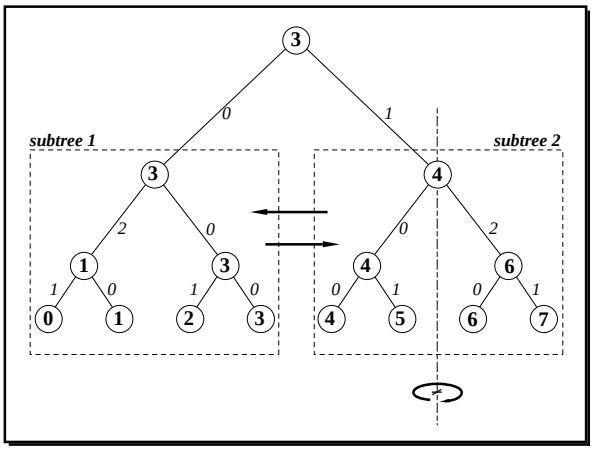
# The use of a tree topology on a distributed memory system for the implementation of the Conjugate Gradient method




Arian Paul Wals

Master's Thesis in Mathematics
Supervisor: Dr. W. Hoffmann

Faculty of Mathematics and Computer Science
University of Amsterdam
October 1994

## Introduction

In many cases large linear systems Ax=b are solved by using an iterative method. Iterative methods are often much cheaper than direct methods: iterative methods are of order $n^2$ times the number of iterations, direct methods are of order $n^3$. A nice property of iterative methods is that they are quite easily to parallelise. The parallel operations that are needed are: the matrix-vector product, the innerproduct and for some methods the transposed matrix-vector product. For the innerproduct and, in case of dense matrices, also for the matrix-vector product, the important communication routines are gathering data from a set of processors in one processor and sending data from one processor to a set of processors (broadcasting). These routines are each others reverse so the resulting algorithms are very similar.

Consider broadcasting over p processors. The optimal routine is the one that in each communication step doubles the number of processors that contain the data. The total number of steps needed for this operation is $^2$logp. In this paper a topology will be presented that is specifically written for such a communication scheme. This topology will be used in the matrix-vector product that is part of an implementation of the conjugate gradients (CG) method and for an implementation of the innerproduct. The CG method is implemented for dense matrices, uniformly sparse matrices (using sparse BLAS) and band matrices. A comparison is made with traditional topologies like a grid and a torus. All tests have been done on the parsytec GCel that is located at SARA.

# 1 The Parsytec GCel

We have used the parsytec GCel that is located at SARA. The GCel is a distributed memory system with 512 processors. A short description of the hard- and software is given in the next two paragraphs. In the third paragraph the performance will be examined.

## 1.1 The hardware

The GCel is a machine with 32 clusters of 16 nodes which results in a total of 512 processors. The nodes are situated in a square mesh of 16 by 32. Each node contains a T805 transputer from Inmos and 4 Mbyte of memory.
The machine is split up in several partitions that can be allocated by users. A partition consists of a number of clusters. This induces that a partition of one cluster is the smallest number of processors that can be allocated. Therefore one cluster of 16 nodes is called an atom. Figure 1.1 shows an example of a partitioning of a GCel with 64 nodes.
The T805 has 4 Kbyte of on chip ram. Part of this ram can be used as very fast stack memory. We will show in paragraph 1.3 that the use of this ram can speed up the performance of some routines.

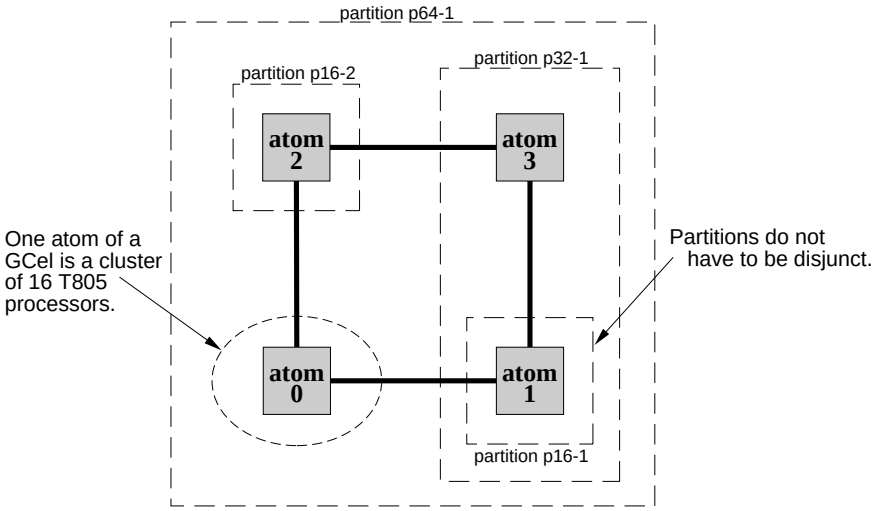


Figure 1.1

## 1.2 The software

The software environment of the parsytec is parix. A very powerful feature of parix is the support for virtual topologies. For the same set of processors a torus, a pipeline and a ring topology can be used all at the same time. This gives one the ability to use the optimal topology for each subroutine. The virtual topology feature is meant for use with the parsytec GC, but is for compatibility also implemented on the GCel. The GC has special hardware to take care of the communication, virtual links will then be almost as fast as local links. This does not hold for the GCel and the virtual topology library should be used with care. Other interesting features are parmacs: a portable programming model and patop: an interactive performance analyser.

## 1.3 The performance of the T805

The T805 is not very fast and the compiler doesn't do very much optimising too. For most routines the speed will not exceed 0.45 Mflops. Unfortunately there's only one optimised library available: BLAS level 1, but parix provides two special features for speeding up the computations on a single processor that will be discussed below:

- *As mentioned before, parix supports the use of the internal ram of the processor as stack memory.*
- *The compiler contains an intermediate code optimiser.*

The parix 1.2 release provides only the BLAS level 1 routines (level 1 BLAS defines vector operations). The BLAS level 2 (matrix-vector) and level 3 (matrix-matrix) are not available (yet). We have tested a few of the BLAS routines and compared them with the same routines implemented in fortran without any optimisation. As we can see in table 1.1 the use of BLAS routines can speed up a program's performance dramatically.

| Subroutine: | BLAS Mflops | Non optimised Mflops | Relative speed of BLAS* |
|---|---|---|---|
| DDOT | 1.25 | 0.35 | 3.52 |
| SDOT | 1.77 | 0.43 | 4.11 |
| DAXPY | 1.03 | 0.35 | 2.93 |
| SAXPY | 1.46 | 0.38 | 3.65 |
| DSCAL | 0.71 | 0.22 | 3.32 |
| SSCAL | 1.29 | 0.27 | 4.57 |

Table 1.1 (* The numbers of the second column divided by the numbers of the third column.)

The intermediate code optimizer is an optional compiler pass. It is called with the switch -OI. If one doesn't like waiting one has to use the switch with care, because the compiling time increases a lot. We have tested the OI-switch on the non optimised routines of the previous test. The results are given in table 1.2.

| Subroutine: | With -OI Mflops | Non optimised Mflops | Relative speed* |
|---|---|---|---|
| DDOT | 0.40 | 0.35 | 1.13 |
| SDOT | 0.45 | 0.43 | 1.04 |
| DAXPY | 0.40 | 0.35 | 1.13 |
| SAXPY | 0.42 | 0.38 | 1.10 |
| DSCAL | 0.22 | 0.22 | 1.00 |
| SSCAL | 0.27 | 0.27 | 1.02 |

Table 1.2 (* The numbers of the second column divided by the numbers of the third column.)

Stack optimisation will cost more work. Each routine has to be optimised seperately. There are two ways of optimising the stack usage (actually there is a third one: removing stack checks but this is not encouraged):

Preventing unnecessary stack extensions.

*Stack extension is a very expensive operation. But it can be prevented by enlarging the stack size. We have tried this but if the code was compiled with the option '-W1,-Xstack_extension=0' the resulting executable produced screens full of error messages or just hang.*

Using fast memory for the stack.

The fast internal ram can be used for the stack. although the OI-switch does invoke some stack optimisation, it will not make use of the internal ram; this should be compelled manually. For *expensive subroutines (or functions) this may lead to considerable performance improvement.*

We will go a little bit more into detail about the internal ram. Although the internal memory is very small it can still be used for subroutines (or functions) that are small and expansive. The internal ram is only 4K and most of it is already in use by the shell. According to our

experiments there's left about 630 bytes. With such a small amount of ram it is most likely that stack extension can't be prevented, but for each subroutine call the stack extension is only called once. When the subroutine is not too trivial this won't take a significant amount of time.

A program for the matrix-vector product is used as an example. This program uses the routine DOT (a straightforward implementation of a double precision innerproduct). DOT is compiled with the following switches:

*f77.px -W1,-Xforce_stack_class=1,-Xforce_stack_size=630 [other options] -c dot.f.*
*-Xforce_stack_class=1* tells the compiler to use the internal ram.
*-Xforce_stack_size=630* tells the compiler to use exactly 630 bytes for the stack.

For comparison we have also compiled it without optimisation. Figure 1.3 shows that the optimised version is up to 36% faster.

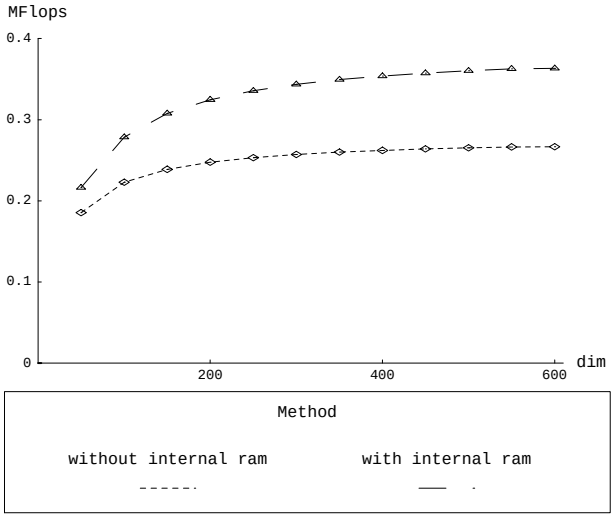



Figure 1.3

## 2 Trees and other topologies

Sending data from one processor to a $p$ by $p$ square of processors, using a grid or torus topology, will cost $\sqrt{p}$ communication steps. Assuming that the virtual links are not dramatically slower than physical links, this can be done much more efficiently. In theory the number of processors that contain the data can be doubled in each communication step. This will cost $^2\log p$ steps. The question is, whether a topology to do this can be made, and whether it will be efficient.

It is important to keep the following in mind when making virtual topologies, for it could have a large influence on the efficiency:

> *Consider the bottom row of a mesh of processors. Assume there is a virtual link between processor 0 and 2 and one between processor 1 and 3. When using the two links both at the same time, these links will try to use the connection between processor 1 and 2. In the worst case this causes that the communication will be performed serially.*

The set of processors to which the data is sent does not always have to be the complete mesh. In many cases the set will be a row or column of the mesh. It can be shown that with combinations of trees for rows and columns, a tree for a grid can be made.

Our aim is to implement the conjugate gradient (CG) method on the parsytec and we want to apply the above mentioned idea to the matrix-vector product. A standard way to implement a matrix-vector product is based on the following: project the transposed matrix on the square grid of processors and distribute the $x$ vector over the diagonal of the grid. By storing the transposed matrix one can use DDOT, with increment 1, for the matrix-vector product which is a little bit faster than DAXPY, which should be used in case of storing the matrix as such. A global description of the actual algorithm is as follows:

> *Step 1) Distribute the x vector of each diagonal processor over its row.*
> *Step 2) Perform the matrix-vector product in each processor.*
> *Step 3) Gather and add the b vectors from each column and put the result in the diagonal processor.*

When using a grid topology the communication step 3 (resp. 1) consists of gathering (resp. distributing) over a pipe of $\sqrt{p}$ processors. This will cost $\sqrt{p}-1$ communication steps. When a torus is used, the processors of a row (resp. a column) are connected by a ring, which will result in $\sqrt{p}/2$ communication steps.
It might be possible to use only $^2\log \sqrt{p}$ steps, as we mentioned before. Table 2.1 illustrates the effect of this.

*Number of communication steps for steps 1 and 3:*

| Topology: | 16 procs | 64 procs | 256 procs |
|---|---|---|---|
| grid | 3 | 7 | 15 |
| torus | 2 | 4 | 8 |
| optimal | 2 | 3 | 4 |

Table 2.1

## 2.1 Creating the optimal topology

We want to make an optimal topology which gathers all data from a row of processors. Figure 2.1 shows which communications should be performed. The topology needed for this communication scheme will be given later.

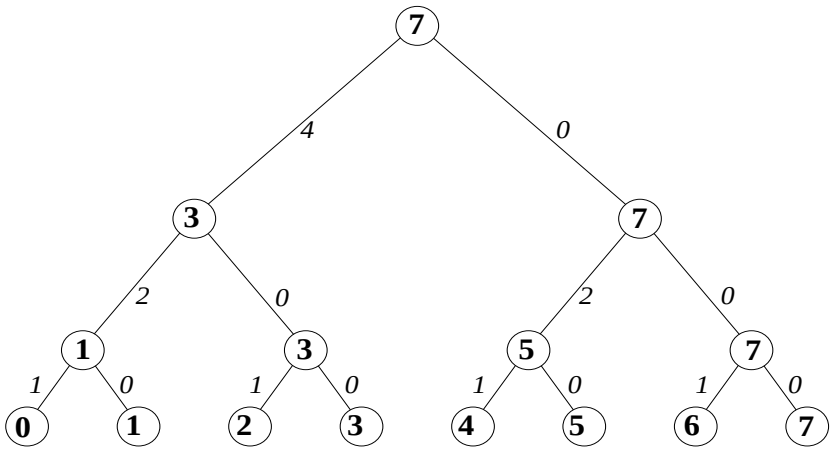



Figure 2.1

Explanation:
A subtree like the one depicted in figure 2.2 means that the data from processors 0 and 1 is gathered in processor 1. This will take one communication step. The edges define where the data comes from.

With a little of bit imagination, a topology can be seen in the graph from figure 2.1. The edges are the links of the topology and the weights at the links denote the physical length of the links. Off course links with weight zero do not really exist. Figure 2.3 shows the topology that can be obtained after removing the links with weight zero from figure 2.1.

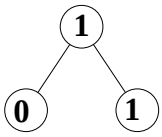



Figure 2.2

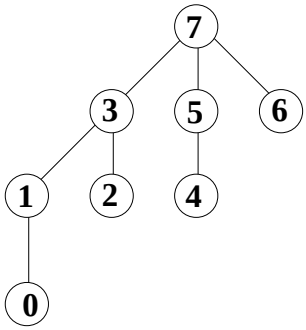



Figure 2.3

### 2.1.1 Generalisation

In the previous examples the data always had to be collected or gathered in the processor at most right end of the row (resp. column). For the matrix-vector product the data in a column of the grid must be collected in the diagonal processor. This means that the algorithm for creating the topology should be able to put an arbitrary processor at the top of the tree. An efficient way to do this is the following:

> *For a gather operation the data is always sent towards the top processor. Consider a row of 8 processors with processor 3 as the top. In the first step of the communication, the data is gathered from processor 4 and 5 in 4 because processor 4 is closer to processor 3 than processor 5.*

When looking at the tree going with this topology as depicted in figure 2.4, it can be noticed that at each level the processors are sorted from left to right. This ensures that links at the same level in the tree will not cross and that the problem mentioned at the beginning of this chapter does not occur.

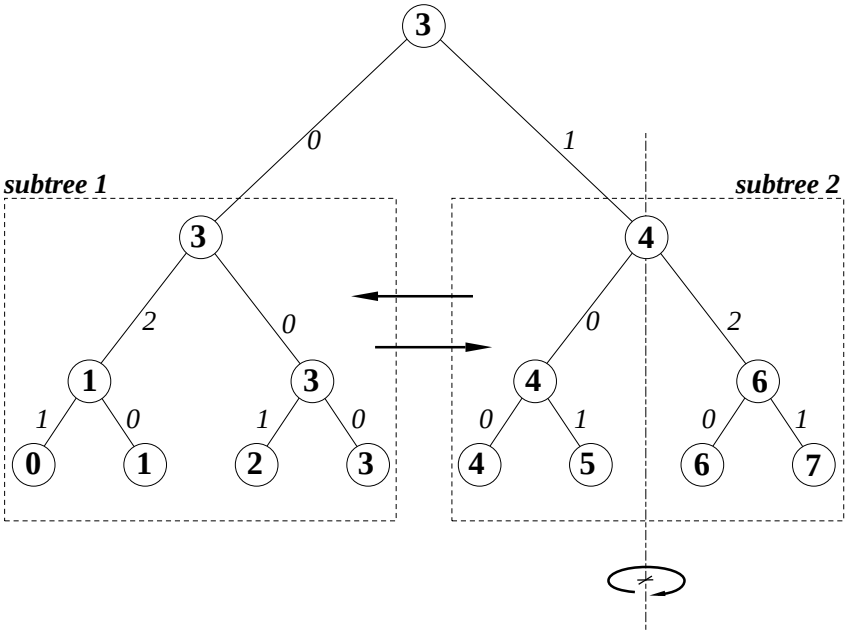



Figure 2.4

### 2.1.2 The algorithms

Now we have determined the right topology and we can design an algorithm for its construction. In order to do this we will change the tree in figure 2.4 a little to make it similar to the tree in figure 2.1 (i.e. top processor should be at the most right position and all edges that are pointing to the left should have weight zero). The order of the processors at the bottom row have to be changed in the following manner: mirror the most right subtree of depth 2 and exchange the two subtrees of depth 2. Algorithm 2.1 shows how this can be done in general.

```
                              MAKEID

Given:
    dim     : dimension of the grid
    l       : 2log(dim)
    treeid  : an array with the initial sequence of the processors
    top     : points to the place in the array with the processor
that             should end at the top of the tree

Algorithm:
    power=1
    do j=1,l
       power=power*2
       limit=(dim*(power-1))/power
       if (top .le. limit) then
           do i=1,dim-limit
              dum(i)=treeid(dim-i+1)
              treeid(dim-i+1)=treeid(limit-i+1)
           end do
           do i=1,dim-limit
              treeid(2*limit-dim+i)=dum(i)
           end do
           top=top+dim-limit
       end if
    end do
```

Algorithm 2.1

Each step of the outer do-loop looks one level deeper into the tree. Step *i* looks at the two rightmost subtrees of depth *l-i* (*l=*$^2$*log* $\sqrt{p}$). If the top is located in the left subtree the right subtree is mirrored and the two subtrees are exchanged. The initial sequence of processors will usually be the sorted list of processor id's.

After determining the correct order of the processors, the topology can be created with algorithm 2.2. For the matrix-vector product we have to create such a tree for each row and each column. It is useful to notice that if *link(i+1)=-1* then *link(i)* leads upwards (e.g. if data has to be gathered and *link(i+1)=-1* then the processor has to send its data, else the processor will receive data from another processor).

### 2.1.3 The communication on the tree

For the use of the tree topology we present two communication routines. They are called *gathertree* and *sendtree*. The two routines look much alike so we will only discuss *sendtree*. As mentioned before *treeid(i)=-1* can be of very good use for the communication. The complete routine consists of *l* steps. Each step goes one level down in the tree. Step *i* looks whether *treeid(i+1)=-1*, if this is true and *treeid(i)>-1* then the processor is going to receive, else it is going to send. The complete routine is given in algorithm 2.3.

```
                              MAKETREE

Given:
    dim     : dimension of the grid
    l       : 2log(dim)
    treeid  : an array with the bottom row (of the tree) of
processors
    myid    : the processor identity

Algorithm:
    reqid=1
    power=1
    tree=NewTop(l)
    do j=1,l
      link(j)=-1
    end do
    do j=1,l
       power=power*2
       do n=1,dim/power
           if (treeid(n*power) .eq. myid) then
               link(j)=AddNewLink(tree,treeid(n*power-
               power/2),reqid)
           end if
           if (treeid(n*power-power/2) .eq. myid) then
               link(j)=AddNewLink(tree,treeid(n*power),reqid)
           end if
       end do
    end do
```

Algorithm 2.2

```
                              SENDTREE

Given:
    tree,link,treeid,l,dim: see maketree
    x       : the variable that is to be send
    n       : the dimension of x

Algorithm:
    length=8*dim
    If (link(l) .ge. 0) Then
       If (treeid(procdim) .eq. myid) Then
           Call Send(tree,link(l),x,length)
       Else
           Call Recv(tree,link(l),x,length)
       End If
    End If
    Do i=l-1,1,-1
       If (link(i) .ge. 0) Then
           If (link(i+1) .ge. 0) Then
               Call Send(tree,link(i),x,length)
           Else
               Call Recv(tree,link(i),x,length)
           End If
       End If
    End do
```

Algorithm 2.3

## 2.2 Testing the performance of the tree

We have tested the performance of a 16 processor tree on the bottom row of a 16 by 16 mesh of processors and compared the results with the bottom rows of a grid, a torus and an xgrid. There are two torusses: the one from the virtual topology library is denoted by *torus* and the one denoted by *xgrid* is made from a grid topology by connecting the end points. We made the xgrid because the torus from the library is not faster than the grid. This is probably caused by links that are crossing and the slow communication between links with length greater than 1. The xgrid is not optimal, but in each row or column there is only one link that is crossing the other ones.

In theory the tree is 15/4=3.75 times faster than the grid, in practice this is only 2.4. The torus uses $\sqrt{p}/2$ communication steps so in theory the tree is twice as fast as a torus, in practice this is only 1.4. These differences are, of course, caused by the fact that the links with length one are much faster than links with length greater than one. For larger grids we can of course expect more advantage from the tree topology. Under the assumption that the models proposed in the next chapter are not too bad, the tree is 3.9 times faster than a grid topology on a 32 by 32 mesh of processors. The graphs for the communication times are plotted in figure 2.5:

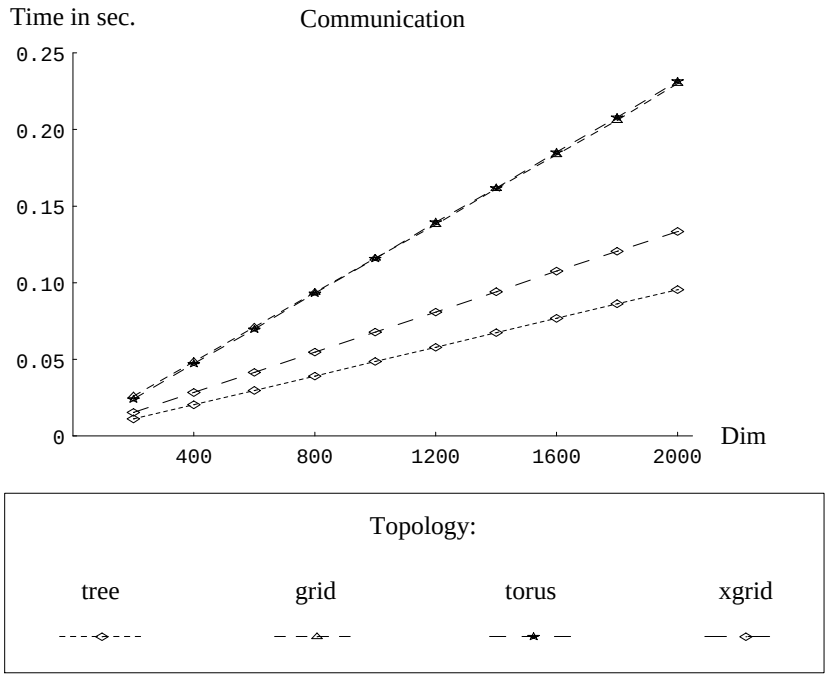



Figure 2.5

## 2.3 Timing models

In order to be able to explain the results of the tests that will follow, a few models for the communication times for different topologies will be presented in this chapter. Models are included for communication over virtual links and for broadcasting over a line of processors. Because we don't exactly know how the torus from the virtual topology library is made, we are not able to make a reliable model for this topology.

### 2.3.1 Model for the grid

A timing model for the grid is following Zhang and Hoffmann [1] given by:

$$T^{grid}(n,h,d) = 103.7 + 77.0 \cdot (h-1) + 0.90 \cdot n + 0.15 \cdot (d-1) \cdot n + 0.91 \cdot (h-1) \cdot n \quad \mu sec,$$

n: the number of bytes,

h: the number of hops,

d: the number of atoms.

This model implies a model for the broadcasting speed of the bottom row of a grid. It is given by:

$$T^{grid}_{broadcast}(n,p) = T^{grid}(n,15,4) = 1181 + 14.09 \cdot n \quad \mu sec.$$

### 2.3.2 Model for the virtual links

We have performed the tests for the communication on plain virtual links of different lengths. As remarked by Zhang and Hoffmann the communication speed is about 15% slower for processors in different atoms. The speed of links with length greater than 1 is about 67% slower than of links with length 1. The models are given by:

$$T^{Vlink}_{1}(n,d) = 102 + 0.90 \cdot n + 0.17 \cdot (d-1) \quad \mu sec,$$

$$T^{Vlink}_{2,3}(n,d) = 102 + 1.51 \cdot n + 0.18 \cdot (d-1) \quad \mu sec,$$

$$T^{Vlink}_{>3}(n,h) = 102 + 1.68 \cdot n + 0.0042 \cdot h \cdot n \quad \mu sec.$$

The errors for small n are quite large, so the start-up times are not very reliable.

### 2.3.3 Model for the tree

With these formulas we have made a model for the broadcasting speed on the tree. The model for the broadcasting speed of a tree with 16 processors and top at processor 15 is given by:

$$T^{tree}_{16,15}(n) = T^{Vlink}_{1}(n,1) + T^{Vlink}_{2,3}(n,1) + T^{Vlink}_{>3}(n,4) + T^{Vlink}_{>3}(n,8) = 408 + 5.82 \cdot n \quad \mu sec.$$

The maximum error for this model is about 5%.
When looking at a (sub)tree of depth *l* it can be noticed that all the processors form a concatenated sequence. This holds of course also for subtrees of depth *l-1*. The link in the highest level of the tree is between the top and the processor in the left subtree that is closest to the top. The maximum distance between these two processors is always $2^{l-1}$. It is clear that when the top is at the processor with the highest id, this maximum is attained in each subtree. This means that this is the slowest tree that can be made with the proposed algorithm (but it is of course the fastest tree with that particular top). The timing models for trees with top at other processors are not important for the matrix vector product because the

slowest tree determines the total communication time. The fastest tree is obtained by taking the top in the middle. We will use this tree for the innerproduct. The model is given by:

$$T^{tree}_{16,7} = T^{Vlink}_{1}(n,1) + T^{Vlink}_{2,3}(n,1) + T^{Vlink}_{>3}(n,4) + T^{Vlink}_{1}(n,2) = 408 + 5.15 \cdot n \quad \mu sec.$$

### 2.3.4 Model for the xgrid

In section 2.2 we introduced the xgrid. A model for the broadcasting speed at the bottom row of the 16 by 16 xgrid can be obtained from the above formulas as follows:

$$T^{xgrid}_{broadcast}(n) = 5 \cdot T^{Vlink}_{1}(n,1) + T^{Vlink}_{1}(n,2) + T^{Vlink}_{>3}(n,15) = 714 + 7.28 \cdot n \quad \mu sec.$$

The maximum error of this model is about 10%.

# 3 Conjugate gradients for dense and sparse matrices

## 3.1 The matrix-vector product

We have implemented the matrix-vector product for three topologies: a tree topology ($MV_{tree}$), a torus ($MV_{torus}$) and an xgrid ($MV_{xgrid}$). The communication times for a grid and the torus are the same, so we can assume that the speed-ups are also the same. The internal ram is used for this test, which results in a performance for a single processor of 1.24 Mflops.

Figure 3.1 shows that even for a full matrix-vector product the performance can be increased significantly by using the tree topology. Because we are interested in the effect of the communication times, the dimension/√p is printed at the x-axis.

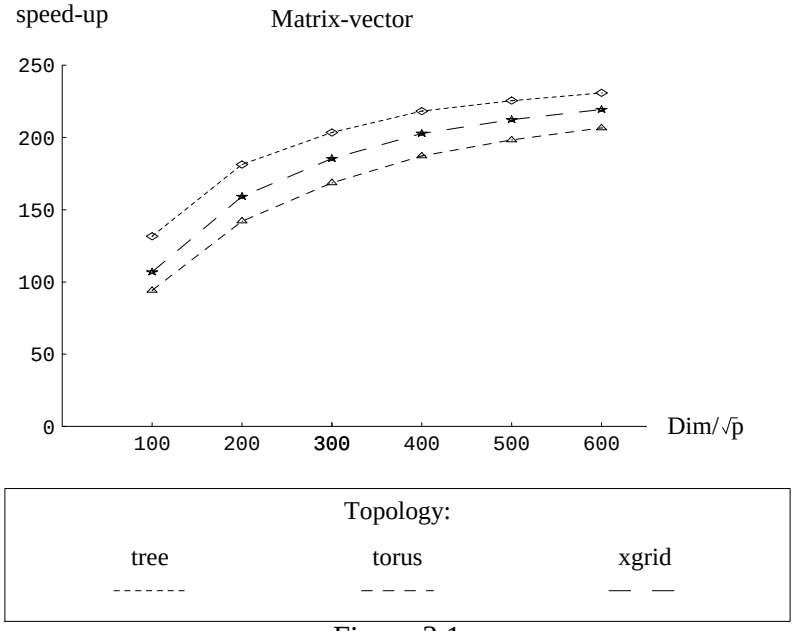


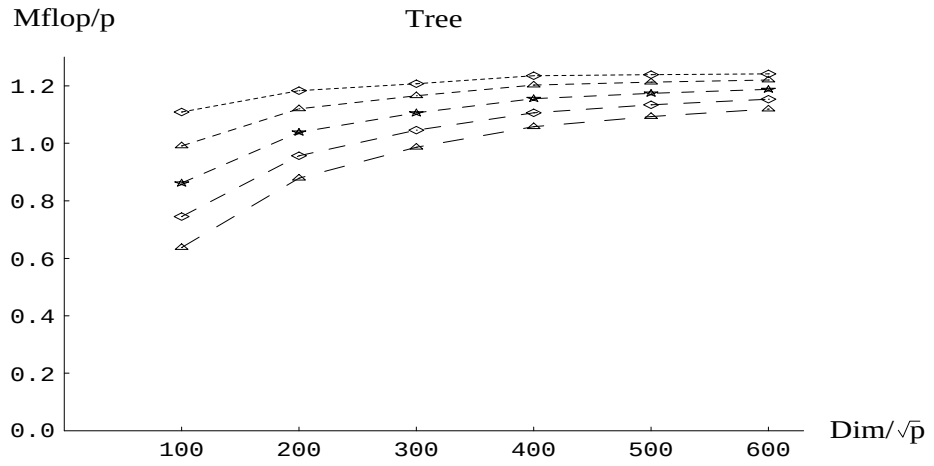

Figure 3.1

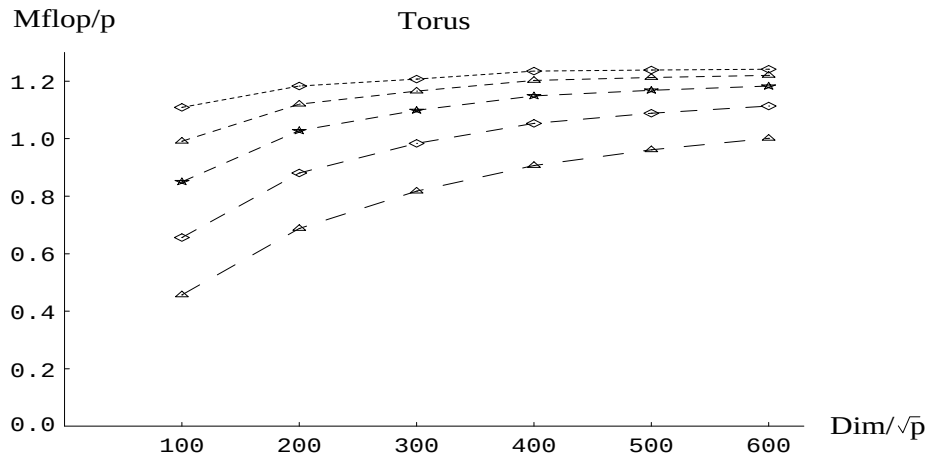
To compare the scalability of two routines, we need to compare the results for different numbers of processors. To do this we have plotted the Mflops rate per processor for $MV_{tree}$ and $MV_{torus}$ in figure 3.2. As we can see $MV_{tree}$ is much better scalable than $MV_{torus}$. Each time the dimension of the grid is doubled, the performance of $MV_{torus}$ is decreasing with fast increasing steps while $MV_{tree}$ is decreasing with a small (almost) constant value.

*Speed-ups for the matrix-vector product:*

| Dim/√p: | 100 | 200 | 300 | 400 | 500 | 600 |
|---|---|---|---|---|---|---|
| Tree | 131.6 | 181.3 | 203.5 | 218.2 | 225.6 | 230.9 |
| Torus | 94.2 | 142.1 | 168.7 | 187.2 | 206.6 | 198.3 |
| Xgrid | 107.0 | 159.3 | 185.4 | 202.9 | 212.4 | 219.5 |

Table 3.1




| Number of processors: | | | | |
|---|---|---|---|---|
| 1 | 4 | 16 | 64 | 256 |

Figure 3.2

## 3.2 Dense matrices

We have parallelised the conjugate gradient algorithm that is described in Van der Vorst [2]. The matrix-vector product of paragraph 3.1 is perfectly suited for parallelising this algorithm. This routine assumes that the *x* and *b* vectors are distributed over the diagonal processors. As a consequence all the additional vectors that are needed for the CG algorithm must also be distributed over the diagonal. This has the disadvantage that all vector operations will be calculated on only $\sqrt{p}$ processors, but when the matrix is full, or not too sparse, we don't have to worry about this, because the matrix-vector products will cost much more time than the vector operations.

For the matrix-vector product we can choose between a couple of topologies: a grid, a torus, an xgrid and a tree. Note that the matrix have to be symmetric positive definite which means that we can use DDOT without any problems.

For the innerproduct we need a topology over the diagonal, a simple pipeline is good enough (in case of a torus there are several diagonals, but we take the processors with $x_{id}=y_{id}$). Of course we can also use a tree but, the number of processors is too small to notice any difference.

The CG algorithm:

$x_0$ = initial guess; $r_0 = b - A \cdot x_0$;
$p_{-1} = 0$; $\beta_{-1} = 0$;
$\rho_0 = (r_0, r_0)$;
for i = 0,1,2,....
  $p_i = r_i + \beta_{i-1} \cdot p_{i-1}$;
  $q_i = A \cdot p_i$;
  $\alpha_i = \dfrac{\rho_i}{(p_i, q_i)}$;
  $x_{i+1} = x_i + \alpha_i \cdot p_i$;
  $r_{i+1} = r_i - \alpha_i \cdot q_i$;
  if $x_{i+1}$ accurate enough then quit;
  $\rho_{i+1} = (r_{i+1}, r_{i+1})$;
  $\beta_i = \dfrac{\rho_{i+1}}{\rho_i}$;
end;

Algorithm 3.1

The number of operations for the CG algorithm is given by:

$2 \cdot n \cdot (n+1) + iter \cdot (n \cdot (2 \cdot n + 9))$, with n the dimension and iter the number of iterations.

As can be seen from inspection of algorithm 3.1.

### 3.2.1 The tests

We have chosen to test CG with the tree and the torus on a partition of 16 by 16 processors. The matrix is full and $x_0=0$. As we are not interested in the convergence we can keep the number of iterations fixed at 20.
With a torus a processor doesn't know the identities of the other processors. This makes it very hard to implement the pipeline. A lot of communication is needed to let the torus cooperate with other topologies (the xgrid would be a much better choice in this case).
The results are plotted in figure 3.3. The tree is, as expected, faster than the torus. The communication time is a function of n and the computation time a function of $n^2$, this has the effect that for small n the difference between $CG_{tree}$ and $CG_{torus}$ is quite large. Table 3.2 gives the speed of $CG_{tree}$ relative to $CG_{torus}$.

| dimension: | 800 | 2400 | 4800 | 7200 | 9600 |
|---|---|---|---|---|---|
| relative speed: | 1.38 | 1.30 | 1.20 | 1.15 | 1.12 |

Table 3.2

We estimated the performance of a single processor with the use of the internal ram at 1.24 Mflops. With a dimension of 9600 the speed-up of $CG_{tree}$ is 227 and the speed-up of $CG_{torus}$ is 203.

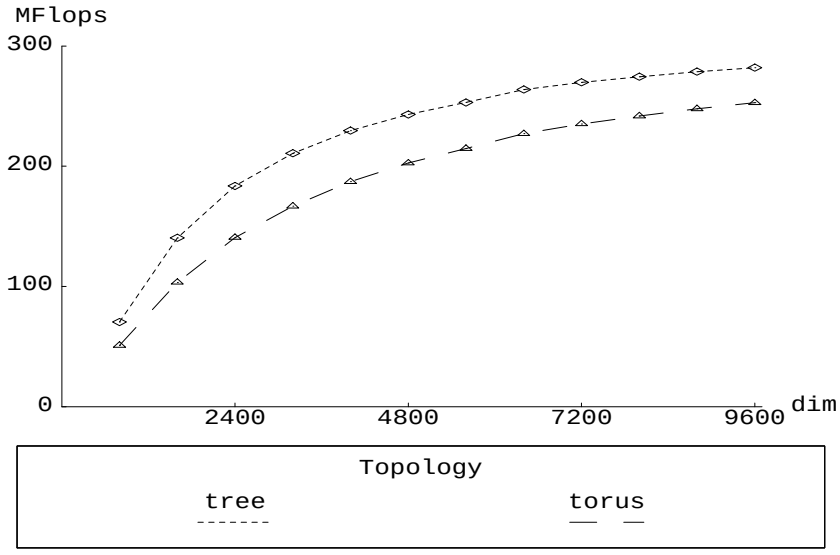



Figure 3.3

## 3.3 Sparse matrices

When using sparse matrices the communication time will increase relative to the computation time. This has the effect that the difference between $CG_{tree}$ and $CG_{torus}$ will increase. Another effect is that the vector operations in the CG iteration (DDOT and DAXPY) will cost more time relative to the matrix-vector product.

### 3.3.1 Sparse BLAS

In 1985, Dodson and Lewis [3] proposed their sparse BLAS extensions and developed this proposal further in [5] (a full implementation of this package can be found on netlib). These ideas have been the basis for Heroux [4] to write his proposal for level 3 sparse BLAS. In this proposal he introduces a few matrix storage methods. We are going to use the compressed sparse column (CSC) method.
The CSC method stores each column as a sparse vector. A sparse vector is stored as follows: *A double precision array is used which is just long enough to contain the non trivial components. and an integer array with indices is used to map the stored values on their proper positions within the vector.*
For a matrix all sparse columns are stored in two arrays (one with the values and one with the indices) and an additional array of dimension n+1 is used to store the pointers to the begin of each column. An example is given below:

The original matrix is given by:

$$A = \begin{pmatrix} 1.0 & 0 & 0.3 & 6.2 \\ 0 & 8.5 & 0 & 0 \\ 0 & 3.1 & 0 & 7.8 \\ 2.3 & 0 & 0 & 0 \end{pmatrix}$$

The compressed form is given by:

$$VAL = (1.0 \quad 2.3 \quad 8.5 \quad 3.1 \quad 0.3 \quad 6.2 \quad 7.8)$$

$$INDX = (1 \quad 4 \quad 2 \quad 3 \quad 1 \quad 1 \quad 3)$$

$$PNTR = (1 \quad 3 \quad 5 \quad 6 \quad 8)$$

Note that the number of non zeros in row i equals $PNTR(i+1) - PNTR(i)$ and the total number of non zeros equals $PNTR(n+1) - 1$.
We can use DDOTI, the sparse version of DDOT, for our matrix vector product, because the matrix that we want to use will be symmetric. DDOTI has the advantage over DAXPY that it can easily be optimised by using loop enrol. Optimisation is certainly necessary, otherwise the speed of DDOTI is lower than 0.33 Mflops. The internal ram is used for the outer loop of the matrix-vector product, this gives the best results. The inner loop (the one from DDOTI) hasn't got to do very much due to the sparsity and the loop enrol, while the outer loop has to run from 1 to n (n can be quite large). It should however be noticed that in Dodson, Grimes and Lewis [5] the advice is given to use different code, separately optimised for small and large number of non zeros per column (i.e. for large number of NZ per column it might be desirable to use the internal ram for the inner loop).

### 3.3.2 Timing models

The timing model for the sparse matrix-vector product can be split up into two parts: the communication time model, and the computation speed model. The first model is already given in paragraph 2.3. The computation speed depends on dim and 1/dim, the formula is given by:

$$V_{compute}(dim) = 172 + \frac{182}{dim \cdot 10^{-3}} - 0.23 \cdot dim \cdot 10^{-3} \quad \text{Mflops.}$$

This model is obtained by doing a least squares fit on the measured data.
The number of non zeros is $512 \cdot 10^5$, and the number processors is 256. The graphs for this model and the measured values are plotted in figure 3.4. The peaks in the graph can be ascribed to the loop enrol that is used in DDOTI. The performance decreases from 187 Mflops to 137 Mflops, this can be explained by the following: when the dimension grows, the computational work does not increase (NZ is constant), but the speed will decrease because DDOTI is called much more often (with a very small value for NZ).

The model for the matrix vector product is given by:

$$V^{tree}_{matvec}(n, dim) = T^{tree}_{communicate}(n) + \frac{2 \cdot NZ}{V_{compute}(dim)} =$$

$$2 \cdot (408 + 5.82 \cdot n + \frac{512 \cdot 10^5}{172 + 182 \cdot 10^{-3} \cdot dim^{-1} - 0.23 \cdot dim \cdot 10^{-3}}).$$

The graphs for this model are plotted in figure 3.5.

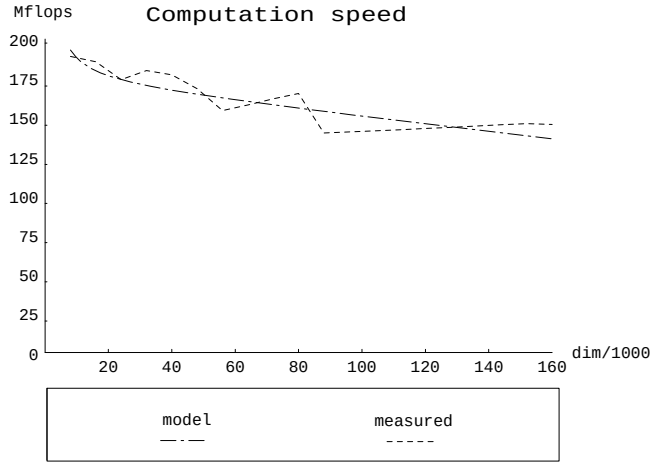



Figure 3.4

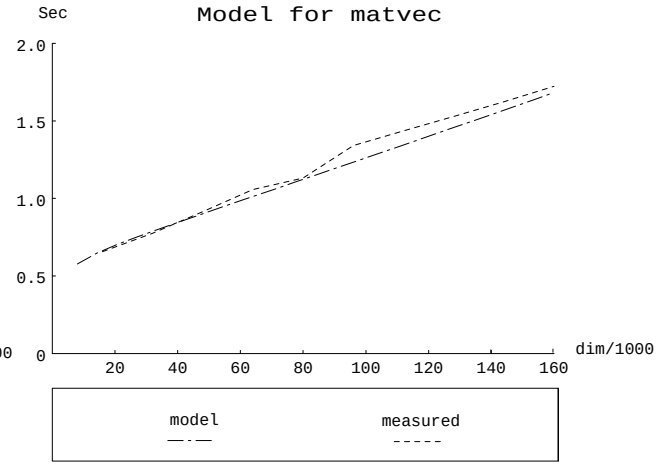



Figure 3.5

### 3.3.3 The tests

The matrices that we have used for the tests are uniformly sparse (a constant distance in the x and the y direction) and have dimensions from 10,000 up to 160,000, the number of non zeros is $512 \cdot 10^5$. All tests have run on 256 processors. The speed of the CG method depends completely on the speed of the matrix-vector product, therefore we have tested this first. Figure 3.6 shows that the tree is up to 71% faster than the torus (58.9 Mflop vs. 34.3 Mflop). The Parsytec does not have enough ram to do large enough tests in order to estimate the Mflops rate on one processor (we expect that it will be about 0.8 Mflops), therefore we cannot give an accurate value for the speed-up.

We have used the same type of matrix for the CG method. The matrix has $512 \cdot 10^5$ non zeros and is made as follows:

$$A_{i,j} = 20.0d0 \text{ when } i = j,$$

$$A_{i,j} = -5.0d0 \text{ when } i + 1 = j, \text{ or } i = j + 1.$$

The rest of the non zeros have the value 0.01 and are distributed with equal distance in x and y direction. The number of iterations is kept fixed at 20. The results are plotted in figure 3.7.

When the graphs from figure 3.6 and 3.7 are compared we can conclude that the matrix-vector product still consumes most of the processor time in the CG algorithm. The difference between the sparse versions of $CG_{tree}$ $CG_{torus}$ is as expected larger than when using dense matrices: $CG_{tree}$ is up to 63% faster than $CG_{torus}$ (65.6 Mflop vs. 40.2 Mflop).

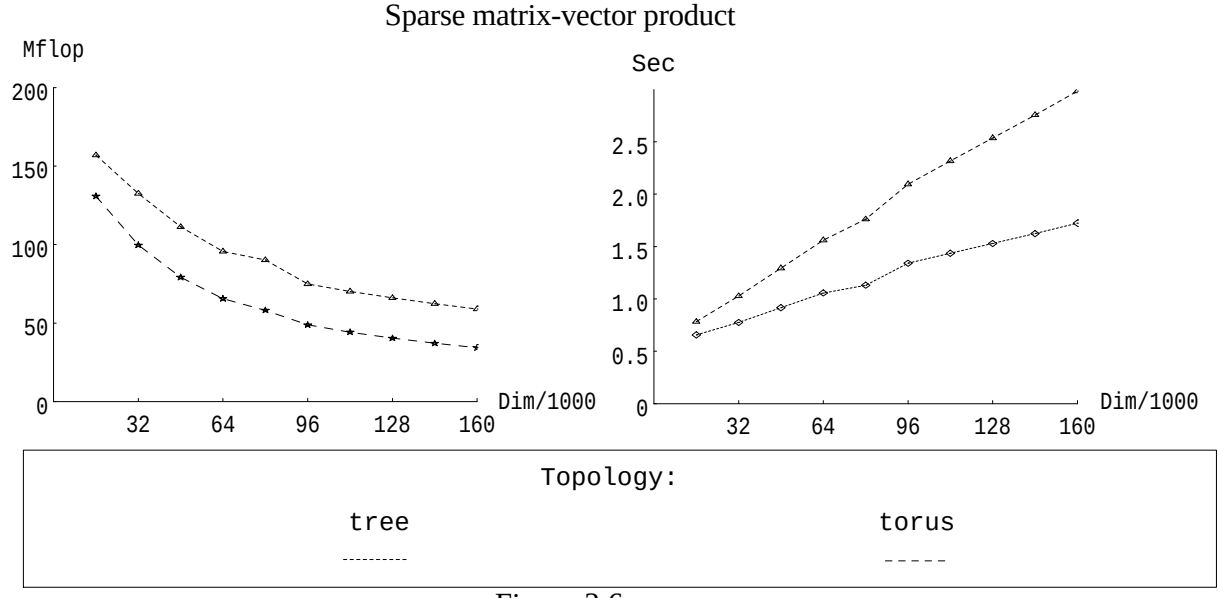



Figure 3.6

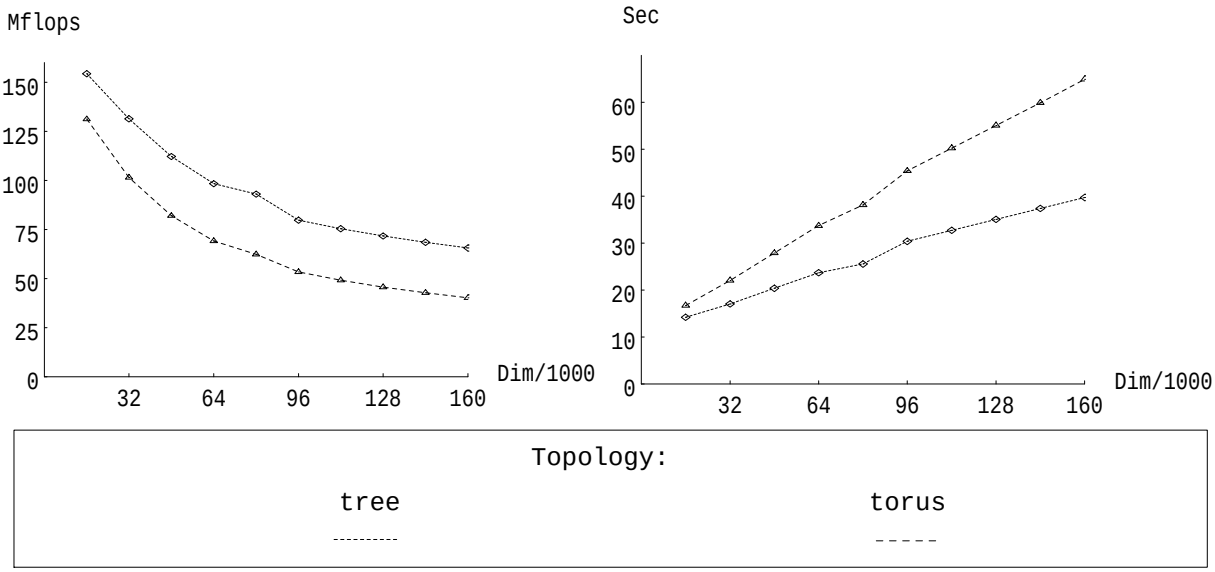

Sparse Conjugate Gradients
Mflops
150
125
100
75
50
25
0
32
64
96
128
160
Dim/1000
Sec
60
50
40
30
20
10
0
32
64
96
128
160
Dim/1000
Topology:
tree
torus

Figure 3.7

# 4 Conjugate gradients method for band matrices

In this chapter we will discuss the CG method for linear systems that result from partial differential equations. The standard problem that is used as an example is Poissons equation in 3 dimensions:

$$\begin{cases} \Omega = \{\vec{x} = (x,y,z) \mid 0 < x < 1,\ 0 < y < 1,\ 0 < z < 1\} \\ \Delta\vec{u} = f(\vec{x}) \text{ in } \Omega \\ u(\vec{x}) = 0 \text{ on } \partial\Omega \end{cases}$$

Discretisation is defined by putting

$$h = \frac{1}{l+1} \qquad x_i = ih \qquad y_j = jh \qquad z_k = kh \qquad (0 \le i,j,k \le l+1)$$

We now introduce

$$v_{i,j,k} \approx u(x_i, y_j, z_k) \qquad f_{i,j,k} = f(x_i, y_j, z_k)$$

A discretised version of the problem is given by:

$$(v_{i-1,j,k} - 2v_{i,j,k} + v_{i+1,j,k}) + (v_{i,j-1,k} - 2v_{i,j,k} + v_{i,j+1,k}) + (v_{i,j,k-1} - 2v_{i,j,k} + v_{i,j,k+1}) = h^2 f_{i,j,k}$$

On the boundary u(x,y,z) is given, so the range of i, j and k is {1,2,...,*l*}. For the boundary we require

$$v_{0,j,k} = v_{l+1,j,k} = v_{i,0,k} = v_{i,l+1,k} = v_{i,j,0} = v_{i,j,l+1} = 0$$

## 4.1 the matrix-vector product

We showed that the matrix of the discretised version of Poissons equation in three dimensions has seven diagonals and is symmetric. This can be generalised to *n* dimensions, thus leading to a sustem with *2n+1* diagonals. The distance of the outmost diagonal to the main diagonal, which is very important as we will show later, equals $l^{n-1}$. In general the lower diagonals (including the main diagonal) are stored in an array of dimension $l^n$ by *k*, with *k=n+1*. An additional array of dimension *k-1* is needed to store the distance of the diagonals to the main diagonal.
The matrix is distributed over the processors in a way which for three diagonals is illustrated in figure 4.1. This distribution has, besides some nice properties that we will discuss later, the advantage that the symmetry of the matrix is used optimally. It is assumed that
$l^{n-1} \le l^n/p$, *or* $p \le l$. With this condition it is clear that a pipeline topology is a good choice, because only communication between neighbour processors is needed.

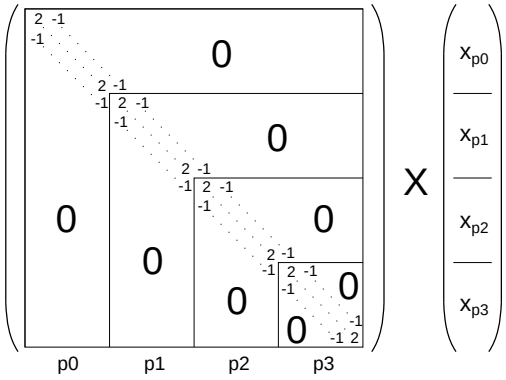


Figure 4.1

The vector *x* is distributed in such a way that the product of the lower diagonals with *x* can be calculated without communication. This is one of the other nice properties of the distribution, there is an easy and efficient way to use asynchronous communication. First

the product of the lower triangular part of *A* and *x* is computed. For the product of the remainder of *A* with *x*, processor *p* needs part of $x_{p-1}$. This communication step can be done during the computation of the first part of the product. Part of the result of the first computation is needed in the next processor, again the communication can be done asynchronously during the second computation.
The matrix-vector product is schematically given by:

**Notation:**
m is the distance of the outmost diagonal to the main diagonal
$A_0^p$ is the main diagonal of A in processor p, n is length of this vector
$A_1^p$ is the part of A below the main diagonal in processor p
$A_2^p$ is the part of A above the main diagonal in processor p
The symbol ˜ is used for part results

**Computation:**
$(x_1^p, x_2^p, \ldots, x_m^p)^T$ is asynchronously sent to p - 1 and is received in $(x_{n+1}^{p-1}, x_{n+2}^{p-1}, \ldots, x_{n+m}^{p-1})^T$
$(A_0^p + A_1^p)(x_1^p, x_2^p, \ldots, x_n^p)^T = (\tilde{b}_1^p, \tilde{b}_2^p, \ldots, \tilde{b}_{n+m}^p)^T$
The asynchronous communications are synchronized
$(\tilde{b}_{n+1}^p, \tilde{b}_{n+2}^p, \ldots, \tilde{b}_{n+m}^p)^T$ is asynchronously sent to p + 1 and is received in dum
$A_2^p\,(x_1^p, x_2^p, \ldots, x_{n+m}^p)^T + (\tilde{b}_1^p, \tilde{b}_2^p, \ldots, \tilde{b}_n^p)^T = (\tilde{b}_1^p, \tilde{b}_2^p, \ldots, \tilde{b}_m^p, b_{m+1}^p, b_{m+2}^p, \ldots, b_n^p)^T$
The asynchronous communications are synchronized
$(b_1^p, b_2^p, \ldots, b_m^p)^T = (\tilde{b}_1^p, \tilde{b}_2^p, \ldots, \tilde{b}_m^p)^T + (\mathrm{dum}_1, \mathrm{dum}_2, \ldots, \mathrm{dum}_m)^T$

Algorithm 4.1

This approach is applicable in a much more general situation, namely to band matrices of various type. The only restriction is that the distance of the outmost diagonal to the main diagonal is greater or equal to the dimension divided by the number of processors.

The efficiency of this routine will be very good because the communication is local, and the communication is partially hidden behind the computations. For a really fast implementation it will be necessary to use an optimised version of the element wise product, that is used to calculate the product of a diagonal with a vector.

The routine is tested on 16 processors. The matrix *A* is the one that arises from the Poisson equation in 3 dimensions. The space is distributed in $l^3$ boxes of equal size. The resulting seven diagonal matrix has dimensions $l^3$ by $l^3$. For comparison the times are plotted for the routine without communication and with synchronous communication.

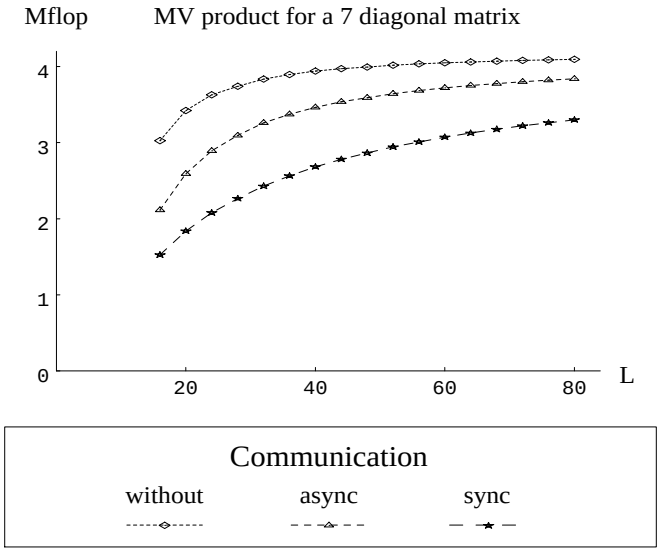



Figure 4.2

## 4.2 The innerproduct

For a parallel innerproduct a few communication routines are needed. Although only scalars need to be sent, it might be interesting to test if the optimal topology, which uses only $^2logn+^2logm$ communication steps (on a mesh of n by m processors), will be more efficient than a grid, which uses $(n+m)/2$ communication steps. The idea behind this is that when sending such small amounts of data the start-up time for the communication becomes more important. When the computation time is kept small by taking the dimension of the vectors not too large, the communication will take a significant amount of time and we might get the expected difference in the efficiency.

The implementation of the innerproduct is straightforward. The vectors $x$ and $y$ are chopped into $p$ ($p$ the number of processors) parts of equal length $x_1...x_p$, $y_1...y_p$. Processor $i$ gets the parts $x_i$ and $y_i$. First each processor calculates the local innerproduct (processor $i$ calculates $x_i.y_i$) then the local innerproducts are gathered and added up in one processor and finally the innerproduct itself is sent back to all processors.

In case of a grid topology, gathering is done by sending all local innerproducts to one column and then to one processor of that column. It will be most efficient when the column and the processor are in the middle of the grid (the maximum distance between a processor in the middle of the grid and an arbitrary processor is only $\sqrt{p}$). Broadcasting the final result is the reverse of the gathering process.

The same approach also works for the tree topology. Again we have to send the local innerproducts to the middle column. A tree topology with top processor in the middle column is made for each row. The next step sends the local innerproducts to one processor of the middle column. Another tree topology is made for the middle column with top in the middle of that column. Note that the combination of the two tree topologies forms a new tree topology. The communication routines gathertree and sendtree are used for the communication. A few adjustments were necessary to make them work optimally for scalars (i.e. DAXPY isn't needed anymore).

### 4.2.1 The tests

Since the difference between $\sqrt{p}$ and $^2logp$ is not significant for small $p$ we have tested the two innerproducts for 256 and 512 processors. The dimension per processor varies from 50

to 12800. For 256 processors the tree is from 5% to 48% faster than a grid depending on the dimension of the vectors. When the number of processors is 512 the tree is from 10% to 78% faster than a grid. Figure 4.3 shows that for very large dimensions the tree will not be significantly faster than the grid.

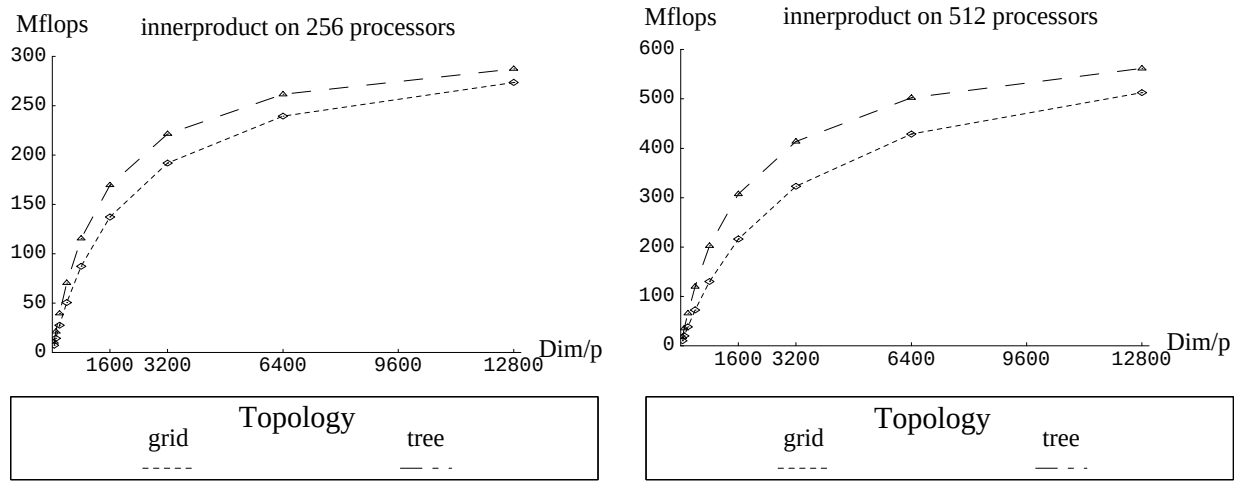



Figure 4.3

## 4.3 Tests for CG

We have used methods of §4.1 and §4.2 in the conjugate gradients algorithm to solve Poissons equation in three dimensions, as described in the introduction of this chapter. The system $Ax=y$, with $A$ a symmetric seven diagonal matrix of dimension $l^3$ , has to be solved. The distance of the outmost diagonal to the main diagonal is $l^2$, this leads to the condition that the number of processors is greater than $l$. To make a good comparison the number of iterations is kept fixed at 20. The results are plotted in figure 4.4.

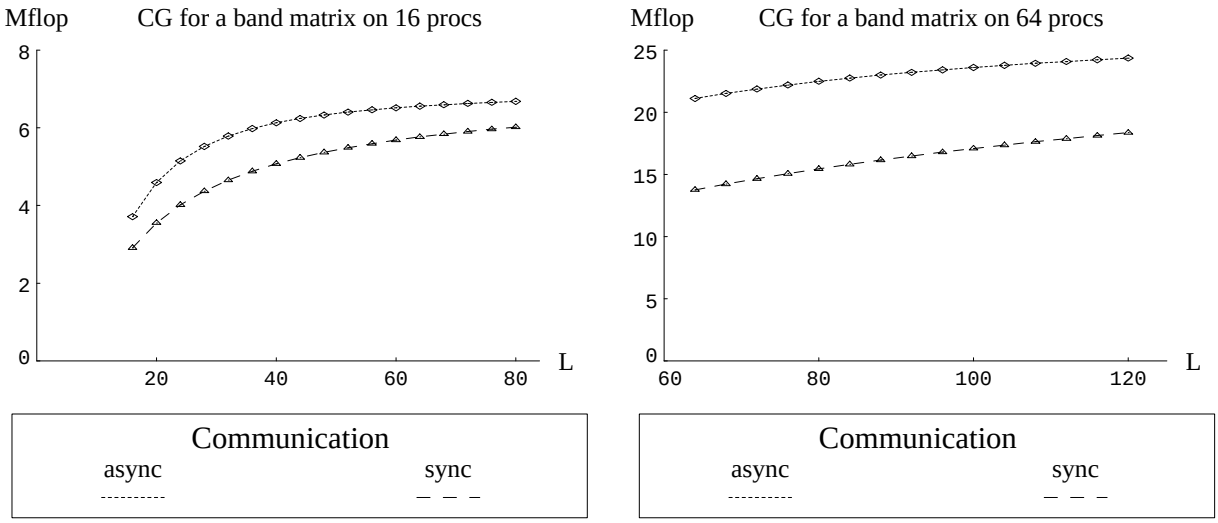



Figure 4.4

## Conclusion

From the test results we can conclude that, thanks to the new tree topology, a high degree of parallelism is attained for the dense version of the conjugate gradients method. The tree also results in a more scalable code. These properties do not hold for the sparse version. In this case the communication costs to much time in comparison with the computation. But of course more advantage is derived from the tree.

The tree is an optimal topology for sending and gathering data from a set of processors. A hypercube of high enough order can, theoretically, be optimal, but it is not easy to map such a many dimensional object efficiently onto a line or grid of processors. The tree does not have any useless links and this, together with the two dimensional representation, makes the tree very easy to use.

The implementation of CG for pde is much easier. The matrix vector product only needs local communication, this enables us to use asynchronous communication. A very efficient code can be obtained in relatively short time. A disadvantage is that BLAS does not provide an elementwise product.

The future machines (Parsytec GC and Power Stone) will have much faster processors and faster communication. The faster processors have the effect that the communication becomes more important, the faster communication has the opposite effect. Another change in the communication is that the virtual links and physical links are almost equally fast. This might imply that the hypercube can efficiently be used on these machines, which will make the tree topology superfluous.

## References


[1] Z. Zhang and W Hoffmann, (1993), *"Some basic performance tests on the parsytec GCel"*, Technical Report CS-93-16, University of Amsterdam.

[2] H.A. van der Vorst, (1992), *"Lecture notes on iterative methods"*, CERFACS, TR/TA/92/75.

[3] D.S. Dodson and J.G. Lewis, (1985), *"Proposed sparse extensions to the basic linear algebra subprograms"*, ACM SIGNUM Newsletter 20, 1, 20, pp. 22-25.

[4] M. A. Heroux, (1992), "A proposal for a sparse BLAS toolkit", SPARKER working note #2, Cray Research, /TR/PA/92/90, USA.

[5] D.S. Dodson R.G. Grimes and J.G. Lewis, (1988), "Model implementation and test package for the sparse basic linear algebra subprograms".

## Appendix: source codes

The source codes of two programs are presented in this appendix. The program CGPDE is the program described in chapter 4. It uses asynchronous communication for the matrix-vector product and a tree topology for the innerproduct. The second program, CGSPARSE, is discussed in paragraph 3.3. This program uses a tree topology for the matrix-vector product. The routine SYNCGRID is used to synchronise a grid. It is written by drs. K. Potma.