clear that the bag-size of the input has no influence on the total processing time. It is clear the the total time depends linear on the set-size of the input. Measurements with smaller bag-size, and therefore a smaller set-size, underline those observations \(^{25}\). It is possible to construct a formula that expresses the time needed to process a slice in terms of the weight \( (W) \) of the used elementwise processable function:

\[ T_{\text{slice}}(W) \approx 14.6 + 0.274W \]

The costs for processing a certain input with a sequential algorithm can be expressed in terms of the set-size \( (M) \) of the input and \( W \):

\[ T_{\text{seq}} = MT_{\text{slice}}(W) \]

\(^{25}\) Those measurements are not included in this graph for easy reference.
4.4.3 Constructing a cdd

In figure 29 one can see the number of blocks constructed for a certain B in case of an input of the SAME and the STEP type. For both types also the minimum and the maximum set-size and bag-size of the constructed blocks is given. In figure 29 we wrote e.g. $N_{\text{max}}$ to denote the bag-size of the largest block instead of $N_{\text{max,block}}$ in order to save space.

<table>
<thead>
<tr>
<th>B</th>
<th>#blk</th>
<th>$N_{\text{max}}$</th>
<th>$N_{\text{min}}$</th>
<th>$M_{\text{max}}$</th>
<th>$M_{\text{min}}$</th>
<th>#blk</th>
<th>$N_{\text{max}}$</th>
<th>$N_{\text{min}}$</th>
<th>$M_{\text{max}}$</th>
<th>$M_{\text{min}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1968</td>
<td>24</td>
<td>12</td>
<td>8</td>
<td>4</td>
<td>1751</td>
<td>25</td>
<td>11</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>50</td>
<td>1024</td>
<td>33</td>
<td>24</td>
<td>11</td>
<td>8</td>
<td>896</td>
<td>41</td>
<td>25</td>
<td>28</td>
<td>16</td>
</tr>
<tr>
<td>100</td>
<td>512</td>
<td>63</td>
<td>54</td>
<td>21</td>
<td>18</td>
<td>448</td>
<td>77</td>
<td>51</td>
<td>55</td>
<td>36</td>
</tr>
<tr>
<td>250</td>
<td>128</td>
<td>240</td>
<td>231</td>
<td>80</td>
<td>79</td>
<td>208</td>
<td>210</td>
<td>128</td>
<td>210</td>
<td>76</td>
</tr>
<tr>
<td>500</td>
<td>64</td>
<td>474</td>
<td>465</td>
<td>158</td>
<td>155</td>
<td>104</td>
<td>418</td>
<td>258</td>
<td>418</td>
<td>154</td>
</tr>
<tr>
<td>1000</td>
<td>32</td>
<td>942</td>
<td>933</td>
<td>314</td>
<td>311</td>
<td>52</td>
<td>834</td>
<td>518</td>
<td>834</td>
<td>310</td>
</tr>
<tr>
<td>2500</td>
<td>16</td>
<td>1878</td>
<td>1872</td>
<td>626</td>
<td>624</td>
<td>14</td>
<td>2294</td>
<td>1666</td>
<td>1666</td>
<td>1248</td>
</tr>
<tr>
<td>5000</td>
<td>8</td>
<td>3753</td>
<td>3747</td>
<td>1251</td>
<td>1249</td>
<td>7</td>
<td>4586</td>
<td>3333</td>
<td>3333</td>
<td>2498</td>
</tr>
<tr>
<td>10000</td>
<td>4</td>
<td>7503</td>
<td>7497</td>
<td>2501</td>
<td>2499</td>
<td>4</td>
<td>9167</td>
<td>3333</td>
<td>6667</td>
<td>3333</td>
</tr>
<tr>
<td>25000</td>
<td>2</td>
<td>15000</td>
<td>15000</td>
<td>5000</td>
<td>5000</td>
<td>2</td>
<td>18332</td>
<td>11668</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>50000</td>
<td>1</td>
<td>30000</td>
<td>30000</td>
<td>10000</td>
<td>10000</td>
<td>1</td>
<td>30000</td>
<td>30000</td>
<td>20000</td>
<td>20000</td>
</tr>
</tbody>
</table>

Note that the number of blocks constructed depends not only on B and N but also on the input type. For the SAME input type the bag-sizes of the blocks are exactly 1/3 of the set-sizes. For the SAME input type we observe that $M_{\text{max,same}}$ and $M_{\text{min,same}}$ do not differ much. This means that all blocks can be processed in about the same amount of time ($T_{\text{step}} = M_{\text{set}}$). For the STEP input type the bag-sizes of the largest and the smallest block differ much more. In some cases $M_{\text{min,step}}$ is 1/3 of $M_{\text{max,step}}$ which means that some blocks are 3 times more expensive to process than others.

All the numbers, except for B = 25, in the second column of figure 29 are a power of 2. We did
not intended this when we chose the values of $B$ we wanted to use in our measurements. But now we noticed this we could try to find an approximation for the numbers of blocks that will be cut.

Because every block has a bag-size of at most $B$ we can use $\frac{N}{B}$ as a lower bound. That the function $2^{\log(\frac{N}{B})}$ is a nice approximation for the \textit{SAME} input type, with $B$ not too small, can be seen in figure 30. It is also clear from this figure that our approximation does not work for the \textit{DISJ} input type. Note that the x-axis in figure 30 is logarithmic and ranges only from 20 to 500. This restriction of the x-axis was done in order to make the graph more clear.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure30.png}
\caption{Number of blocks cut for a certain $B$ with $N = 30000$.}
\end{figure}

In order to construct $v$ blocks, one has to make $v - 1$ cuts. Now we define $T_{\text{cut}}$ to be the time needed to cut 1 block in half. Included in $T_{\text{cut}}$ is the time the master needs to select a block, tell the slaves about it, receive the results from the slaves after they cut the block and store the 2 new constructed blocks. Measurements we did showed that for the PowerXplorer $T_{\text{cut}} \approx 1400$.

Now we can calculate the costs of constructing a complete disjoint decomposition in the case where e.g. $B = 500$, $N = 30000$ and the input is of the \textit{STEP} type. Figure 29 shows that there will be 104 blocks constructed which means that 103 cuts had to be made. Therefore the costs for constructing this CDD is $103 + T_{\text{cut}} \approx 144,200$ clock ticks.

\subsection{Finding an optimal $B$}

The parallel program for the elementwise processable function consists of 3 parts: constructing a total disjoint decomposition (cutting the blocks), distributing blocks over the slaves which process them (elementwise processing) and collecting the partial results from the slaves.

Instead of measuring all 3 parts we decided to leave out the collecting of the partial results. This is in fact nothing else then slaves sending messages to their master. The total size of those messages depend on the set-size of the input and will therefore not be influenced by the size of $B$ and the weight of the elementwise processable function.

In figure 31 are the results for the \textit{SAME} input type with the weights 10, 100 and 1000 displayed. The 3 extra horizontal lines show the time needed by the sequential algorithm. One could conclude from this picture that for a high weight a $B$ of size between 250 and 25 000 would be the most
efficient choice. For a lower weight a value of $B$ should be taken in a narrower area somewhere between 10 000 and 25 000.

In [Föth 01] a $B$ of size $M/K^2$ (remember, $M$ is the set-size) is expected to be a good choice: "In general the dominant costs are the processing steps, that's why $B = M/K^2$ is a good choice." Using this will result in a $B$ of $30 000/9 = 1 111$ which is indeed a good choice for a heavy weight. In section 4.4.7 we will try to find the most optimal value $B_{opt}$.

![Figure 31: Cutting and ep-time for the SAME input type.](image)

The major performance increase seen in figure 31 for increasing $B$ from 25 to 50 and 100 is due to the cost reduction in constructing a total disjoint decomposition. For $B = 25$ this cutting of the blocks is responsible for about 50% of the costs.

Now let us try to understand the results for $B = 10 000$, 25 000 and 50 000 with the use of the following data:

<table>
<thead>
<tr>
<th></th>
<th>nrBlocks</th>
<th>cut-Time</th>
<th>ep-Time</th>
<th>total Time</th>
<th>avg. blocksize</th>
<th>ep-time block</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 000</td>
<td>4</td>
<td>4 213</td>
<td>1 599 680</td>
<td>1 603 893</td>
<td>2 500</td>
<td>721 500</td>
</tr>
<tr>
<td>25 000</td>
<td>2</td>
<td>1 550</td>
<td>1 686 765</td>
<td>1 688 315</td>
<td>5 000</td>
<td>1 443 000</td>
</tr>
<tr>
<td>50 000</td>
<td>1</td>
<td>1</td>
<td>2 982 181</td>
<td>2 982 182</td>
<td>10 000</td>
<td>2 886 000</td>
</tr>
</tbody>
</table>

In case $B = 50 000$ there is only one block and our implementation should behave like the sequential program. We see that the time needed by the parallel program is $2,982,182 - 2,886,000 = 96,181$ time ticks slower. Most of this extra time is used for communicating, like e.g. the distribution of the data that is hosted by the other slaves. From figure 11 on page 11 it is known that communicating 10,000 integers takes 41,378 time ticks.

Comparing the results where $B = 10 000$ with $B = 25 000$ we see just a small difference. This seems to be strange because when $B = 25 000$ only 2 blocks need to be processed and the third processor is not used in the elementwise processing. For $B = 10 000$ 4 blocks need to be processed and all processors can be used, so one would expect a much more efficient result (cutting time can be neglected in this situation). Because there are 4 blocks that need to be processed, one of the processors has to process 2 blocks. These 2 blocks will each have an average size of 2 500 which will take as much time to process as 1 block of size 5 000.
In figure 32 the speedup for processing the elementwise processable function is presented in case of the SAME, STEP and DISJ input type. The time needed to cut the blocks is left out. Although for 3 processors we would expect a speedup of 3 it can be observed from figure 32 that no speedup is reached larger than 2.

![Speedup Graph](image)

**Figure 32: Speedup elementwise processing**

### 4.4.5 Idle time

In the previous section we saw that we could not reach a speedup better than 2. There are 2 possible causes for these poor results: communication costs and idle time. In section 4.3.1 on page 40 we expected to be confronted with idle time. Therefore we will concentrate on idle-time and not on communication costs in this section.

It is obvious that idle time on the master is not decreasing the speedup of our algorithm. Idle time on the slaves can occur in various situations and some of them we will not measure:

**Waiting for a new block to process**:
After a slave has notified the master that it is ready to process a new block, it has to wait until the master answered his request.

Although this is idle time we will not take it into account. We expect the master to react almost immediately on requests and therefore that idle time will be rather small. The total idle time at this point will depend on the number of blocks a slave has to process. Measurements have shown that only for small $B$ ($< 50$) and small $W$ ($< 100$) this idle-time can have some significant influences on the total performance but for larger values the maximum idle-time does not exceed the 400 clock-ticks.

Idle time at this point could be avoided by sending slaves blocks to process in advance. A problem with this is that it will hamper the load balancing that is tried to be done with the construction of the complete disjoint decomposition of the input.

**Waiting for other slaves to send data**:
When a slave received a new block to process, data has to be requested from the other slaves. Because slaves can not always react immediately on a request and answering a request will cost time also, there will be idle time at this point.
This idle time interests us the most and it will become clear later on that this is affecting performance most.

**Slaves are ready processing but are waiting to answer requests from other slaves:**
Because an idle slave has nothing else to do anymore, except returning the partial results to the master and answering the request for data, idle time is not effecting overall performance. The overall performance is affected by the unbalanced load which depends on the way the input is divided in blocks. Therefore this idle time will not be measured.

![Figure 33: Idle time](image)

The idle time presented in figure 33 is the average over all 3 the processors. E.g. in the case where $B = 25000$ the idle time on the different slaves was $20434$ (sd$^{26}$ $552$), $1308$ (sd $224$) and $0$ (sd $0$).

Figure 33 shows that idle time depends on the weight of the elementwise processable function and the size of $B$ (the number of blocks that need to be processed). We feel the need to warn the reader for the results displayed for $B = 25$. The standard deviation for all three weights was about $300000$, which is about $20\%$ of the total average idle time.

In figure 34 the same data as in figure 33 is presented, but now the idle time is presented as a percentage of the total time needed for the elementwise processing. In most cases more than $20\%$ if the elementwise processing time is caused by idle time. In section 4.4.6 we will try to bring this number down.

### 4.4.6 Reducing the idle time

One of the largest cause of idle time is the impossibility of slaves to answer requests made by other slaves immediately. Only when a call to the function `nextMessage()` is made a slave can receive a request for data and answer it. Because such a call is only made before and after processing a complete block those calls are not made very frequent.

Because there is no call to `nextMessage()` during the elementwise processing of a block we add some extra code to the function `EP()` which will check after the processing of each element from a block if there is a possible request to answer. If there is, this request will be answered immediately.

---

$^{26}$Standard deviation
This way we create an implementation that gives a higher priority to answering requests compared to our original implementation.

3.1 Optimization

```c
void EP( Y **y, X **x, SEQH seqH, int myId)
/* Program 3.1, optimised version */{
    ... declarations of vars ...

    while ( X.isNotEmpty(*x) )
    {
        e = X.getMinimum(*x);
        ... process element e ...

        /* Is there a request for data? */
        bufId = pvm.nrecv( -1, -1);
        if ( bufId > 0 )
        {
            /* Yes, there is a request for data */
            pvm.bufinfo( bufId, &tmp, &msgTag, &tid);
            AnswerRequest(tid, seqH, myId);
            /* Tell nextMessage() we answered a request */
            nextMessage(seqH, -2);
        }
    }
} /* EP */
```

The PVM function `pvm.nrecv()` is a non-blocking version of the `pvm.recv()`. We can use a wildcard for the message tag because at this moment the only messages arriving will be requests. If there is a request it will be answered with `AnswerRequest()` and `nextMessage()` will be informed that a request is handled. This is necessary because otherwise the static integer `nrBlocks` owned by `nextMessage()` will have the wrong value. The complete code for the optimized functions `EP` and

---

When 2 requests for data arrive at the same time, only one will be answered in our optimization. The second
nextMessage() can be found in section C.3.5 on page C.3.5.

Note that in this solution the function pvm_nrecv() is called ‘set-size’ times in total. The cost of 1 such call is about 8.5 time ticks. For the SAME input type with B = 25 this will add about $10000/3 \times 8.5 = 28333$ extra time ticks to the costs. In case $B = 50000$ those extra costs will be 85,000.

Figure 35: Speedup elementwise processing after optimization

Figure 35 displays the speedup reached by our optimized program. In most cases a speedup of 2.5 is reached and in case of the DISJ input type the speedup is even reaching the value of 2.8. In the case where $B = 25000$, 1 processor has to do about 50% of the work and therefore a speedup of 2 is the maximum reachable. With $B = 50000$ our parallel program uses only 1 processor to process the input and therefore a speedup of 1 is the maximum reachable value.

4.4.7 Model

In this section we will make a model which we can use to analyze the results of our implementation. We will start with a very simple model and extend it until the values produced by the model predict the behavior of our implementation.

One can express the time needed by an algorithm as the sum of the costs of all communications, processing steps and idle times. In this case the algorithm consist of 3 parts that we can analyze separately. Therefore we will express the time needed by the algorithm ($T$) as the sum of the time needed to construct a cdd ($T_{cdd}$), elementwise process the blocks ($T_{ep}$) and distribute the results back to the master ($T_{dist}$):

$$T = T_{cdd} + T_{ep} + T_{dist}$$

Analogous to the previous sections we will not consider $T_{dist}$ in our model.

request will have to wait until the next element is processed. It would have been better when we would have used "while (pvm_nrecv() > 0)" instead of the if solution. We mentioned the if solution because this one was used in our measurements. We do not expect significant differences in performance when using the while construction and therefore the decision was made not to redo the measurements.
A simple model without communication

In section 4.4.3 on page 60 we found that cutting a block \((T_{cut})\) costs about 1400 clock ticks. To find \(T_{odd}\) we need to multiply \(T_{cut}\) with the total number of times a block has to be cut. The actual number of cuts depend on \(N, B\) and the type of the input. In this model we will settle for a lower bound. In figure 30 we established \(\frac{N}{B}\) as a lower bound for the total number of blocks. To construct \(\frac{N}{B}\) blocks one need to perform \(\frac{N}{B} - 1\) cuts.

\[
T_{odd} = (\frac{N}{B} - 1)T_{cut}
\]

The time to elementwise process the blocks can be expressed as the sum of the time needed by communications \((T_{com})\), time needed by processing the data \((T_{proc})\) and the idle time \((T_{idle})\).

\[
T_{ep} = T_{com} + T_{proc} + T_{idle}
\]

Because we want to keep our model simple at this point we will take \(T_{com} = T_{idle} = 0\) and only concentrate on \(T_{proc}\).

In section 4.4.2 on page 59 we established the formula \(T_{slice}(W) \approx 14.6 + 0.274W\) which we will rewrite in the more general form \(T_{slice}(W) = \beta + \gamma W\). Multiplying this with \(M\), the set-size of the total input, and dividing the result by \(K\) will result in the average costs per processor. This average costs per processor reflects the situation where we have a perfect load-balance.

\[
T_{proc} = \frac{1}{K} MT_{slice}(W)
\]

The value of \(M\) is mostly unknown initially and therefore we will try to avoid the use of it in our model. \(M\) depends on \(N\) and on the type of input that has to be processed. Fortunately we can replace \(M\) by \(\varphi_{input} N\) for the input types SAME, STEP and DISJ with for \(\varphi_{input}\) in case \(K = 3:\)

\[
M = \varphi_{input} N, \text{ with } \varphi_{input} = \begin{cases} 
\frac{1}{3} & , \text{if type is SAME} \\
\frac{2}{3} & , \text{if type is STEP} \\
1 & , \text{if type is DISJ}
\end{cases}
\]

The model constructed this far is

\[
T = (\frac{N}{B} - 1)T_{cut} + \frac{N}{K} \varphi_{input} T_{slice}(W)
\]

Adding unbalanced load to the model

We are not satisfied with this model. One of our goals was to find an optimal \(B\) \((B_{opt})\) but \(T\) reaches a minimum in our model when \(B\) goes to \(\infty\).

In real we are not always able to balance the load perfect on the available slaves. Therefore an \(\alpha\) is introduced to express this unbalance in the load.

\[
T_{proc} = (\frac{N}{K} \varphi_{input} + \alpha) T_{slice}(W)
\]

67
The smallest possible unbalance in our model is 1 block with set-size \( \varphi_{\text{input}} B \). Therefore we will choose \( \alpha = \varphi_{\text{input}} B \).

\[
T_{\text{proc}} = \left( \frac{N}{K} + B \right) \varphi_{\text{input}} T_{\text{slice}}(W)
\]

This will give us the following model:

\[
T = \left( \frac{N}{B} - 1 \right) T_{\text{cut}} + \left( \frac{N}{K} + B \right) \varphi_{\text{input}} T_{\text{slice}}(W)
\]

In order to find \( T_{\text{min}} \) one first has to find \( B_{\text{opt}} \), the optimal value for \( B \). \( B_{\text{opt}} \) is the \( B \) for which \( \frac{\partial}{\partial B} T = 0 \). \( T_{\text{min}} \) can thereafter be constructed by replacing every occurrence of \( B \) in \( T \) by \( B_{\text{opt}} \).

\[
\frac{\partial}{\partial B} T = -\frac{NT_{\text{cut}}}{B^2} + \varphi_{\text{input}} T_{\text{slice}}(W)
\]

\[
B_{\text{opt}} = \sqrt{N} \sqrt[2]{\frac{T_{\text{cut}}}{\varphi_{\text{input}} T_{\text{slice}}(W)}}
\]

Adding communication to our model

Recall from section 2.3 on page 15 that the costs of a communication of size \( u \) can be expressed with the simple communication cost model:

\[
T_{\text{msg}}(v) = t_s + vt_i
\]

Every block that needs to be processed will be assigned to a processor. This is done by the master by sending 3 integers \( T_{\text{msg}}(3) \). Every time a slave has completed the processing of a block he notifies the master \( T_{\text{msg}}(1) \). A slave needs to request data from all the other \( (K - 1) \) slaves when processing a block. The costs of these requests is \( (K - 1)T_{\text{msg}}(3) \). The \( (K - 1) \) answers with data a slave receives are on average \( \frac{K-1}{K} B \) integers large.

Multiplying those communications by the total number of blocks \( \frac{N}{B} \) results in the total communication costs. Note that we do not take \( \frac{N}{B} + 1 \), the unbalanced situation, as the total number of blocks because we do not want to make the model unnecessarily complicated. Some of these communications could occur in parallel. Fortunately this is never the case with communications involving the same slave.

\[
T_{\text{com}} = \frac{1}{K} \frac{N}{B} \left( T_{\text{msg}}(3) + T_{\text{msg}}(1) + (K - 1)T_{\text{msg}}(3) + (K - 1)T_{\text{msg}}(\frac{K-1}{K} B) \right)
\]

With this \( T_{\text{com}} \) we can construct our new model \( T \). For the sake of simplicity we introduce constant \( C_0 = T_{\text{msg}}(3) + T_{\text{msg}}(1) + (K - 1)T_{\text{msg}}(3) \) and replace \( T_{\text{msg}}(\frac{K-1}{K} B) \) by \( t_s + \frac{K-1}{K} B t_i \).

\[
T = \left( \frac{N}{B} - 1 \right) T_{\text{cut}} + \left( \frac{N}{K} + B \right) \varphi_{\text{input}} T_{\text{slice}}(W) + \frac{N}{BK} C_0 + \frac{N(K-1)}{BK} t_s + \frac{N(K-1)^2}{K} t_i
\]

What rests is finding a formula for \( B_{\text{opt}} \).

\[
\frac{\partial}{\partial B} T = -\frac{NT_{\text{cut}}}{B^2} + \varphi_{\text{input}} T_{\text{slice}}(W) - \frac{N}{B^2 K} C_0 - \frac{N(K-1)}{B^2 K} t_s
\]
\[ B_{opt} = \sqrt{N} \sqrt{\frac{KT_{cut} + C_0 + (K - 1)t_s}{K\varphi_{input}T_{slice}(W)}} \]

It is obvious from the previous formula that \( B_{opt} \) depends on e.g. \( N \). When we rewrite \( C_0 \) into its original form it becomes clear that \( B_{opt} \) does not depend on \( K \).

\[ B_{opt} \approx \sqrt{N} \sqrt{\frac{T_{cut} + 2t_s + 3t_i}{\varphi_{input}T_{slice}(W)}} \]

Now we will compare our model with the measurements we made in the previous section. To find \( B_{opt} \) for the PowerXplorer we need to use the following values: \( T_{cut} = 1400, t_s = 178.4, t_i = 4.12, \alpha = 14.6 \) and \( \gamma = 0.274 \):

<table>
<thead>
<tr>
<th>Type /W</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAME</td>
<td>3030</td>
<td>1962</td>
<td>743</td>
</tr>
<tr>
<td>STEP</td>
<td>2143</td>
<td>1377</td>
<td>525</td>
</tr>
<tr>
<td>DISJ</td>
<td>1750</td>
<td>1124</td>
<td>429</td>
</tr>
</tbody>
</table>

Figure 36: \( B_{opt} \) for \( N = 10000 \).

In figure 36 one can see that when \( W \) increases the value for \( B_{opt} \) decreases. This is logical because a larger \( W \) means larger costs for processing a block and therefore a larger unbalance in our model. A smaller \( B_{opt} \) will decrease the size of blocks and therefore the costs for processing a block. The costs of a smaller \( B_{opt} \) are a larger \( T_{cdd} \) and more communication costs caused by more blocks.

The same happens when the input-type changes. The set-size of a block from the \( STEP \) input-type is larger than the set-size of a block from the \( SAME \) input-type. This means larger costs for processing a block and therefore a larger unbalance in our model.

In figure 37 one can see the difference between our model and the measurements we did in the previous section. A difference of 50% means that our model predicts half the time measured. A negative value of for example -100% means our model predicts twice as much time as measured.

Figure 37: Difference between measured results and our model for \( N = 30000 \) and \( W = 1000 \).
Figure 37 shows that our model is quite accurate for $B \in [250\ldots2500]$ with $N = 30000$ and $W = 1000$. For $B < 250$ the results of our model are better than in real. We suspect that this is caused by the increased number of communications that have to be performed.

The poor results for $B > 2500$ are caused by the way we unbalanced the load in the model. Recall that we use $(\frac{N}{K} + B)$ for determining how many elements a slave has to process. For $B = 50000$ our model expects $30000/3 + 50000 = 60000$ elements to be processed by the slave with the most work instead of the actual $30000$ ($-100\%$). For $B = 25000$ our model expects $35000$ elements to be processed which is much more than the actual $15000$ elements (there are $2$ blocks in this situation), a difference of $-133\%$.

4.5 Remarks

The first publication of [Fóth 01] appeared in [Varg 01, p 180-194]. Unfortunately there were some small errors in this version that were changed in later versions/private communications. For all who have no access to a later version, some important changes can be found in section B on page 77.

4.5.1 Optimization of the algorithm

One can optimize our implementation in many ways. We already mentioned some in section 4.3.2 where we could have implemented some datatypes more efficient. At this point we are not interested in optimizations concerning this kind of implementation details.

In our implementation the master is assigning blocks to slaves in a random way. One could try to assign blocks in such a way that communication costs are minimized. Take for example the input and its complete disjoint decomposition presented in the upper half of figure 27 on page 59. It is obvious that communication costs can be avoided when the first block, ($\ll 1, 2, \ldots, 7 \gg, \ll \gg$ , $\ll \gg$ ), would be processed by the first slave because this slave already hosts all elements needed.

An other way to optimize the algorithm is to try to balance the load better over the available processors. In the model we constructed we created an unbalance in the load by assigning one of the slaves an extra block. By taking this block as small as possible one could optimize the load balance. This could be done by the master which should first offer large blocks to the slaves keeping the smallest ones until the end.

An other solution for optimizing the load balance is adjusting the value for $B_{opt}$ dynamic. The first blocks that need to be processed could be large, followed by smaller ones. Blocks can be constructed in advance of the elementwise processing or dynamic whenever they are needed. Implementation of this last optimization is not trivial.
5 Conclusion

An algorithm in UNITY describes what should be done and not when, where and how it should be done. Transforming UNITY algorithms to PVM turned out to be a process of finding an optimal solution for the when, where and how questions.

Sometimes UNITY algorithms do define when a certain action should be taken. For example the assignment statement

\[ \prod_{i=0}^{N} i, \ A[i] := i + 1, \ B[i] \]

forces a certain order on the assignment of elements from array \( B \) to array \( A \). When implementing such assignments in a different order one has to proof correctness of the new implementation.

Because UNITY does not define where something should be done one has a lot of freedom in choosing the number of tasks one wants to implement for and which variable will be assigned to which task. We saw in our cases that when transforming abstract programs this question has to be solved first. The implementation and the efficiency of the implementation depends heavily on this. In case of the elementwise processable function we choose for an inefficient solution and had to adjust our implementation afterwards to increase efficiency.

After solving the when and where question the how question was rather easy. In case of the associative function we solved this in a formal way. The elementwise processable function showed this could be done also in a more informal way.

We believe that when one would like to make general rules for transforming abstract parallel algorithms into concrete ones UNITY is not a good choice as abstract language. In case one has to use UNITY we would advise to start with a class of algorithms where the when and where questions are already solved.
A Proofs

A.1 Proof of: \((8) \Rightarrow \neg(6) \land \neg(7)\)

\[\forall(i : i \in [1 \ldots n] : t(i) = 2^k(i))\]  \hspace{1cm} (5)

\[(i - 2 \times t(i) + 1 \geq 1) \land (k(i - t(i)) \geq k(i))\]  \hspace{1cm} (6)

\[\left( i - 2 \times t(i) + 1 < 1 \right) \land \left( i - t(i) \geq 1 \right) \land \left( k(i - t(i)) = \left\lfloor \log(i - t(i)) \right\rfloor \right)\]  \hspace{1cm} (7)

\[k(i) = \left\lfloor \log(i) \right\rfloor\]  \hspace{1cm} (8)

To proof: \((8) \Rightarrow \neg(6) \land \neg(7)\)

This proof can be divided into two parts; \((8) \Rightarrow \neg(6)\) and \((8) \Rightarrow \neg(7)\). To proof the first part we will show that \(i - 2 \times t(i) + 1 \geq 1\) \(\equiv False\) with the help of \((8)\) and invariant \((5)\).

\[
i - 2 \times t(i) + 1 \geq 1
\]

\[\equiv\]

\[i \geq 2 \times t(i)\]

\[\equiv\]

\[\{ (5) \}\]

\[i \geq 2 \times 2^k(i)\]

\[\equiv\]

\[\{ (8) \}\]

\[i \geq 2 \times 2^{\left\lfloor \log(i) \right\rfloor}\]

\[\Rightarrow\]

\[\{ i \leq 2^{\left\lfloor \log(i) \right\rfloor} \}\]

\[i \geq 2 \times i\]

\[\equiv\]

\[\{ i \in [1 \ldots n] \}\]

\[False\]

For the second part of the proof it sufficient to show that \(i - t(i) \geq 1\) \(\equiv False\). This goes analogous to the previous proof.

A.2 Proof of: \((13)\)

To proof \((13)\):

\[
\sum_{i=1}^{T} [\log(i)] = \sum_{i=1}^{\left\lfloor \log(T) \right\rfloor} T - 2^{(i-1)}
\]

First we would like to introduce the following lemma:

\[
\sum_{i=1}^{\left\lfloor \log(n+1) \right\rfloor} n - 2^{(i-1)} = \sum_{i=1}^{\left\lfloor \log(n) \right\rfloor} n - 2^{(i-1)} \hspace{1cm} \text{for } n \in \mathbb{N} \land n \geq 2
\]

(35)

It is obvious that lemma \((35)\) is true for the case where \([\log(n+1)] = [\log(n)]\). Rests us to prove that lemma \((35)\) also holds when \([\log(n+1)] = [\log(n)] + 1\).
\[
\sum_{i=1}^{[\log(n+1)]} (n+1) - 2^{(i-1)}
= [\log(n+1)] + \sum_{i=1}^{[\log(n+1)]} n - 2^{(i-1)}
= \{ \text{lemma (35)} \}
[\log(n+1)] + \sum_{i=1}^{[\log(n+1)]} n - 2^{(i-1)}
= \{ \text{induction hypothesis} \}
[\log(n+1)] + \sum_{i=1}^{n} [\log(i)]
= \sum_{i=1}^{n+1} [\log(i)]
\]

Now we will proof (13) with induction. We will take (13) with \( n \) substituted for \( T \) as induction hypothesis. It is obvious that our hypothesis holds for \( n = 1 \). rest us to show that when our hypothesis is true for \( n \), it also holds for \( n + 1 \).
A.3 Proof of: Elementwise processability of the union of sets

\[ \forall s \in \text{range}(x_u^{(i)}) : \forall t \in \text{range}(x_v^{(j)}) : x_u^{(i)}(s) \neq x_v^{(j)}(t) \]  \( (16) \)

\[ \exists i \in \text{Perm}(1, 2, \ldots, r) : \forall u \in [1..k] : x_u = x_u^{(i(1))} ++ x_u^{(i(2))} ++ \ldots ++ x_u^{(i(r))} \]  \( (17) \)

\[ F_i(x^{(1)}) \cup F_i(x^{(2)}) \cup \ldots \cup F_i(x^{(r)}) = F_i(x) \]  \( (18) \)

\[ F_i(x^{(1)}) \cap F_i(x^{(2)}) \cap \ldots \cap F_i(x^{(r)}) = \emptyset \]  \( (19) \)

To proof: \((18) \land (19) \equiv \text{True}\) for all possible total disjoint decompositions of \(x\).

We will use without proof that\(^{28}\)

\[ [x_0 ++ x_1] = [x_0] \cup [x_1] \iff [x_0] \cap [x_1] = \emptyset \]

First we will introduce the following lemma:

\[ F(x_0^{(1)} ++ x_0^{(2)}, x_1^{(1)} ++ x_1^{(2)}) = F(x_0^{(1)}, x_1^{(1)}) \cup F(x_0^{(2)}, x_1^{(2)}) \]  \( (36) \)

Proof of lemma \((36)\):

\[ F(x_0^{(1)} ++ x_0^{(2)}, x_1^{(1)} ++ x_1^{(2)}) = \]

\[ [x_0^{(1)} ++ x_0^{(2)}] \cup [x_1^{(1)} ++ x_1^{(2)}] = \]

\[ [x_0^{(1)}] \cup [x_0^{(2)}] \cup [x_1^{(1)}] \cup [x_1^{(2)}] = \]

\[ F(x_0^{(1)}, x_1^{(1)}) \cup F(x_0^{(2)}, x_1^{(2)}) \]

Now we are able to construct the proof of \((18)\):

\[ F(x) = \]

\[ F(x_0, x_1) = \]

\[ F(x_0^{(1)} ++ x_0^{(2)} ++ \ldots ++ x_0^{(r)}, x_1^{(1)} ++ x_1^{(2)} ++ \ldots ++ x_1^{(r)}) = \{ \text{Lemma (36)} \} \]

\[ F(x_0^{(1)}, x_1^{(1)}) \cup F(x_0^{(2)}, x_2^{(2)}) \cup \ldots \cup F(x_0^{(r)}, x_1^{(r)}) = \]

\[ F(x^{(1)}) \cup F(x^{(2)}) \cup \ldots \cup F(x^{(r)}) \]

Now we are only left with proofing \((19)\):

For \(a, b, c\) and \(d\) sets and \((a \cap c) = (b \cap d) = \emptyset\) we would like to introduce the following lemma:

\[ (a \cup b) \cap (c \cup d) = (b \cap c) \cup (a \cap d) \]  \( (37) \)

\(^{28}\)Remember that repetition is not allowed in sequences.
The proof of lemma (37) goes as follows:

\[
(a \cup b) \cap (c \cup d)
= \{ \text{Distributivity over } \cap \} (a \cup b) \cap c) \cup ((a \cup b) \cap d)
= \{ \text{Distributivity over } \cap \} ((a \cap c) \cup (b \cap c)) \cup ((a \cap d) \cup (b \cap d))
= \{ (a \cap c) = (b \cap d) = \emptyset \} (\emptyset \cup (b \cap c)) \cup ((a \cap d) \cup \emptyset)
= (b \cap c) \cup (a \cap d)
\]

We should proof (19) for all possible total disjoint decomposition. One can make a difference between total disjoint decompositions with 1 block, 2 blocks or more blocks. The case where there is just 1 block is trivial. Let's proof the case where there are 2 blocks. Remark: Remember that the different blocks in a total disjoint decomposition are disjoint (16):

\[
F(x^{(1)}) \cap F(x^{(2)})
= F(x_0^{(1)}, x_1^{(1)}) \cap F(x_0^{(2)}, x_1^{(2)})
= ([x_0^{(1)}] \cup [x_1^{(1)}]) \cap ([x_0^{(2)}] \cup [x_1^{(2)}])
= \{ \text{Remark, Lemma (37)} \} ([x_1^{(1)}] \cap [x_0^{(2)}]) \cup ([x_0^{(1)}] \cap [x_1^{(2)}])
= \{ \text{Remark} \} \emptyset \cup \emptyset
= \emptyset
\]

Left is the case where there are more than 2 blocks in the total disjoint decomposition. Fortunately this is a rather easy proof:

\[
F(x^{(1)}) \cap F(x^{(2)}) \cap F(x^{(3)}) \cap \cdots \cap F(x^{(r)})
= \{ \text{See previous proof} \} \emptyset \cap F(x^{(3)}) \cap \cdots \cap F(x^{(r)})
= \emptyset
\]
A.4 Proof of STEP

We have to proof that for the \textit{STEP} input type with dimension 3 the bag-size is equal to

\[ 3n - n \div 2 - 2 \times (n \div 3) + n \div 6 \]

The formula of Da Silva and Sylvester [Lint 01, p. 115] can be used to count the number of elements in a set which is constructed by the union of smaller subsets. With \( A, A_1, A_2 \) and \( A_3 \) sets with \( A = A_1 \cup A_2 \cup A_3 \) this formula is (\(|A| \) denotes the number of elements in set \( A \)):

\[
|A| = |A_1| + |A_2| + |A_3| - |A_1 \cap A_2| - |A_1 \cap A_3| - |A_2 \cap A_3| + |A_1 \cap A_2 \cap A_3|
\]

It is easy to verify that after \(|A_1| + |A_2| + |A_3|\) the elements in \( A_1 \cap A_2 \), \( A_1 \cap A_3 \) and \( A_2 \cap A_3 \) are counted twice and therefore we should subtract this number from the total. But now the number of elements in \( A_1 \cap A_2 \cap A_3 \) are subtracted twice and therefore they should be added to the total. With the Venn diagram from figure 38 this can be verified easy.

![Venn diagram](image)

Figure 38: Venn diagram

Now let \([x_1] = A_1, [x_2] = A_2\) and \([x_3] = A_3\). Then \(|A_1| + |A_2| + |A_3| = 3n\). Elements in \( A_1 \cap A_2 \) are all the even numbers in \([x_1]\). This are exact \( n \div 2 \) numbers. In order to find \(|A_1 \cap A_3|\) we have to count how many numbers in \( x_2 \) are a multiple of 3. It is easy to see that this are exact \( n \div 3 \) numbers.

For \( A_2 \cap A_3 \) we are searching for every number that can be divided by 2 and 3 in \([x_2]\). Because both 2 and 3 are prime we know that when \( 2 \mid r \) (2 divides \( r \)), for \( r \in \mathbb{N} \), and \( 3 \mid r \) then also \( 2 \times 3 \mid r \). Therefore we are looking for every number that is a multiple of 6. In \([x_2]\) this are \( n \div 3 \) elements.

The same way one can find \(|A_1 \cap A_2 \cap A_3| = n \div 6\).

76
B  Corrections for [Fôth 01]

p 180, Formal approach: In the second paragraph lob(\(i\)) is mentioned as a method of seq. In contrast with what the author claims, it is not defined in [WRMP 01]. The same is the case with the hib(\(i\)) method. One should be aware that the phrases "element \(i\) from the sequence" and "the \(i\)-th element from the sequence" do not have to point to the same element. The notation \(s(i)\) defined as follows

\(s(i)\): the element of \(s\) which is indexed by \(i\) (the \((i - \text{lob}(s))\)-th element of \(s\))

indicates that elements of \(s\) should be indexed in a successive way. The value of lob(s) should not be equal to the index of the first element (this would make the 1-st element the 0-th) but to it’s predecessor.

p 186, first line: The initialization of the variables \(g\) and \(h\) should be \(d_{i(1)}/2\) and \(d_{i(k)}/2\) respectively.

p 190, program 6.1: The use of lov(\(i\)) is not correct and should be replaced by lob(\(i\)) (2x). (The same error exists in invariant (18)) We are looking here for the index of the first element and not for the value of that element.

p 190, program 6.1: Variable \(x_{j+1}\) in the assignment to \(\text{blk}[j+1].f\) has the wrong index. (The same error exists in invariant (18)) The correct assignment is:

\(\text{blk}[j+1].f := x_{j+1}(\text{lov}(x_{j+1}) + \frac{\text{dom}(x_{j+1}) - 1}{2})\)

p 190, HALFVALUE: On page 185, \(f_i\) is defined as \(x_i((s + e)/2)\) where \(s\) the starting and \(e\) the ending index of the appropriate subsequence is. In HALFVALUE, an index is used that is exactly 1/2 bigger\(^{30}\). The correct equation for HALFVALUE is:

\(b(j)[i].f = x_i(b(j)[i].s + \frac{b(j)[i].d - 1}{2})\)

At first sight, this is not a big problem, but when implementing it can change the behavior of the program. The 'old' HALFVALUE should be rounded down. Take for example \(s = 0 \land d = 1\) and it is clear that rounding up to the next integer is giving an index out of range. In the 'new' situation rounding is a choice of the programmer.

p 191, program 6.3: Specification 6.5 tells that \(ub[i].d = bv'[i].d - lb[i].d\). For program 6.3 this means that the assignment should be:

\(ub[i].d := bv[i].d - (u_i - bv[i].s + 1)\)
\(:= bv[i].s + bv[i].d - u_i - 1\)

An other problem is the assignment to \(ub[i].f\). This should be:

\(ub[i].f = x_i((u_i + 1 + bv[i].s + bv[i].d)/2)\)

p 193, specification 6.8: In the invariant one should remove the first occurrence of the close parentheses and \(yv[[i]]\) should be replaced by \(yv'[j]\).

\(^{29}\)It is defined on the function type, not on the sequence type.

\(^{30}\)Remember we choose \(\mathbb{N}\) for IND.
C Programs

Apart from the obvious fact that programs have to be correct one of the main concerns we have when writing them is their readability. Therefore we will stick as close to the abstract counterpart as possible and only perform those optimizations that do not compromise readability. In fact research has shown [Magu 01] that the number of errors in compact C code is significant bigger than 'larger' designed programs. Actually, most intelligent compilers are able to optimize the code in such a way that compact code is not always faster.

Where possible optimizations are performed on expensive operations. There where the readability will be hampered or the shape of the abstract program will be lost this kind of optimization will be omitted.

C.1 Compiling and linking

All presented programs were compiled and run on a Linux and a PowerXplorer system. For both systems the used Makefile is included. Note that for Linux besides the pvm libraries also the math library needs to be included with the -lm option. The values for PVM_ROOT, PVM_ARCH and ARCHLIB are provided by the environment.

Linux:

PRE = cc -g -I$(PVM_ROOT)/include -o
POST = -L $(PVM_ROOT)/lib/$(PVM_ARCH) -lpvm3 -lm $(ARCHLIB)

all:
  make root
  make node

root:
  $(PRE) root root.c $(POST)
  mv root $(HOME)/pvm3/bin/$(PVM_ARCH)

node:
  $(PRE) node node.c $(POST)
  mv node $(HOME)/pvm3/bin/$(PVM_ARCH)

PowerXplorer:

COMP_H = ppx ancc -c
COMP_T = -I$(PVM_ROOT)/include
LINK_H = ppx ancc -o
LINK_T = -L$(PVM_ROOT)/lib/$(PVM_ARCH) -lpvm3

root:
  $(COMP_H) root.c $(COMP_T)
  $(LINK_H) root root.o $(LINK_T)

node:
  $(COMP_H) node.c $(COMP_T)
  $(LINK_H) node node.o $(LINK_T)
C.2 Parallel asynchronous computation of an associative function

We decided to design one extra task to start all the other tasks that will participate in the calculation. Variable \textit{value} should be used to pass the values of \( a_i \).

C.2.1 Root


c\texttt{ root.c }

```c
#include <stdlib.h>
#include <pvm3.h>

#define MSG_INIT 1
#define MSG_READ 2

main()
{
  int nrTasks, info, i;
  int *tids;
  float value;

  printf("Number of Tasks to start: ");
  scanf("%d", &nrTasks);

  tids = malloc((nrTasks+1) * sizeof(int));

  tids[0] = pvm_mytid();
  info = pvm_spawn("node", 0, PvmTaskDefault, ",", nrTasks, &tids[1]);

  if ( info < nrTasks ) printf("Could not start processes.");
  else
    {
      /* Send every process information needed */
      for( i = 1; i < nrTasks; i++ )
        {
          value = i;
          pvm_initsend(PvmDataDefault);
          pvm_pkmint(&i, 1, 1);
          /* Send id, note this is not equal to the taskId */
          pvm_pkmint(&nrTasks, 1, 1);
          pvm_pkfloat(&value, 1, 1);
          /* Value to calculate */
          pvm_pkmint(tids+i, nrTasks, 1);
          pvm_send( tids[i], MSG_INIT );
        }

    free( tids );
    pvm_exit();
    return 0;
}
```

C.2.2 Node

One should rewrite function \( fFunction \) in such a way that \( fFunction(x, y) = f(x \ll y) \). The addresses of the tasks \([1 \ldots nrTasks]\) will be stored in the array \textit{tids} ranging from \([0 \ldots nrTasks - 1]\). In order not to make mistakes with this, no optimization has be done with the indices of variable \textit{tids}.
Array gs can be replaced by a single variable. Result of this is, besides the usage of less memory, that every task uses the same amount of memory. It is also possible to remove variable k. In section C.2.3 these optimizations are done.

```c
#include <stdlib.h>
#include <pvm3.h>
#include <math.h>

#define MSG_INIT 1
#define MSG_READ 2

float cl2(float x) { return ceil(log(x)/log(2.0)); }

float fFunction(float x, float y)
{
    /* Here is the actual function f calculated */
    return x+y;
}

void sendTo(float value, int taskId)
{
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(&value, 1, 1);
    pvm_send(taskId, MSG_READ);
}

float receiveFrom(int taskId)
{
    float value;
    pvm_recv(taskId, MSG_READ);
    pvm_upkfloat(&value, 1, 1);
    return value;
}

float Computation(int i, int nrTasks, int *tids, float value)
{
    int t, k;
    float *gs;
    float result;
    gs = malloc((cl2(i)+1) * sizeof(float));
    /* So */
    gs[0] = value; t = 1; k = 0;
    if (i<nrTasks) sendTo(g0, tids[i+1-1]);
    /* S */
    while (i-2*t+1 >= 1)
    {
        value = receiveFrom( tids[i-t-1]);
        gs[k+1] = fFunction(gs[k], value);
        t *= 2; k++;
        /* Is our result needed on any other task? */
        if (i+t<nrTasks) sendTo(gs[k], tids[i+t-1]);
    }
    /* i-2*t+1 < 1 */
    if (1-t >= 1)
    {
```
value = receiveFrom( tids[1-t-1]);

gs[k+1] = fFunction(gs[k], value);
t += 2; k++;

/* Is our result needed on any other task */
if ((t+i<nrTasks) sendTo(gs[k], tids[i+t-1]);)
}

/* k == ceil(log(t)) */
t += 2;
while (t+i< nrTasks)
{
    /* Sending our result to tasks that need it */
    sendTo(gs[k], tids[i+t-1]);
    t += 2;
}

result = gs[k];
free(gs);
return result;
}

main()
{
    int myTid, myId, myParent, nrTasks;
    float value;
    int *tids;

    myTid = pvm.mytid();
    myParent = pvm.parent();

    /* Receive the information we need to know */
    pvm_recv(myParent, MSG.INIT);
    pvm.upkint( &myId, 1, 1);
    pvm.upkint( &nrTasks, 1, 1);
    pvm.upkfloat( &value, 1, 1);
    tids = malloc((nrTasks) * sizeof(int));
    pvm.upkint( tids, nrTasks, 1);

    /* Ready to start computation */
    value = Computation(myId, nrTasks, tids, value);
    printf("\nResult is: %d : %f\n", myId, value);

    free(tids);
    pvm.exit();
    return 0;
}

C.2.3 A more compact solution

Because only function Computation in node.c changed, the other code will not be reprinted. Note that variable k is removed and that array gs is replaced by a single float variable.

procedure Computation

float Computation(int i, int nrTasks, int *tids, float value)
{
    int t;
    float gs;

C.3 Elementwise Processing

C.3.1 Declarations

decs.h

/*
 * Ewp/decs.h
 *
 * 2000-10-06 v 0.12
 */
#include <stdlib.h>
#define bool int
#define TRUE 1
#define FALSE 0
#define K 3
#define L 1

/* Message Tags */
#define MSG_BLOCKPART 1
#define MSG_UPPER_BLK 2
#define MSG_LOWER_BLK 3
#define MSG_CUT_BLK 4
#define MSG_EP 5
#define MSG_IDLE 6
#define MSG_TIDS 7
#define MSG_REQUEST 8
#define MSG_COLLECT 9
#define START_TRANSFER 10
#define MSG_TRANSFER 11
C.3.2 Datatypes

datatypes.c

/*
 * Bup/datatypes.c
 *
 * 2000-10-06 v 0.22
 */

#include <stdlib.h>

#define MAXH 2147483647 /* depends on type of H; value depends
 * on system; this is MAXINT, INT_MAX
 * or MAX_INT */

typedef int H;     /* Type depends on elements to compute
 * when changing this, pmu.pkH and pmu.upkH
 * should be changed also */
typedef int IND;   /* Not used in further program
 * just mentioned to be complete */
typedef struct TBlockpart {
    int s;        /* starting index of blockpart */
    int d;        /* number of elements in blockpart */
    H f;         /* mean-value of blockpart */
} BLOCKPART;
typedef BLOCKPART BV[K];
typedef struct TBvs {
    BV seq;
    int dom;   /* Number of elements linked after this point */
    struct TBvs *next;
} TBVS;
typedef TBVS *BVSV;

typedef BVSV BVSP[K];    /* Not used in implementation */

typedef struct Tseqh {
    int size;     /* maximum number of elements in sequence */
    int lo;       /* lowest index of existing elements in sequence */
    H *seqh;
} SEQH;

typedef struct Tx {
    int bagSize;
    SEQH rows[K];
} X;

typedef struct Tp {
    H h;
    struct Tp *next;
} P;
typedef struct Tpl
{
    int size;
P *set;
} PL;

typedef PL Y[L];

typedef Y Y[V[K]; /* Not used in implementation */
/
*******************************************************************************/
void SEQH.init(SEQH *seqH, int size);
/
*******************************************************************************/

int pvm.pkH( H *h, int nrItems, int stride)
{
    int info;
    info = pvm.pkint( h, nrItems, stride);
    return info;
} /* pvm.pkH */

int pvm.upkH( H *h, int nrItems, int stride)
{
    int info;
    info = pvm.upkint( h, nrItems, stride);
    return info;
} /* pvm.upkH */

int pvm.pkSEQH( SEQH *seqH, int startPosition, int size)
{
    int info;
    info = pvm.pkint( &size, 1, 1);
    if ( size > 0 )
    {
        info += pvm.pkH( &(*seqH).seqh[startPosition], size, 1);
    }
    return info;
} /* pvm.pkSEQH */

int pvm.upkSEQH( SEQH *seqH)
/* pre: seqH is uninitialized */
{
    int info, size;
    info = pvm.upkint( &size, 1, 1);
    SEQH.init( seqH, size);
    if ( size > 0 )
    {
        info += pvm.upkH( (*seqH).seqh, size, 1);
    }
    (*seqH).lob = 0;
    return info;
} /* pvm.upkSEQH */

int pvm.pkBLOCKPART( BLOCKPART *blockpart, int nrItems, int stride)
{
    int i, info = 0;
    for (i=0; i<(nrItems*stride); i += stride)
    {
        info += pvm.pkint( &blockpart[i].s, 1, 1);
        info += pvm.pkint( &blockpart[i].d, 1, 1);
        info += pvm.pkH( &blockpart[i].f, 1, 1);
    }
}
return info;
} /* pem_pkBLOCKPART */

int pem.upkBLOCKPART( BLOCKPART *blockpart, int nritems, int stride)
{
    int i, info = 0;
    for (i=0; i<nritems*stride; i += stride)
    {
        info += pem.upkint( &blockpart[i].s, 1, 1);
        info += pem.upkint( &blockpart[i].d, 1, 1);
        info += pem.upkH( &blockpart[i].f, 1, 1);
    }
    return info;
} /* pem_upkBLOCKPART */

void BVS.init(BVS *bvs)
{
    *bvs = NULL;
} /* BVS.init */

void BVS.free(BVS *bvs)
{
    BVS bvsUse;
    while (*bvs != NULL)
    {
        bvsUse = (*bvs) -> next;
        free(bvs);
        *bvs = bvsUse;
    }
} /* BVS.free */

void BVS.loext(BVS *bvs, BV bv)
/* post: bus == <<bv>> ++ bus */
{
    int i;
    BVS bvsNew;
    bvsNew = (TBVS *) malloc( sizeof( TBVS ));
    bvsNew -> next = *bvs;
    bvsNew -> dom = BVS.dom( *bvs ) + 1;
    for ( i=0; i<K; i++) bvsNew -> seq[i] = bv[i];
    *bvs = bvsNew;
} /* BVS.loext */

void BVS.lorem( BVS *bvs)
/* bvs' == << a.0, a.1,..., a.r >> */
/* bus == << a.1,...,a.r >> */
{
    BVS bvsUse;
    if (BVS.dom( bvs ) != 0)
    {
        bvsUse = *bvs;
        *bvs = bvsUse -> next;
        free(bvsUse);
    }
} /* BVS.lorem */

void BVS.lolv(BVS bvs, BV bv)
/* pre: dom(bvs) != 0 */
/* post: bv = lov(bvs) */
{
    int i;
}
for ( i=0; i<K; i++ ) bv[i] = bvs -> seq[i];
} /* BVs:lov */

int BVs.dom(BVS bvs)
/*@ ret: dom(bvs) */
{  
  if ( bvs == NULL ) return 0;
  else return bvs -> dom;
} /* BVs:dom */

void SEQH_init(SEQH *seqH, int size)
{
  (*seqH).size = size;  /* size of the used fixed-array */
  (*seqH).lob = size;  /* sequence is empty initially */
  if (size != 0)
    (*seqH).seqh = (H*) malloc(size * sizeof(H));
  else
    /* just in case seqh should be empty */
    /* What does malloc(0) ?? */
    (*seqH).seqh = NULL;
} /* SEQH:init */

void SEQH_free(SEQH *seqH)
{
  free((*seqH).seqh);
  (*seqH).size = 0;
  (*seqH).lob = 0;
} /* SEQH:free */

void SEQH_fill(SEQH *seqH)
/*@ At this moment, this is just filled
* program-code only used for testing */
{
  int i, j;

  (*seqH).lob = 0;
  srand(pvm_mytid());
  (*seqH).seqh[0] = (int) (K*rand()/(RAND_MAX+1.0));
  for (i=1; i<(*seqH).size; i++)
    {  
      j = (int) ((float)K*rand()/(RAND_MAX+1.0));
      /* +1, because needs to be 1-1 mapping */
    }
} /* SEQH:fill */

int SEQH_lob(SEQH seqH)
{
  return seqH.lob;
} /* SEQH:lob */

int SEQH_dom(SEQH seqH)
{
  return (seqH.size - seqH.lob);
} /* SEQH:dom */

H SEQH_val(SEQH seqH, int i)
{
  if (seqH.size <= i) printf("Index out of bound in SEQH_val\n");
  return seqH.seqh[i];
} /* SEQH:val */

H SEQH_lsb(SEQH seqH)
{
  return seqH.seqh[seqH.lob];
}
void SEQH_lorem(SEQH *seqH)
/* pre: dom(seqH) != 0 */
{
    (*seqH).lob++;
} /* SEQH_lorem */

int SEQH_search(SEQH seqH, int s, int d, H h)
/* pre: d > 0 */
+ s == S, d == D
+ seqH[s] <= h < seqH[s + d]
+ We define seqH[s + d] == inf
+ seqH[s-1] == -inf
+ post: S <= s < S + D
+ seqH[s] <= h < seqH[s+1]
*/
{
    int half;
    
    if (seqH.size < d) printf("Index out of bound in SEQH_search\n");
    /* binary search */
    d += s;
    s =--;
    
    while (s+1 != d)
    {
        half = (d+s) / 2;
        if (seqH.seqH[half] <= h) s = half;
        else d = half;
    }
    
    return s;
} /* SEQH_search */

void SEQH_copy( SEQH *destination, SEQH source, int startPosition, int size)
/* pre: destination == NULL */
{
    int i;
    
    SEQH_init( destination, size);
    (*destination).lob = 0;
    for (i=0; i<size; i++)
        (*destination).seqH[i] = source.seqH[i+startPosition];
} /* SEQH_copy */

void SEQH_move( SEQH *destination, SEQH *source)
/* post: source is uninitialized */
{
    *destination = *source;
    source = NULL;
} /* SEQH_move */

void X_init(X **x)
{
    int i;
    
    for (i=0; i<K; i++) SEQH_init(&(*x).rows[i], 0);
    (*x).bagSize = 0;
} /* X_init */

void X_free(X **x)
{
    int i;
    
    for (i=0; i<K; i++) SEQH_free(&(*x).rows[i]);

87
(**x).bagSize = 0;
} /* X.free */

void X.moveSeqH(X *x, SEQH *seqH, int i)
/*
 * pre: 0 <= i < K
 * post: seqH == NULL, z[i] == seqH'
 */
{
    if (SEQH.dom((**x).rows[i]) != 0 )
        printf("X not empty on i %d\n", i);
    else {
        (**x).bagSize += SEQH.dom(seqH);
        SEQH.move( &(**x).rows[i], seqH);
    }
} /* X.moveSeqH */

void X.copySeqH(X *x, SEQH seqH, int i, int startPosition, int size)
/*
 * pre: 0 <= i < K, z[i] == empty
 * post: z[i] == seqH[startPosition .. startPosition + size - 1]
 */
{
    if (SEQH.dom((**x).rows[i]) != 0 )
        printf("X not empty on i %d\n", i);
    else {
        (**x).bagSize += size;
        SEQH.copy( &(**x).rows[i], seqH, startPosition, size );
    }
} /* X.copySeqH */

int X.dom(X x)
{
    return x.bagSize;
} /* X.dom */

H X.lov( X x)
/*
 * pre: x.bagSize>0 */
{
    int i;
    H j, h = MAX_H,
    for (i=0; i<K; i++)
    {
        if (SEQH.dom(x.rows[i]) > 0 )
            { j = SEQH.lov(x.rows[i]);
              if ( j < h) h = j;
            }
    }
    return h;
} /* X.lov */

bool X.lorem( X x, int i, H h)
/*
 * pre: 0 <= i < K, for all g in z : h <= g
 * ret : true  ==> h' + + z[i] == z[i]'
 * false  ==> z == z' \ h != lov(z[i])
 */
{
    if ( (SEQH.dom((**x).rows[i]) != 0 ) & & ( SEQH.lov((**x).rows[i]) == h) )
        { SEQH.lorem((**x).rows[i]);
          (**x).bagSize -= 1;
          return TRUE;
        }
    else return FALSE;
} /* X.lorem */

void Y.init(Y *y)
```c
{ int i;
    for( i=0; i<L; i++)
    {
        (*y)[i]. size = 0;
        (*y)[i]. set = NULL;
    }
} /* Y.init */

H Y.removeH(Y *y, int i)
/* pre: 0 <= i < L */
{ H value;
    P *remove;
    value = (*y)[i]. set -> h;
    remove = (*y)[i]. set;
    (*y)[i]. set = (*y)[i]. set -> next;
    free(remove);
    (*y)[i]. size --;
    return value;
} /* Y.removeH */

void Y.free(Y *y)
{ int i;
    for (i=0; i<L; i++)
        while ((y[i]. size > 0)
            Y.removeH(y, i);
} /* Y.free */

void Y.addH(Y *y, int i, H h)
/* pre: 0 <= i < L /\ h not in y[i]
* post: y[i] == y[i]+ h */
{ P *newP;
    newP = (P*) malloc( sizeof( P ));
    (*newP).h = h;
    (*newP).next = (*y)[i]. set;
    (*y)[i]. set = newP;
    (*y)[i]. size ++;
} /* Y.addH */

/*
 * The functions pvm.pkY and pvm.ipkY andAdd can be more efficient when
 * one renames them so that not L-times memory needs to be allocated
 */

int pvm.pkY(Y *y)
{ int i, j, size, info = 0;
    H *hArray;
    struct Tp *nextElement;
    for (i=0; i<L; i++)
    {
        size = (*y)[i]. size;
        info += pvm.pkint( &size, 1, 1);
        hArray = (H*) malloc( size * sizeof( H ));
        nextElement = (*y)[i]. set;
        ```
while (nextElement != NULL)
{
    (hArray)[j] = nextElement -> h;
    nextElement = nextElement -> next;
}
pvm.pkH( hArray, size, 1);
free( hArray);
}
return info;
} /* pvm.pkY */

int pvm.upkY_andAdd( Y *y)
{
    int i, size, info = 0;
    H *hArray;

    for (i=0; i<L; i++)
    {
        info += pvm.upkint( &size, 1, 1);
        hArray = (H*) malloc( size * sizeof( H));
        info += pvm.upkH( hArray, 1, 1);
        while (size>0)
        {
            Y_addH( y, i, hArray[size-1]);
            size --;
        }
        free( hArray);
    }

    return info;
} /* pvm.upkY_andAdd */

C.3.3 Master

master.c

/*
 *  Eup/root.c
 *  ==> master.c
 *
 *  2000-08-30 v 0.22
 */

#include <pvm3.h>
#include <math.h>
#include "dec.h"
#include "datatypes.c"

int Opt(int N)
{
    return N/(K+1);
    // return N/pow(K,2);
} /* Opt */

H CutValue(BV bv)
{
    int permutation[K];
    int p, q, r;
    float g, h;

    90
/* Sorting */
for (p=0; p<K; p++) permutation[p] = p;

/* For small K, this solution is the most efficient measured in time
* and in memory usage. */
for (p=0; p<K-1; p++)
    for (q=p+1; q<K; q++)
        if (bv[permutation[p]].f > bv[permutation[q]].f)
            r = permutation[p];
            permutation[p] = permutation[q];
            permutation[q] = r;

/* Middle */
p = 0;
q = K-1;
g = bv[permutation[0]].d/2;
h = bv[permutation[K-1]].d/2;
while (p<q)
    { if (g<h)
        
            g += bv[permutation[p]].d/2.0 + bv[permutation[p+1]].d/2.0;
            p++;
        
    
    else
        
            h += bv[permutation[q]].d/2.0 + bv[permutation[q-1]].d/2.0;
            q--;
        
    }
    return bv[permutation[p]].f;

} /* CutValue */

int ReadBlockpart( int tid, int msgTag, BLOCKPART *blockpart)
/* ret: id of task message received from. */
{
    int id;

    pvm_reccv( tid, msgTag);
    pvm_upkint( &id, 1, 1);
    pvm_upkBLOCkPART( blockpart, 1, 1);

    return id;
} /* ReadBlockpart */

void SendBlock( int tid, int msgTag, BV bv, int nItems)
{
    pvm_initsend( PvmDataDefault);
    pvm_pkBLOCkPART( bv, nItems, 1);
    pvm_send( tid, msgTag);
} /* SendBlock */

int ReadBlockVector( BV bv, int msgTag)
/* ret: bag.size
* part: bv[i] == blockpart received from task i */
{
    BLOCKPART blockpart;
    int id, j, N=0;

    for (j=0; j<K; j++)
    {
        id = ReadBlockpart( -1, msgTag, &blockpart);
        bv[id] = blockpart;

    
}
N += blockpart.d;
}
return N;
} /* ReadBlockVector */

void SendCutInfo(BV ub, H h, int *tids)
{
    int j;
    for (j=0; j<K; j++)
    {
        ub[j].f = h;
        SendBlock(tids[j], MSG.CUT.BLK, &ub[j], 1);
    }
} /* SendCutInfo */

int whichTaskIsIdle()
{
    int buffid, taskId, tmp1, tmp2;
    buffid = pvm.recv(-1, MSG.IDLE);
    pvm.bufinfo(buffid, &tmp1, &tmp2, &taskId);
    /* ?? Do we need to free this message buffer? */
    return taskId;
} /* whichTaskIsIdle */

main()
{
    int nProcessingSlaves, idlId, info, tids[K+1];
    int B, N, bagSizeUb, bagSizeLb;
    H h;
    BV blk, ub, lb;
    BVS bb, b;
    Y y;
    tids[0] = pvm.mytid();
    info = pvm.spaun("node", 0, PvmTaskDefault,"", K, &tids[1]);
    // info = pvm.spaun("node", 0, PvmTaskDefault+PvmTaskDebug, ",", K, &tids[1]);

    if (info<K) printf("Could not start processes.");
    else
    {
        /* init some variables */
        BVS.init(&bb);
        BVS.init(&b);

        /* Distribute all the tids */
        pvm.initsend(PvmDataDefault);
        pvm.pkinit(&tids[1], K, 1);
        pvm.mcast(&tids[1], K, MSG.TIDS);

        /* Program 6.1 */
        N = ReadBlockVector(blk, MSG.BLOCKPART);
        B = Opt(N);
        /* Do we need to cut this block? */
        if (N>B) BVS.loext(&bb, blk); else BVS.loext(&b, blk);

        /* Program 6.2 */
        while (BVS.dom(bb) != 0)
        {
            /* CutValue */
            BVS.Jov(bb, blk);
            h = CutValue(blk);

            /* Cut */
            SendCutInfo(blk, h, &tids[1]);
    
    92
bagSizeLb = ReadBlockVector(lb, MSG.LOWER.BLK);
bagSizeUb = ReadBlockVector(ub, MSG.UPPER.BLK);

    /* Update */
    BVS.Lorem(&bb);
    if (bagSizeLb>B) BVS.Joext(&bb, lb); else BVS.Joext(&b, lb);
    if (bagSizeUb>B) BVS.Joext(&bb, ub); else BVS.Joext(&b, ub);

}  /* Program 6.4 */

nrProcessingSlaves = 0;
while (BVS.dom(b) != 0)
{
    if (nrProcessingSlaves < K)
    {
        idlId = tids[nrProcessingSlaves+1];
        nrProcessingSlaves++;
    }
    else
    {
        idlId = whichTaskIsIdle();
        BVS.Jov(b, blk);
        BVS.Lorem(&b);
        SendBlock(idlId, MSG.EP, &blk[0], K);
    }

/* There will be nrProcessingSlaves messages with tag MSG.IDLE
 * that we did not read. */
    BVS.free(&b);

    /* Program 6.5 */
    Y.init(&y);

    while (nrProcessingSlaves>0)
    {
        pvm_recv(-1, MSG.IDLE);  /* Clean-up the MSG.IDLE's */
        pvm_recv(-1, MSG.TRANSFER);
        pvm_upkY_andAddl(&y);
        nrProcessingSlaves--;
    }

/* end program */
    Y.free(&y);

    pvm.exit();
    return 0;
}  /* main */

C.3.4 Slave

slave.c

/*
 * Eup/node.c
 * == slave.c
 * *
 * 2000-10-06 v 0.26
 */

93
*/
#include <pvm3.h>
#include "decs.h"
#include "datatypes.c"

void AnswerRequest(int tid, SEQH seqH, int myId)
{
    int s, d;
    pvm_upkint(&s, 1, 1);
    pvm_upkint(&d, 1, 1);
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&myId, 1, 1);
    pvm_pkSEQH(&seqH, s, d);
    pvm_send(tid, MSG_COLLECT);
} /* AnswerRequest */

int nextMessage(SEQH seqH, int myId)
{
    static int nrBlocks = 1; /* We start with an uncult block */
    static int nrCollect = 0; /* we didn't collect messages yet */
    static int msgTag; /* last read msgTag */
    int buflid, tid, tmp;

    if (myId == -1)
    {
        /* Return last msgTag */
        return msgTag;
    }
    else
    {
        if (nrBlocks > 0)
        {
            buflid = pvm_recv(-1, -1);
            pvm_bufinfo(buflid, &tmp, &msgTag, &tid);
            switch (msgTag)
            {
            case MSG_REQUEST:
                /* An other task is processing a block */
                AnswerRequest(tid, seqH, myId);
                nrBlocks--;
                break;
            case MSG_COLLECT:
                /* Data is ariving that we requested */
                nrCollect++;
                if (nrCollect == K-1)
                {
                    /* All needed information is collected */
                    nrBlocks--;
                    nrCollect = 0;
                }
                break;
            case MSG_CUT_BLK:
                /* We need to cut a block, so there will be 1 more block */
                nrBlocks++;
                break;
            case MSG_EP:
                if (K==1)
                {
                    /* No need to request data, so decrease nrBlocks */
                    nrBlocks--;
                }
                break;
            }
        }
    }}
void SendBlockpart(int myId, int tid, int msgTag, BLOCKPART blockpart)
{
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&myId, 1, 1);
    pvm_pkBLOCKPART(&blockpart, 1, 1);
    pvm_send(tid, msgTag);
} /* SendBlockpart */

void RequestData(int *tids, BV bv, int myId)
{
    int i;

    for (i=0; i<K; i++)
    {
        if (i != myId)
        {
            pvm_initsend(PvmDataDefault);
            pvm_pkint(&bv[i].s, 1, 1);
            pvm_pkint(&bv[i].d, 1, 1);
            pvm_send(tids[i], MSG_REQUEST);
        }
    }
} /* RequestData */

void CollectData(X *x, BV bv, SEQH seqH, int myId)
{
    int i, id;
    SEQH s;

    for (i=0; i<(K-1); i++)
    {
        nextMessage(seqH, myId);
        pvm_upkint(&id, 1, 1);
        /* s is uninitialized */
        pvm_upkSEQH(&s);
        /* s is initialized and contains a seq(H) */
        X_moveSeqH(x, &s, id);
        /* s is uninitialized */
    }

    /* Store the data we own */
    X_copySeqH(x, seqH, myId, bv[myId].s, bv[myId].d);
} /* CollectData */

H F(H e, int i, bool *sl)
{
    /* calculate F.i */
    return e; /* The Elementwise Processable Function */
} /* F */

void EP(Y *y, X **x)
/* Program 3.1 */
{
    H e, fe;

    95
bool s[K];
int i;

while ( X.dom( *x ) != 0 )
{
    e = X.jou( *x );

    for ( i=0; i<K; i++ )
        if ( X.lore( x, i, e ) )
            /* e in x'.i */
            sl[i] = TRUE;
        else
            /* e not in x'.i */
            sl[i] = FALSE;

    for ( i=0; i<L; i++ )
        {  
            fe = F( e, i, sl );
            Y.addH( y, i, fe );
        }
} /* EP */

main()
{
    int myTid, myParent, tids[K], myId = 0;
    int j;
    SEQH seqH;
    BLOCKPART blockpart, newBlockpart, emptyBlockPart = { 0, 0, 0 };
    Y yv;
    BV bv;
    X x;

    myTid = pvm.mytid();
    myParent = pvm.parent();

    /* init some variables */
    SEQH.init(&seqH, maxNrx);
    /* Provide initial values */
    SEQH.fill(&seqH);

    /* Receive all the tids */
    pvm.recv( myParent, MSG.TIDS);
    pvm.uptim( tids, K, 1 );

    /* Find myId, myId == 0 */
    while ( tids[myId] != myTid ) { myId++;

    /* Program 6.1 */
    blockpart.s = 0;
    blockpart.d = SEQH.dom( seqH );
    blockpart.f = SEQH.val( seqH, blockpart.s + (blockpart.d - 1)/2 );
    SendBlockpart( myId, myParent, MSG.BLOCKPART, blockpart );

    /* Program 6.3 */
    while ( nextMessage( seqH, myId ) == MSG.CUT_BLK )
    {
        pvm.upkBLOCKPART( &blockpart, 1, 1 );

        if ( blockpart.d > 0 )
            j = SEQH.search( seqH, blockpart.s, blockpart.d, blockpart.f);
        else
            j = -1;

        /* Construct lowerBlockpart */
        if ( ( j - blockpart.s + 1 ) > 0 )

96
{  
  newBlockpart.s = blockpart.s;
  newBlockpart.d = j - blockpart.s + 1;
  newBlockpart.f = SEQH.val( seqH, (blockpart.s + j) / 2);
  SendBlockpart( myId, myParent, MSGLOWER_BLK, newBlockpart);
}
else
  SendBlockpart( myId, myParent, MSGLOWER_BLK, emptyBlockpart);

/* Construct upperBlockpart */
if( j+1 < blockpart.s + blockpart.d )
{
  newBlockpart.s = j + 1;
  newBlockpart.d = blockpart.s + blockpart.d - j - 1;
  newBlockpart.f = SEQH.val( seqH,
                              ( j + blockpart.s + blockpart.d + 1 ) / 2);
  SendBlockpart( myId, myParent, MSGUPPER_BLK, newBlockpart);
}
else
  /* upperBlockpart is empty */
  SendBlockpart( myId, myParent, MSGUPPER_BLK, emptyBlockpart);
}

/* Program 6.4 */
if( nextMessage( seqH, -1 ) == MSG_EP )
{
  Y.init(&yv);
  X.init(&xx);

  do
  {
    pvm.upkBLOCKPART( bv, K, 1);

    /* request needed data */
    RequestData( tids, bv, myId);

    /* collect needed data */
    CollectData( kx, bv, seqH, myId);

    /* yv = EP( yv, z, bv ) */
    EP( &yv, z, &v);

    pvm.initSend(PvmDataDefault);
    pvm.send( myParent, MSG_IDLE);
  }
  while( nextMessage(seqH, myId) == MSG_EP );

  nextMessage == START_TRANSFER */
  X.free(&xx);

  /* Program 6.5 */
  pvm.initSend( PvmDataDefault);
  pvm.pkY( &yv);
  pvm.send( myParent, MSG_TRANSFER);

  /* Stop program */
  Y.free(&yv);
}

SEQH.free(&seqH);

pvm.exit();
return 0;
} /* main */
C.3.5 Optimalization

- Verhaaltje over wat dit is -

optimalization.c

```c
int nextMessage(SEQH seqH, int myId)
{
    static int nrBlocks = 1;  /* We start with an uncut block */
    static int nrCollect = 0;  /* we didn't collect messages yet */
    static int msgTag;
        /* the last tag we got */
    int bufid, tid, tmp;

    if (myId == -1)
    {
        /* What was the last msgTag? */
        return msgTag;
    }
    else if (myId == -2)
    {
        /* Slave answered a request on his own */
        nrBlocks--;
        return 0;
    }
    else
    {
        if (nrBlocks > 0)
        do
            {
                bufid = pvm_recv(-1, -1);
                pvm_buffer(bufid, &tid, &msgTag, &tid);
                switch (msgTag)
                {
                case MSG.REQUEST:
                    /* An other task is processing a block */
                    AnswerRequest(tid, seqH, myId);
                    nrBlocks--;
                    break;
                case MSG.COLLECT:
                    /* Data is arriving that we requested */
                    nrCollect++;
                    if (nrCollect == K - 1)
                    {
                        /* All needed information is collected */
                        nrBlocks--;
                        nrCollect = 0;
                    }
                    break;
                case MSG.CUT_BLK:
                    /* We need to cut a block, so there will be 1 more block */
                    nrBlocks++;
                    break;
                case MSG.REF:
                    if (K == 1)
                    {
                        /* No need to request data, so decrease nrBlocks */
                        nrBlocks--;
                    }
                    break;
                }
            } while ((msgTag == MSG.REQUEST) && (nrBlocks > 0));

    if (nrBlocks == 0)
        return START_TRANSFER;
    else
```

98
return msgTag;
}

} /* nextMessage */

void EP( Y *y, X *x, SEQH seqH, int myId)
/ * Program 3.1, optimised version */
{
    H c, fc;
    bool sl[K];
    int i;
    int msgTag, bufd, tmp, tid;

    while ( X.dom(*x) != 0 )
    {
        e = X.Jov(*x);

        for ( i=0; i<K; i++)
            if ( X.Jorem( x, i, e) )
                /* e in x \cdot i */
                sl[i] = TRUE;
            else
                /* e not in x \cdot i */
                sl[i] = FALSE;
        /* e not in x */

        for ( i=0; i<L; i++)
            { fe = F( e, i, sl); 
            Y.addH( y, i, fe); }

        /* Is there a request for data */
        bufd = pvm.recv( -1, -1);
        if ( bufd > 0 )
            { /* Yes, there is a request for data */
            pvm.binfof( bufd, &tmp, &msgTag, &tid);
            AnswerRequest(tid, seqH, myId);
            /* Tell nextMessage() we answered a request */
            nextMessage(seqH, -2);
            }
    }
} /* EP */
References

[Chan 01] K.M. Chandy / J. Misra
    Parallel Program Design - A Foundation
    1988, Addison-Wesley

[Corm 01] T.H. Cormen / C.E. Leiserson / R.L. Rivest
    Algoritmusok
    Original English version: Introduction to Algorithms

[Dijk 01] E.W. Dijkstra
    A Discipline of Programming 1976, Prentice-Hall

[Feij 01] W.H.J. Feijen / A.J.M. van Gasteren
    Lecture notes - Design of Multiprograms
    AvG 122 / WF 208 - Eindhoven, 12 May 1995

[Fost 01] Foster
    Designing and Building of parallel programs
    http://www.mcs.anl.gov/dbpp/

[Fóth 01] Ákos Fóthi / Zoltán Horváth / Tamás Kozsik
    Parallel Elementwise Processing - A Novel Version
    Appeared in [Varg 01, p 180-194]

[Greg 01] Tibor Gregorics / Sándor Sike
    Weak Elementwise Processing
    Appeared in [Varg 01]

[Horv 01] Zoltán Horváth
    Parallel asynchronous computation of the values of an associative function
    Acta Cybernetica, Vol. 12, No. 1, Szeged, 1995

[Kald 01] Anne Kaldewaij
    Programming: The Derivation of Algorithms
    1990, Prentice-Hall - ISBN 0 13 204108 1

[Lint 01] Prof. dr. J.H. van Lint / Dr. J.W. Nienhuys
    Discrete Wiskunde
    1991, Academic Service - ISBN 90 6233 368 0

[Magu 01] Maguire, Stephan A.
    Writing Solid Code
    1993, Microsoft Press

[Misr 01] Jayadev Misra
    More on Strengthening the Guard
    notes on UNITY 19-20, July 17, 1990

[Pars 01] Parsytec Gmbh
    PowerPVM: Design and Implementation Description [design.ps]
    PowerPVM: Installation and Use [install.ps]
    Both and more documents can be found with the distribution of PowerPVM and at
    http://www.parsytec.de/

[PVM 01] Oak Ridge National Laboratory
    PVM: Parallel Virtual Machine
    http://www.epm.ornl.gov/pvm/
[PVM 02] Massachusetts Institute of Technology  
*PVM: A Users' Guide and Tutorial for Networked Parallel Computing*  

[PVM 03] Oak Ridge National Laboratory  
*PVM 3 user's guide and reference manual*  
ORNL/TM-12187  
http://www.netlib.org/pvm3/ug.ps

[Rice 01] John A. Rice  
*Mathematical Statistics and Data Analysis*  

[Shiv 01] Eötvös Loránd University of Budapest  
*Shiva: Az augusztákos kapcsolt Parsytec PowerXplorer gép*  
http://valerie.inf.elte.hu/shiva/

[Sips 01] Michael Sipser  
*Introduction to the Theory of Computation*  

[Sund 01] V.S. Sunderam  
*PVM: A framework for parallel distributed computing*  

[Varg 01] László Varga (ed)  
*Proceedings on the Fourth Symposium of Programming Languages and Software Tools (Visegrád, Hungary)*  
1995, Eötvös Loránd University of Budapest - ISBN 963 463 013 8

[Wort 01] Leon A. Wortman / Thomas O. Sidebottom  
*C, The Programming Tutor*  

[WRMP 01] Workgroup on Relational Models of Programming  
*Some Concepts of a Relational Model of Programming*  
Appeared in [Varg 01, p 434-445]