Time complexity analysis for distributed memory computers: implementation of parallel conjugate gradient method

Hoekstra, A.G.; Sloot, P.M.A.; Haan, M.J.; Hertzberger, L.O.

Publication date
1991

Published in
Computing Science in the Netherlands : jaarbeurs Utrecht 7 en 8 november 1991 : proceedings

Time complexity analysis for distributed memory computers

Implementation of a parallel Conjugate Gradient method[I]

A.G. Hoekstra, P.M.A. Sloot, M.J. de Haan and L.O. Hertzberger

Parallel Scientific Computing Group
Department of Computer Systems
Faculty of Mathematics and Computer Science
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, tel. 020-5257543

Published in

Computing Science in the Netherlands (J. van Leeuwen, ed.), (Amsterdam), pp. 249-266, 1991.

I This work is supported by a matched funding from SION, FOM and Stichting Biofysica (project NWO 810410004).

Time complexity analysis for distributed memory computers
Implementation of a parallel Conjugate Gradient method[I]

A.G. Hoekstra, P.M.A. Sloot, M.J. de Haan and L.O. Hertzberger

Parallel Scientific Computing Group
Department of Computer Systems
Faculty of Mathematics and Computer Science
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, tel. 020-5257543

Abstract

New developments in Computer Science, both hardware and software, offer researchers, such as physicists, unprecedented possibilities to solve their computational intensive problems. However, full exploitation of e.g. new massively parallel computers, parallel languages or runtime environments requires an approach that combines elements of computer science, numerical mathematics and, in our case, physics. We call this β-computer science.

Here we present an example of a computational intensive physical application, the coupled dipole formulation of elastic light scattering from arbitrary shaped particles. The computational kernel of this method is a large set of linear equations. We solve this set by means of a Conjugate Gradient (CG) method, implemented on a coarse grain distributed memory computer (a Transputer network).

This paper describes the parallelization of a CG method. Two important choices are discussed; what is the best possible decomposition and which processor network topology is most suited. We introduce a general method to answer these questions and investigate its usefulness by applying this method to our application. It is concluded that implementation of the CG method, with a row-block decomposition of the coefficient matrix, on a ring of Transputers is the most efficient choice.

Finally the actual implementation is described, and preliminary experimental timing results are compared with the theoretical time complexity analysis.

I Introduction and background

Traditionally users of the most powerful computing systems are found among researchers in the natural sciences and in specific engineering applications. Still, access to high-end vector computers has been restricted to selected and very specialized groups of researchers. The development of efficient applications, fully utilizing the unique properties of the vector architecture, was a very laborious process.[1] The introduction of good vectorizing compilers relieved this problem, but nevertheless, writing vectorizable code is still considered a specialized skill.

I This work is supported by a matched funding from SION, FOM and Stichting Biofysica (project NWO 810410004).

Computer Science is a fast moving field, and nowadays the revolutionary paradigm of distributed computing has generally been accepted as the candidate to meet the computational power required for recent and future applications.[2] Furthermore, parallel computing brings supercomputer power to a much broader group of users, due to the relative small investments compared to vectorsupercomputers ("distributed computers: the workman's supercomputer"). However, developing efficient code, totally exploiting the possibilities of the parallel hard- and software, is generally believed to be hard. Automatic parallelization tools for distributed memory computers are still in their infancy and formal approaches to parallelize an application hardly exist .

We believe that parallel scientific computing requires a new approach, where knowledge from computer science, numerical mathematics and from the applications field, such as physics, is combined. We propose to refer to this interdisciplinary research, which fills the gap between the formal description of an application and an abstract model of a (parallel) machine in a systematic and structured way, as β-computer science. Here we present some elements of β-computer science in conjunction with an example of parallelizing a specific application.

In previous years we developed both theory and experimental equipment to study the Elastic Light Scattering (ELS) from biological particles.[3,4,5,6,7,8] This research has moved to the computational field. We are developing methods to simulate the ELS from arbitrary shaped micron sized particles, by means of the Coupled Dipole method.[9] The main task is to solve a very large matrix equation[II] :

$$\mathbf{Ax} = \mathbf{b} \,, \qquad [1]$$

where **A** is a symmetric 3Nx3N complex matrix (with $N \sim 10^5$), **b** a known vector and **x** the unknown vector. Large linear systems, such as Eq. [1], are solved by means of iterative techniques.[10] We apply a very powerful iterative technique, the Conjugate Gradient (CG) method.[11]

In this paper we concentrate on a formal description of the parallelization of the CG method. First, in section II, we introduce a general framework to parallelize a scientific calculation, which is applied to the CG-method (section III). Section IV describes the implementation of the parallel CG method on a Transputer network. The last section draws conclusions and gives directions for future research.

II A method to parallelize a problem for SPMD[III] computers

Parallelizing an algorithm and implementing it on a parallel computer is non-trivial and requires careful analysis. A general applicable methodology to parallelize scientific calculations would be very helpful. Here we introduce a generic scheme which captures several steps in going from the definition of a problem to an implementation on a parallel computer.

Parallelism is realized by decomposition of the problem.[2] This can be an algorithmic/task decomposition or a geometric/data decomposition. In the first case the problem is divided into a number of tasks that can be executed in parallel, in the second case the data is divided in groups (grains) and the work on these grains is performed in parallel. The parallel parts of the problem are assigned to processing elements. In most cases the parallel parts must exchange information. This implies that the processing elements must be connected in some way. Parallelizing a problem boils down to answering two questions: what is the best decomposition for the problem and which processor interconnection scheme is best suited.

Obviously we need a metric to decide which decomposition and interconnection scheme must be selected. We choose the total execution time of the parallel program T_{par}. The decomposition and interconnection that minimize T_{par} must be selected. The execution time T_{par}

II Vectors are written as lower case - and matrices as upper case bold faced characters.

III SPMD stands for Single Program Multiple Data, which implies a SIMD computer without global synchronization.

depends on two parameters:

$$T_{par} \equiv T_{par}(p,N) \; , \qquad [2]$$

where p is the number of processing elements and N a measure of the problem size.

We identify five steps between a problem definition and the implementation on a parallel machine, which are schematically shown in figure 1. Every step is independent of the other steps and can be studied without referring to the others. The first three parts ("problem", "methods", "algorithms") are also encountered if the application has to be solved on a sequential computer. The outcome of this part of the analysis can however be different if a sequential or parallel computer is used. This is an important point to remember if one starts to port sequential code to parallel machines; there is a chance that the underlying algorithms and methods are not suited to be parallelized efficiently.

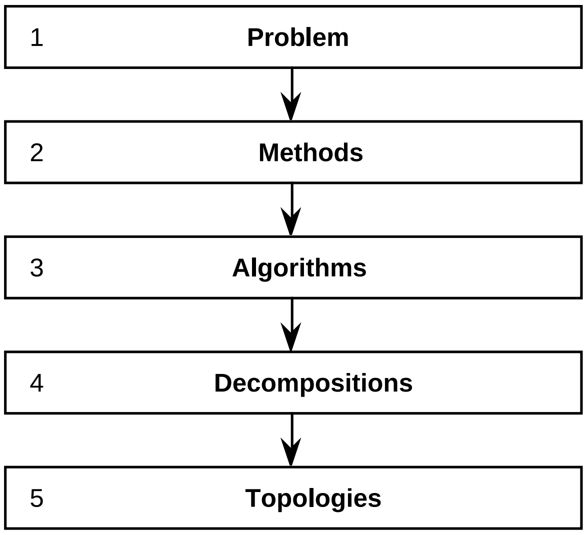


FIGURE 1: The five distinct steps that can be identified in the process of parallelizing a problem.

We will now investigate the various stages involved in the analysis in more detail:

1] The problem: We are faced with a scientific problem that needs large amounts of computing power. The problem is somehow translated into a mathematical formulation (sets of coupled ODE's or PDE's, sets of linear equations etc.).The formal definition often introduces constraints on the class of methods and algorithms that can be used to solve the problem.

2] Methods: The formal, mathematical problem can be solved by various methods. For instance, a large set of linear equations with a sparse coefficient matrix can be solved by several iteration schemes. Very often the formal definition of the problem reduces the number of methods. Further reduction of the collection of methods can be achieved by investigating some of their mathematical properties (such as e.g. stability). Usually this results in a small set of methods, which must be analyzed further.

3] Algorithms: Most methods can be implemented by several algorithms, that differ in stability, accuracy and time complexity. Very often a choice on the basis of stability or accuracy properties can be made. At this point the computation time on 1 processing element (T_{seq}) can be calculated and the fastest problem solver can be selected to be implemented on a sequential computer.

Very often this "best" sequential algorithm is ported to a parallel computer, without considering other methods/algorithms. This strategy simplifies the total analysis, but can be very misleading, as a simple example shows. Suppose you must solve a large, sparse set of linear equations and for some reason you can choose between a Jacobi iteration or a Gauss Seidel iteration. The Gauss Seidel iteration, which converges much faster than the Jacobi iteration, is the best choice on a sequential computer. However, in contrast to the Gauss Seidel iteration, the Jacobi iteration is inherently parallel.[12] Therefore, the total computation time of the Jacobi method, implemented on a parallel computer, can be significantly smaller than the computation time of the Gauss Seidel iteration.

Up to this point no reference to parallelism had to be made at all. The next step, the decomposition, brings us in the domain of parallel computing.

4] Decomposition: After the first three steps we have a set of methods, associated algorithms and expressions for $T_{seq}(N)$. Now the problem must be decomposed to allow distribution among several processing elements. Here it is not necessary to make any reference to a parallel computer. Yet one can derive a formal expression for $T_{par}(p,N)$:

$$T_{par}(p,N) = {T_{seq}(N)}/{p} + T_{calc.np}(p,N) + T_{comm}(p,N) \ . \qquad [3]$$

The decomposition, as was shown above, introduces communication between processing elements; $T_{comm}(p,N)$ is the total communication time of the parallel program. The time $T_{calc.np}(p,N)$ describes the non-parallel calculations. Many algorithms contain calculations that cannot be performed completely in parallel. For example, a parallel inner product is performed by decomposing the vectors in equal parts and assigning these parts to the processing elements. The inner products of the subvectors are calculated in parallel, but the resulting partial inner products in every processing element must be added to find the inner product. These remaining calculations cannot be performed completely in parallel. $T_{calc.np}(p,N)$ accounts for these non-parallel calculations. After decomposition a small set of communication primitives and non-parallel computations can be identified. Therefore $T_{comm}(p,N)$ and $T_{calc.np}(p,N)$ can be expressed as follows:

$$T_i(p,N) = \sum_j a_j t_j(p,N{:}topology) \ , \quad i \in \{calc.pn, comm\} \ , \qquad [4]$$

t_j is the time required to perform a specific communication (e.g. sending a vector from one processor to all other processors) or non-parallel calculations, and a_j is the number of times that specific action j is performed in the algorithm at hand. The t_j depend on the processor interconnection scheme (topology).

We believe that the number of distinct communication and non-parallel calculation primitives that may be identified after any decomposition is limited. In that case the summation in Eq. [4] runs over all the times t_j. The algorithm and decomposition fix the numbers a_j (where $a_j = 0$ is possible). In the future we will investigate this hypothesis in more detail.

Define $T_{loss}(p,N)$ by

$$T_{loss}(p,N) = p[T_{calc.np}(p,N) + T_{comm}(p,N)] \ . \qquad [5]$$

The expressions for the speedup S and efficiency ε of a parallel program[2] take the following form:

$$S(p,N) = \frac{T_{seq}(N)}{T_{par}(p,N)} = \frac{p}{1 + \frac{T_{loss}(p,N)}{T_{seq}(N)}} \ ; \qquad [6]$$

$$\varepsilon(p,N) = \frac{S(p,N)}{p} = \frac{1}{1 + \frac{T_{loss}(p,N)}{T_{seq}(N)}} \ . \qquad [7]$$

The next step is to find expressions for the topology-dependent t_j.

5] Topology: In the previous step knowledge of a parallel machine was not necessary. Now we must specify the topology of the parallel machine. For every topology an expression for t_j can be derived, depending on three parameters:

$$t_j = t_j(p,N:\tau_{calc},\tau_{startup},\tau_{comm}) \; ; \qquad [8]$$

τ_{calc}, $\tau_{startup}$ and τ_{comm} are hardware dependent parameters and denote the time for one floating point calculation, the startup time for communication and the time to communicate a word over a link connecting two processing elements respectively. As was mentioned above, we assume that it is possible to distinguish a relative small set of communication routines that may be identified in any parallel scientific calculation. Therefore it is possible to investigate the time complexity of alternative communication routines, implemented on different topologies, without reference to the application. This knowledge can then be utilized to find the best topology, in terms of minimizing $T_{par}(p,N)$, for the application under investigation. Furthermore one can provide an application programmer with libraries containing the most common communication routines, implemented for different topologies.

Now we have a number of abstract machines (described by the topology, τ_{calc}, $\tau_{startup}$ and τ_{comm}) and expressions for $T_{par}(p,N)$ for some decompositions and algorithms implemented on those machines. At first sight it seems that the number of combinations becomes too large. Fortunately, it is possible to discard combinations after every step, keeping the analysis manageable. Actual measurement of the τ_i's completes the analysis. Now the total computing time for the remaining combinations can be calculated, in a subspace of the total (p,N) space.

This scheme to parallelize a scientific calculation is very general and extensive. In many situations the analysis will not start from scratch at point 1 but (as in our case of the CG method) will start at point 3 or 4. The advantage of this scheme is a total decoupling of hardware and topology from decomposition, algorithms and methods.

The next section utilizes the concepts that were introduced here.

III A parallel Conjugate Gradient method

III.1 Introduction

The CG method[11] is a very powerful iterative method to solve large, sparse matrix equations, as equation [1]. Due to the special nature of the matrix **A** arising from the Coupled Dipole method (symmetric, non-Hermitian), equation [1] is first transformed to normal form:

$$\mathbf{A}^H\mathbf{A}\mathbf{x} = \mathbf{A}^H\mathbf{b} \; . \qquad [9]$$

(the subscript H denotes the Hermitian of **A**) and is subsequently solved by applying the CG method.

This paper investigates the usefulness of our parallelization methodology. As a test case we apply this method to the standard CG method for the normalized equation [9]. The iteration scheme of the standard CG method can be reformulated for the transformed system in such a way that it is not necessary to actually perform the matrix matrix product. The reformulated CG algorithm to the normal equation [9] is given below. In the sequel we will refer to this algorithm as the CG algorithm.

We start the process of parallelizing the CG method at point 4 of the previously introduced scheme: the decomposition.

We have investigated three decompositions (the rowblock, the columnblock and the grid decomposition) and three topologies (the binary tree, the ring and the cylinder topology). The full analysis for the rowblock decomposition on a ring topology will be presented here. For other decomposition/topology combinations we just present the results. The complete analysis will be published elsewhere.[13]

```
Reformulated CG algorithm for the normal equation

Initialize:   Choose a start vector $x_0$ and put
              $\mathbf{r}_0 = (\mathbf{b} - \mathbf{Ax}_0)$
              $\mathbf{p}_0 = \mathbf{A}^H\mathbf{r}_0$
Iterate:      while $|\mathbf{r}_k| \geq \varepsilon|\mathbf{b}|$
                  $\alpha_k = \frac{(\mathbf{A}^H\mathbf{r}_k)^H(\mathbf{A}^H\mathbf{r}_k)}{(\mathbf{Ap}_k)^H(\mathbf{Ap}_k)}$
                  $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k$
                  $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k(\mathbf{Ap}_k)$
                  $\beta_k = \frac{(\mathbf{A}^H\mathbf{r}_{k+1})^H(\mathbf{A}^H\mathbf{r}_{k+1})}{(\mathbf{A}^H\mathbf{r}_k)^H(\mathbf{A}^H\mathbf{r}_k)}$
                  $\mathbf{p}_{k+1} = \mathbf{A}^H\mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$

              stop $\mathbf{x}_k$ is the solution of $\mathbf{Ax} = \mathbf{b}$
```

The CG algorithm contains three distinct computational tasks per iteration:

1] Two matrix vector products, ($\mathbf{A}^H\mathbf{r}_k$ and $\mathbf{Ap}_k$);
2] Three inner products ($[\mathbf{A}^H\mathbf{r}_k].[\mathbf{A}^H\mathbf{r}_k]$, $[\mathbf{Ap}_k].[\mathbf{Ap}_k]$ and $\mathbf{r}_k.\mathbf{r}_k$);
3] Three vector updates ($\mathbf{x}_{k+1}$, $\mathbf{r}_{k+1}$ and $\mathbf{p}_{k+1}$).

The number of floating point operations on one processor is $8N^2$-2N for the matrix vector product (complex numbers!), 7N-1 for the inner products (complex multiplications, but addition of real numbers) and 8N for the vector updates. Therefore

$$T_{seq}(N) = (16N^2 + 41N - 3)\,\tau_{calc}\,, \qquad [10]$$

where we ignored the two divisions (α_k and β_k) and the squareroot operation (norm of $\mathbf{r}_k$). Note that we calculate the time per iteration step, thus neglecting the initialization. We expect the number of iteration to be large enough to allow for this approximation.

III.2 The decomposition

The CG algorithm contains many synchronization points, therefore task decomposition is not well suited here. The parallelism is introduced by data decomposition. The procedure is as follows: define a regular, static decomposition of matrix **A** and examine, on the basis of this decomposition, how the various vectors should be distributed among the processing elements. Furthermore, try to do as much calculations as possible in parallel. These demands imply nonparallel calculations and communication between processing elements, with associated times t_j, to be identified for the CG algorithm. The decomposition must be regular to avoid loadbalancing problems and static to avoid communication of large pieces of the matrix. The consequence of this static decomposition will be that vectors must be send through the network.

We first examine the matrix independent calculations of the CG algorithm (the vector updates and innerproducts). The demand is to do as much calculations as possible in parallel, therefore the vectors are divided in equal parts and distributed among the processing elements. We use a simple symbolic notation to visualize the decomposition of the vectors and the matrix, which is defined in appendix A. The vector update, in BLAS terminology a SAXPY, is performed as depicted in diagram 1:

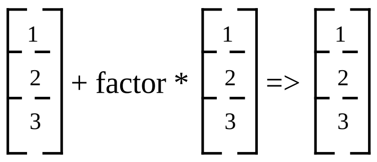


Diagram 1: parallel vector update

The vector update is performed completely in parallel, provided that the scalar "factor" is known in every processing element. The result of the vector update is also evenly distributed over the processing elements. The result vector is either used as input of a matrix vector product ($\mathbf{r}_{k+1}$ and $\mathbf{p}_{k+1}$) or is further processed after the CG algorithm has terminated ($\mathbf{x}_{k+1}$). The computing time for this parallel vector update (complex numbers !) is

$$[T_{par}(p,N)]_{\text{vector update}} = \frac{8N}{p}\tau_{calc} = \frac{[T_{seq}(N)]_{\text{vector update}}}{p} . \qquad [11]$$

The vector update, implemented in this way, is perfectly parallel (on any topology); $T_{loss} = 0$.

The number "factor" (α_k or β_k) is the result of parallel inner products, which is shown in diagram 2:

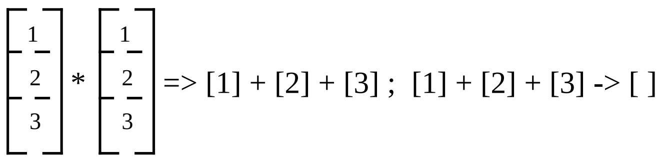


Diagram 2: the parallel innerproduct.

Every processor calculates a partial innerproduct, resulting in a partial sum decomposition of scalars. These scalars are accumulated in every processor and summed, resulting in an inner product, known in every processor. The total time for the parallel innerproduct is:

$$\begin{aligned}[T_{par}(p,N)]_{\text{inner product}} &= \left(\frac{7N}{p} - 1\right)\tau_{calc} + t_{sa} + t_{sa.np} = \\ &\frac{[T_{seq}(N)]_{\text{inner product}}}{p} + \left[\left(\frac{1}{p} - 1\right)\tau_{calc} + t_{sa.np}\right] + t_{sa} ,\end{aligned} \qquad [12]$$

where t_{sa} is the time for a <u>s</u>calar <u>a</u>ccumulate, the total communication time to send the partial inner products resident on each processing element to all other processing elements. The $t_{sa.np}$ is a computing time introduced by summing the partial inner products after (or during) the scalar accumulation. It is obvious that t_{sa} and $t_{sa.np}$ are topology dependent. The innerproduct is an example of a routine that, even in the absence of communication overhead, cannot be parallelized completely. The evaluation of the partial sum decomposition gives rise to calculations that cannot be performed in parallel. More general, every routine that introduces partial sum decompositions cannot be completely parallelized (independent of the type of parallel computer).

The execution times for the vector updates and the innerproducts are independent of the decomposition of the matrix. The two matrix vector products are the only matrix dependent parts of the algorithm. We will now examine the matrix vector product for a rowblock decomposed matrix.

The rowblock decomposition is achieved by dividing **A** in blocks of rows, with every block containing N/p consecutive rows, and assigning one block to every processing element. This is drawn schematically in diagram 3.

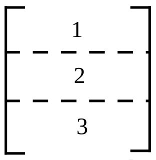


Diagram 3: the rowblock decomposition

Note that **A** is symmetric so that $\mathbf{A}^H$ is also decomposed in rowblock. This means that **A**×vector *and* $\mathbf{A}^H$×vector can be implemented in the same way. The kernel of the parallel matrix vector product is shown in diagram 4.

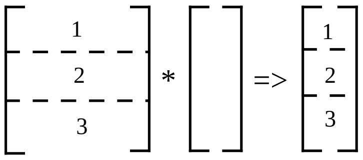


Diagram 4: the kernel of the parallel matrix vector product.

The argument vector must reside in memory of every processing element. However, this vector is always the result of a vector update, which is distributed among the processing elements (see diagram 1). Therefore, before calculating the matrix vector product, every processing element must gather the argument vector. The result is already decomposed in the correct way for further calculations (inner products or vector updates). Now we can draw the total diagram for the parallel matrix vector product (see diagram 5).

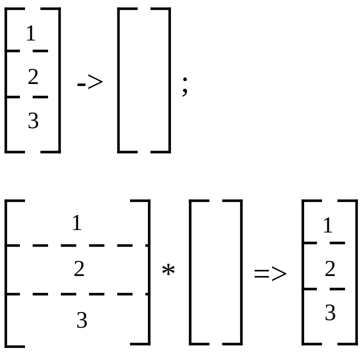


Diagram 5: the total parallel matrix vector product in the CG algorithm, for a row-block decomposed matrix.

The total elapsed time for this operation is

$$[T_{par}(p,N)]_{\text{matrix vector}} = \frac{8N^2-2N}{p}\tau_{calc} + t_{vg} = \frac{[T_{seq}(N)]_{\text{matrix vector}}}{p} + t_{vg}\ , \qquad [13]$$

with t_{vg} the communication time for the vector gather operation.

From Eq. [11, 12, 13] it is obvious that the total execution time per iteration can be expressed in the form of Eq. [3]. The term $T_{seq}(N)/p$ is equal for every decomposition and will be omitted in the rest of the analysis, $T_{loss}(p,N)$ must be minimized.

Table 1 lists the expressions for $T_{loss}(p,N)$ for the CG algorithm, as a function of the decomposition. The values for the coefficients a_j (see Eq. [4]) are 2 for the matrix vector product and 3 for the vector update and the inner product.

	$T_{loss}(p,N)$
row-block	$p\{3([\frac{1}{p} - 1]\tau_{calc} + t_{sa.np}) + 2t_{vg} + 3t_{sa}\}$
column-block	$p\{2(2N[\frac{1}{p} -1]\tau_{calc} + t_{va.np}) + 3([\frac{1}{p} - 1]\tau_{calc} + t_{sa.np}) + 2t_{va} + 3t_{sa}\}$
grid	$p\{2(2N[\frac{1}{p} - \frac{1}{\sqrt{p}}]\tau_{calc} + t_{pva.np}) + 3([\frac{1}{p} - 1]\tau_{calc} + t_{sa.np}) + 2t_{pvg} + 2t_{pva} + 3t_{sa}\}$

Table 1: expressions for $T_{loss}(p,N)$ for the CG algorithm, as a function of three different matrix decompositions; the subscript va stand for vector accumulate, pva for partial vector accumulate and pvg for partial vector gather[13]

Two conclusions can be drawn from Table 1:

1) If communication times can be neglected the rowblock decomposition has the smallest T_{loss}. The T_{loss} is only introduced by evaluating the partial scalar sum decompositions of the innerproducts. The grid decomposition and the columnblock decomposition also give rise to vector partial sum decompositions in the parallel matrix vector products. Their evaluation adds up to T_{loss}.

2) It can be shown that T_{loss} of the columnblock decomposition is always bigger than that of the rowblock decomposition. For every interconnection scheme of the processing elements the vector accumulate + scatter operation of the columnblock decomposition[13] involves messages that are bigger in size than, or equal to the messages in the vector gather operation of the row-block decomposition. Therefore

$\{T_{comm}\}^{\text{column-block}} \geq \{T_{comm}\}^{\text{row-block}}$,

furthermore

$\{T_{calc.overhead}\}^{\text{column-block}} > \{T_{calc.overhead}\}^{\text{row-block}}$,

so that

$\{T_{loss}\}^{\text{column-block}} > \{T_{loss}\}^{\text{row-block}}$.

Therefore, without refering to a parallel computer, we already know that the columnblock decomposition of **A** for the CG algorithm is less efficient than the rowblock decomposition.

III.3 The Topology

The communication times and nonparallel times in table 1 can only be specified with reference to a topology. Although it is not necessary to refer to specific hardware at this point of the analysis, we will restrict ourselves to the possibilities of the Transputer. Table 2 lists the topologies that can be realized with Transputers, assuming that every processing element consists of 1 Transputer and that the network contains at least 1 'dangling' (free) link to connect the network to a host computer.

The topologies are categorized in four groups; tree's, (hyper)cubes, meshes and cylinders. Two tree topologies can be build, the binary and ternary tree. The number of hypercubes is also limited. Only the cube of order 3 is in this group, cubes of lower order are meshes, whereas cubes of higher order cannot be realized with Transputers. Transputers are usually connected in mesh and cylinder topologies. Note that the pipeline and ring topology are obtained by setting $p_2 = 1$. The torus topology (mesh with wrap around in both directions) is not possible because this topology, assembled from Transputers, contains no dangling links.

We will present the time complexity analysis of the rowblock decomposition on a (bidirectional) ring. We have also analyzed the rowblock decomposition on a cylinder (a square mesh with wraparound) and a binary tree. The total time on the cylinder is higher than on the ring and the binary tree. The time on the last two is comparable. Furthermore we investigated the grid decomposition on a cylinder. The details of this analysis can be found elsewhere,[13] but the results will be compared with the rowblock decomposition on a ring.

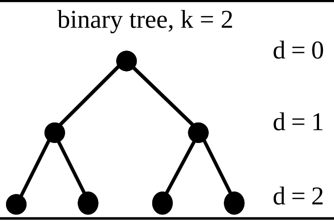
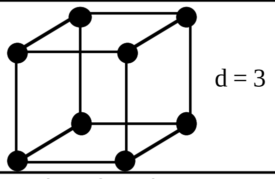
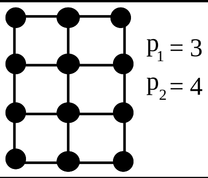
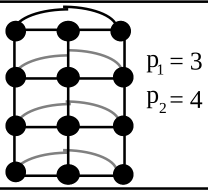
Topology	Order	Number of processors (p)	Diameter	Comment	Example
k-tree	d	$\frac{1}{k-1}k^{d+1} - \frac{1}{k-1}$	2d	$2 \leq k \leq 3$	
Hypercube	d	2^d	d	$d \leq 3$	
Mesh		$p_1 x p_2$	$(p_1-1) + (p_2-1)$	p_1: number of processors in a row. p_2: number of processors in a column.	
Cylinder		$p_1 x p_2$	$\lfloor 1/2(p_1) \rfloor + (p_2-1)$	Wrap around in p_1 direction	

Table 2: Overview of topologies that can be realized with Transputers. The black dots are processing elements (PE). Every PE consists of 1 Transputer. The network must contain at least 1 free link to connect to a host computer. All links are bidirectional, $\lfloor\ \rfloor$ is the floor function.

The row-block decomposition gave rise to a vector gather operation;

$\begin{bmatrix} 1 \\ \hline 2 \\ \hline 3 \end{bmatrix}$ -> $\begin{bmatrix} \\ \\ \\ \end{bmatrix}$,and a scalar accumulate; [1] + [2] + [3] -> [ ] ,

with associated overhead calculations. During the vector gather operation, each processor receives from all other processors in the network a piece of the vector, which is subsequently stored. After the gather, every processor has a copy of the total vector in its local memory. On the bidirectional ring this is achieved as follows (also see Fig. 2):
1) In the first step each processor sends its private part of the vector to the left and the right and, at the same time, receives from the left and the right processor their private part.
2) In the following steps, the parts received in the previous step, are passed on from left to right and vice versa, and in parallel, parts from left and right are received.
After $\lfloor p/2 \rfloor$ steps, each processor in the ring received the total vector (with $\lfloor\ \rfloor$ the floor function).
If we assume that point to point communication of n floating points takes a time

$$t_{\text{point-to-point}} = \tau_{\text{startup}} + n\tau_{\text{comm}} \ , \qquad [14]$$

then $t_{vg}(p,N)$ is

$$t_{vg}^{\text{ring}}(p,N) = \left\lfloor \frac{p}{2} \right\rfloor \left(\tau_{\text{startup}} + 2\frac{N}{p}\tau_{\text{comm}}\right) = \left\lfloor \frac{p}{2} \right\rfloor \tau_{\text{startup}} + \left\lfloor \frac{p}{2} \right\rfloor \frac{2N}{p}\tau_{\text{comm}} \ , \qquad [15]$$

During every communication step each processor simultaneously sends 2 messages and receives 2 messages of N/p complex numbers. A total of $\lfloor p/2 \rfloor$ steps are performed.

	processor (i-1)	processor (i)	processor (i+1)
step 1	<-[i-1]->	<-[i]->	<-[i+1]->
....			
step k	<-{[i+k-2] mod p} {[i-k] mod p}->	<-{[i+k-1] mod p} {[i-k+1] mod p}->	<-{[i+k] mod p} {[i-k+2] mod p}->

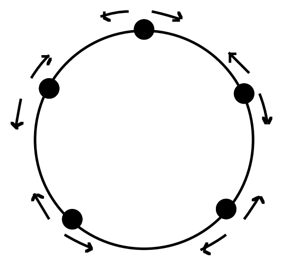


FIGURE 2: The vector gather operation on the bidirectional ring. Each processor simultaneously sends to and receives from both neighboring processors a part of the vector. The -> symbol gives the communication direction, the number between the brackets is the processor number where the package originally came from.

The time for the scalar accumulate is easily derived from the previous analysis. Instead of a complex vector of N/p elements, a real number is sent, the communication pattern is the same as the vector gather. After each communication step, the newly received partial inner products are added to the partial inner product of the receiving processor. In total p real numbers are added. This leads to the following expressions for t_{sa} and $t_{sa.np}$:

$$t_{sa}^{ring} = \left\lfloor \frac{p}{2} \right\rfloor (\tau_{startup} + \tau_{comm}) \, , \qquad [16]$$

$$t_{sa.np}^{ring} = (p - 1)\tau_{calc} \, . \qquad [17]$$

The expressions for T_{loss} in Table 1 can now be further specified, using the expressions for the communication and overhead times derived in the previous paragraphs. Equation [18] gives T_{loss} for the row-block decomposition implemented on a ring topology:

$$\frac{(T_{loss})_{ring}^{row\text{-}block}}{p} = 3\{p + \frac{1}{p} - 2\}\tau_{calc} + \{5\left\lfloor \frac{p}{2} \right\rfloor\}\tau_{startup} + \{\left\lfloor \frac{p}{2} \right\rfloor(\frac{4N}{p} + 3)\}\tau_{comm} \, . \qquad [18]$$

The complete expression for T_{loss} for the grid decomposition on a cylinder is given below:[13]

$$\frac{(T_{loss})_{cylinder}^{grid}}{p} = \{3(2\sqrt{p} + \frac{1}{p} - 3) + 4N(1 - \frac{1}{\sqrt{p}})^2\}\tau_{calc} + \{5\left\lfloor \frac{\sqrt{p}}{2} \right\rfloor + 7\sqrt{p} - 7\}\tau_{startup} +$$

$$\{4N(\frac{1}{\sqrt{p}}\left\lfloor\frac{\sqrt{p}}{2}\right\rfloor + \frac{2}{\sqrt{p}} - \frac{2}{p}) + 3(\left\lfloor\frac{\sqrt{p}}{2}\right\rfloor + \sqrt{p} - 1)\}\tau_{comm} , \qquad [19]$$

The rowblock-ring combination and the grid-cylinder combination can be compared for limiting cases. In our application an interesting limiting value is large N. A close look at Eqs.[18, 19] learns that for large N T_{loss} for the rowblock-ring combination is smaller than for the grid-cylinder combination. In the next section experimental values for τ_{calc}, $\tau_{startup}$ and τ_{comm} are presented and Eqs.[18, 19] are studied in detail.

III.4 The hardware parameters

The model parameters τ_{calc}, $\tau_{startup}$ and τ_{comm} depend on the hardware, on runtime environment and on compilers. Direct derivation of these parameters from hardware specifications alone gives too optimistic (small) values. Therefore reliable numerical values of the model parameters can only be obtained by actual measurements on the parallel computer.

In our department we have access to a Sun hosted Meiko Computing Surface[14] containing 64 T800 Transputers with 4 Mb local memory per processor. The parallel CG method will be implemented in Occam2, under OPS (Occam2 Programming System). The presented values for τ_i are only applicable to parallel programs written in Occam2, running under the OPS on a Meiko Computing Surface. We have measured the following values[15]

τ_{calc} = 2.7 μs;
$\tau_{startup}$ = 13.3 μs;
τ_{comm} = 7.2 μs.

Now we can calculate T_{loss} as a function of p and N. Figure 3 gives T_{loss} as a function of N, for p equals 16 and 64, both for the rowblock-ring and the grid-cylinder combination. For very small N the grid-cylinder has a smaller T_{loss}. However, if N grows the T_{loss} of the rowblock-ring combination is always smaller than for the grid-cylinder combination. This was already concluded in the previous section. For realistic problems the rowblock decomposition implemented on a ring topology is the best choice for the CG algorithm.

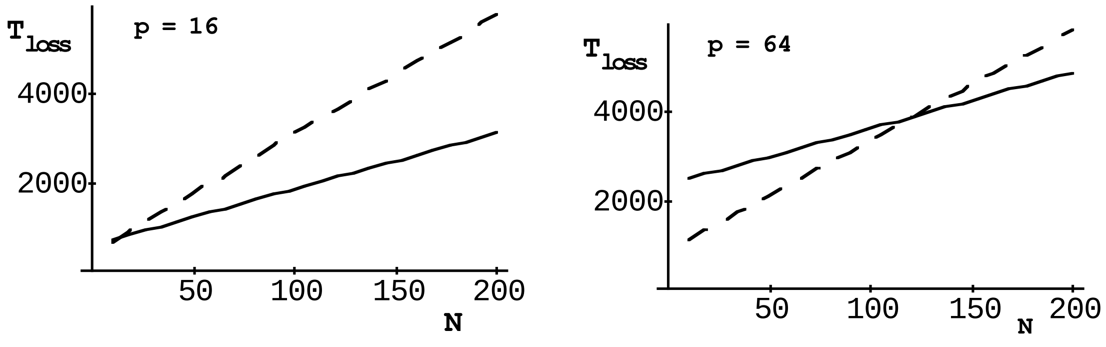


Figure 3: T_{loss} for the rowblock ring (full line) and the grid-cylinder (dashed line) for 16 and 64 processors, as a function of N. T_{loss} is in microseconds.

Figure 4 gives the values of N where T_{loss} for the grid-cylinder combination equals T_{loss} for the rowblock-ring combination. Even for a large number of processors the rowblock-ring combination is to be preferred.

This concludes the analysis. We implemented the CG algorithm, with a row-block decomposition of the matrix, on a bidirectional ring of Transputers. Figure 5 gives the theoretical efficiency for this implementation in a small part of the (p,N) domain. The efficiency of the implementation will be close to 1 if N grows. This is to be expected, since T_{seq}/p is an $O(N^2)$ function, whereas T_{loss}/p is of O(N). From figure 5 it is obvious that for relative small values of N this almost perfect efficiency can be obtained.

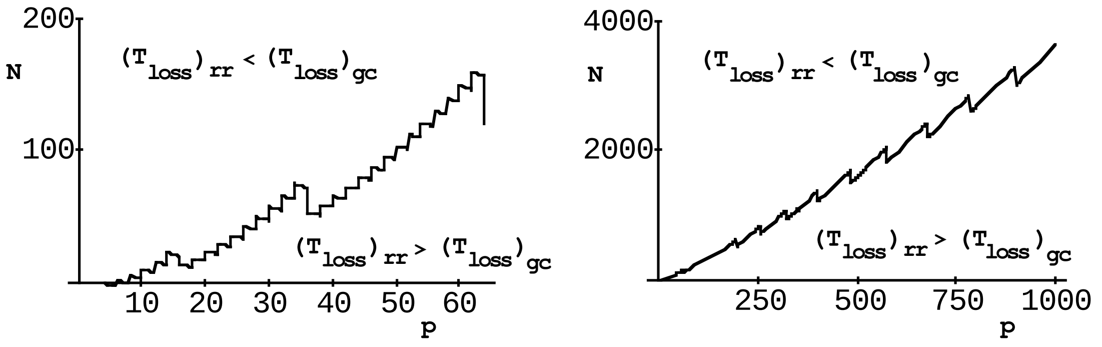


Figure 4: A comparison between the rowblock-ring and the grid-cylinder combinations. The line indicates the (p,N) combinations where T_{loss} for both combinations is equal.

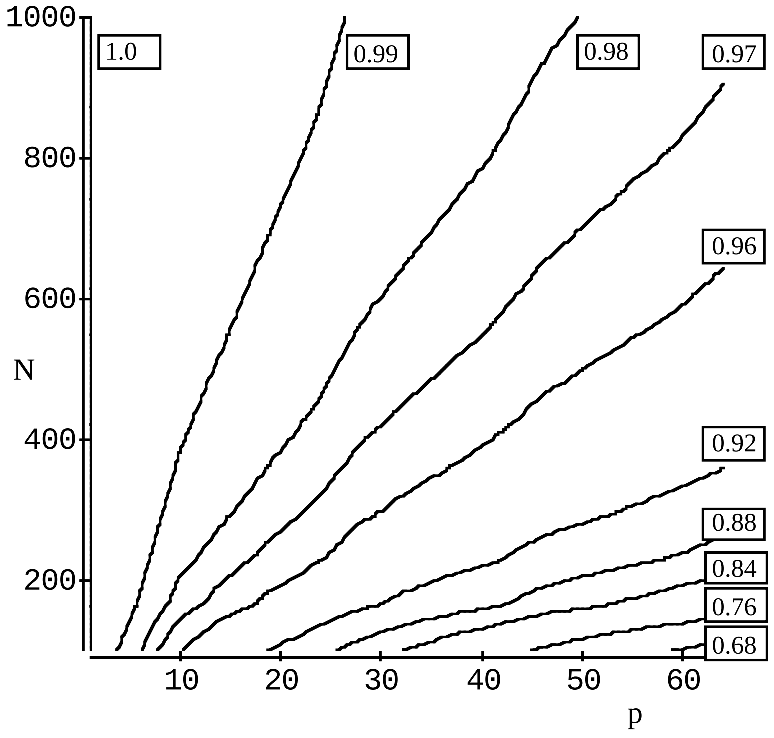


Figure 5: A contourplot of the theoretical efficiency of the CG algorithm, with a row block decomposed matrix, implemented on a ring of Transputers (Meiko computing surface, Occam2, under the OPS).

IV Implementation on a Transputer network

IV.1 Introduction

We have implemented the CG algorithm, with rowblock decomposition of **A**, on a bidirectional ring of Transputers. The implementation language was Occam2. This section describes some implementation issues and presents preliminary measurements of $T_{par}(p,N)$. These performance measurements are compared with the theoretical expression for T_{par}, derived in the previous section.

IV.2 The Implementation

From the analysis in the previous section it is clear that a strict distinction between calculation routines and communication routines plus associated nonparallel calculations can be obtained. Therefore we defined two parallel processes on each Transputer in the ring: a calculator process and a router process (see Fig. 6).

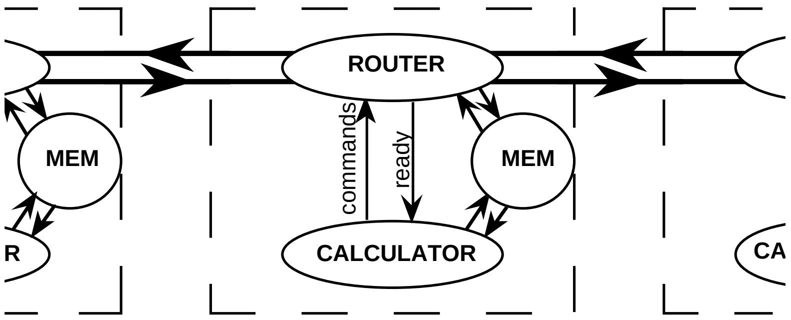


Figure 6 Main processes and streams in a Transputer, mem is the local memory of the Transputer.

The calculator process takes care off the calculations (such as vector updates or the matrix vector products) and issues commands to the router process to start a particular communication routine (such as the vector gather). The router process receives data from and sends data to the neighboring Transputers and stores the received data into local memory. After termination of the communication routine, the router sends a ready signal to the calculator, which proceeds with the next step of the iteration.

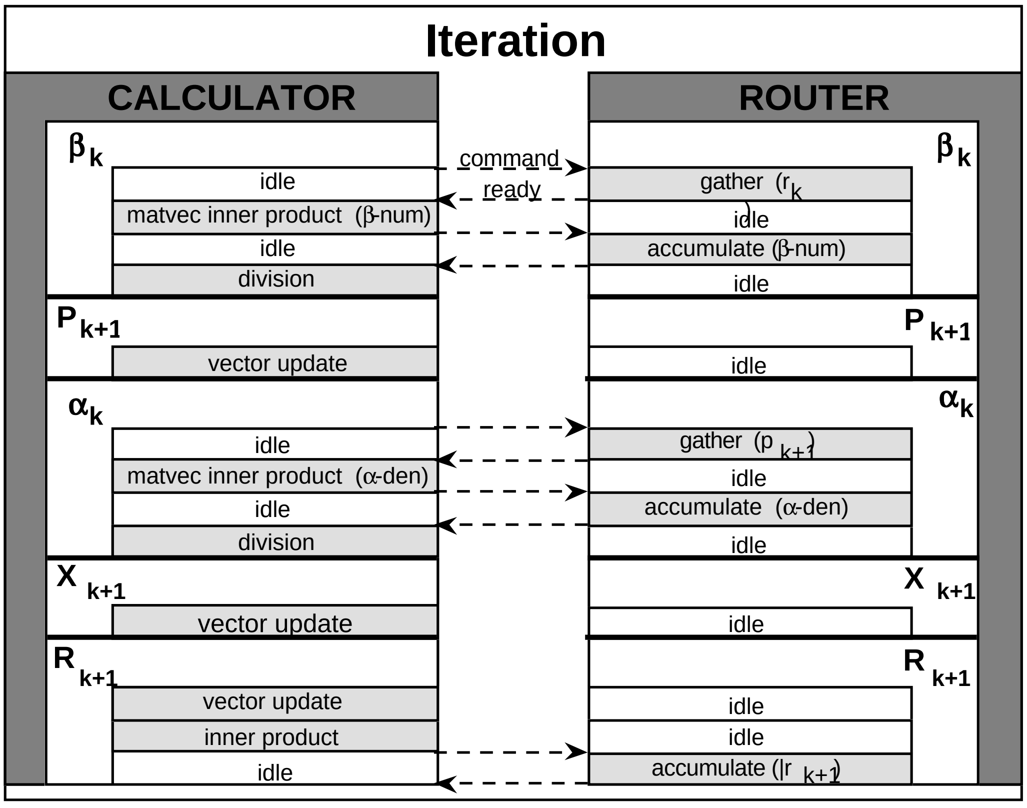


Figure 7. Diagram of the iteration part of the parallel CG algorithm.

This organization of the parallel program facilitates a clear and structured program development. Furthermore, in this way the powerful property of the Transputer to operate the CPU, FPU and link interfaces in parallel[16] can be programmed very easy and efficiently.

The calculator and router process consist of several procedure calls. Figure 7 schematically draws the implementation of the iteration of the CG method. Note that the implementation of the CG algorithm, compared to the formulation in the previous section, is in a slightly different form. As soon as the residual vector $\mathbf{r}_k$ is calculated, the convergence test is performed.

The implementation of the parallel CGmethod was tested by comparing it with an implementation on a sequential computer (SUN Sparc station). First the implementation was tested on 1 Transputer, thus verifying the correctness of the calculation process. This was followed by test runs on more Transputers to verify the setup and communication routines.

IV.3 Performance measurements

Here we present preliminary results of the measurement of $T_{par}(p,N)$, the time per iteration as a function of p and N, and comparison of the results with theory. Figure 8 gives the results for N equals 24, 219 and 495, and $1 \leq p \leq 64$. The relative difference between theory and experiment is approximately 4%.

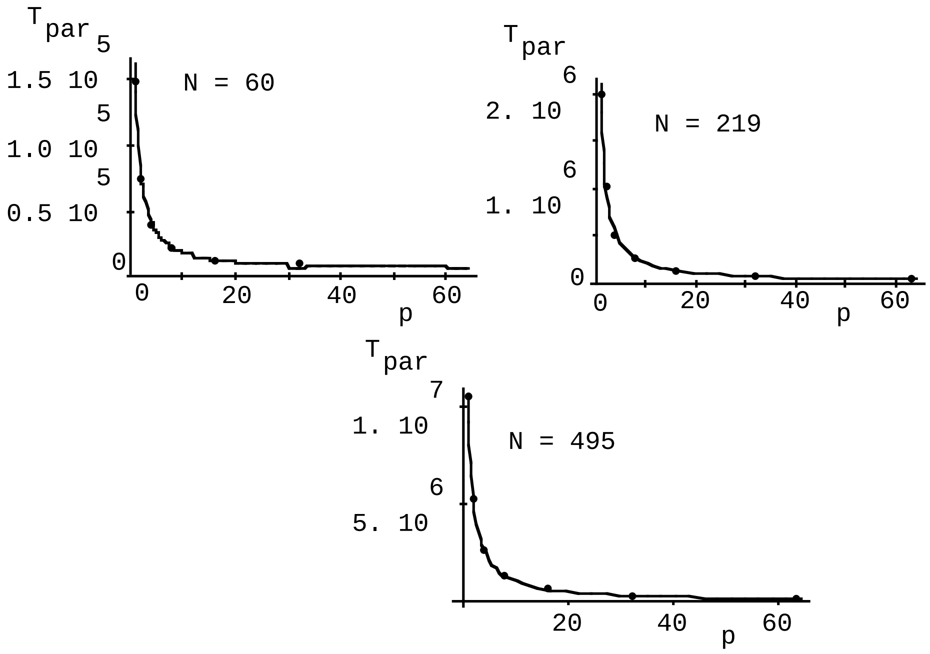


Figure 8: Measurements and calculations of T_{par}, the time per iteration, as a function of p, the number of processors. The line is theory, the dots are the measurements.

V Discussion and conclusions

We have theoretically investigated, and experimentally verified the time complexity of parallel implementations of a CG method. This research is part of a larger project, where the ELS properties of micron sized particles are investigated by means of large (parallel) computer simulations.

First we introduced a concept to come from a problem definition to an implementation on a distributed memory computer. This method consists of five self contained steps. Every step can be viewed as a mapping. The mapping of algorithms on decompositions and the mapping of decompositions on topologies are two new steps if one moves from sequential to parallel computing. We hypothesized that for scientific calculations the number of these two mappings is finite and relative small. If these mappings can be identified it is possible to formally investigate their time complexity, without referring to the underlaying algorithms, methods and applications.

We introduced a formal expression for the total execution time of a parallel program (Eqs. [3, 4]). This expression is very useful to analyze the previously mentioned mappings. The parameter $T_{loss}(p,N)$ (Eq. [5]) directly identifies the parts of the program that reduce the efficiency of the parallel program.

As a test case we parallelized the standard CG method for the normal equation with the introduced approach. First we investigated three decompositions of the matrix **A**; the rowblock, columnblock and grid decomposition. The first two decompositions explicitly used the symmetry property of **A**, whereas the results of the grid decomposition are applicable to any non Hermitian matrix. Without knowledge of the parallel computer it is possible to show that the rowblock decomposition is always more efficient than the columnblock decomposition. This shows the strength of the applied parallelization scheme. It is possible to compare different parallelizations at a very high level, completely decoupled from any reference to a particular (abstract) machine.

We proceeded with the rowblock - and grid decomposition where the mapping of the rowblock decomposition on a ring and the grid decomposition on a cylinder was investigated. In the absence of communication time or for large N, the rowblock-ring combination is most efficient, for any value of the hardware parameters τ_{calc}, $\tau_{startup}$ and τ_{comm}. We measured the hardware parameters and investigated the T_{loss} for both combinations in more detail. If N grows, the rowblock-ring is always better. However, it is important to realize that if N becomes large, the term $T_{loss}(p,N)$ can be neglected compared to $T_{seq}(N)/p$, both for rowblock-ring and for grid-cylinder. For very large problems, both combinations will have a comparable efficiency. We have implemented the row-block ring combination because of a number of pragmatic reasons:

- it contains less different communication routines, thus simplifying the implementation;
- a ring can be implemented with a lower τ_{comm} on the Meiko computing surface than a grid, due to the physical wiring of the Transputers inside the Meiko computing surface;[15]
- a ring topology allows one to grab any number of Transputers p, whereas the cylinder topology requires p to be equal to k^2 (k = 2,3,4...). Especially in multi-user ("multi-domain") environments, like the Meiko, this is a severe constraint if p is large.

The CG algorithm, with rowblock decomposition of the matrix, was implemented on a ring of Transputers, in Occam2 with the Occam2 Programming System. We presented preliminary measurements that were compared with the theory. The agreement was within approximately 4%.

The standard CG method, applied to the normal equation is computational more intensive and less accurate than dedicated CG methods for complex, symmetric matrices (such as e.g. COCG[17]). Therefore, in the future we will analyze and implement such a dedicated method. All these algorithms consist only of matrix vector products, vector updates and inner products.[18] This means that we already have identified the t_j(p,N:topology) for three decompositions of **A** and that we have expressions for those t_j on a ring and cylinder topology. Therefore the time complexity analysis will be very straightforward. Furthermore, we have implemented the communication routines appearing in the rowblock decomposition on a ring topology. Therefore, implementing a new algorithm with rowblock decomposition on a ring can be realized very fast, simply by adapting the calculator process (see Fig. 6).

We introduced a methodology to analyze applications that have to be parallelized for local memory multiprocessors. The usefulness and strength of this formal approach was demonstrated by utilizing it for an application emerging from ELS research. In the future we will elaborate the concept of a finite set of communication routines introduced by decomposition of the problem, and apply it to a variety of applications.

acknowledgements

We wish to thank Dr. W. Hoffmann of the Parallel Scientific Computing group, Faculty of Mathematics and Computer Science, University of Amsterdam, for valuable suggestions and careful reading of the manuscript.

references

1] R.W. Hockney and C.R. Jesshope, *Parallel Computers 2,* (IOP Publisher Ltd, 1988).

2] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving problems on concurrent processors, volume 1, general techniques and regular problems*, (Prentice-Hall International Editions, 1988).

3] P.M.A. Sloot, M.J. Carels and A.G. Hoekstra, "Computer Assisted Centrifugal Elutriation," (In *Computing Science in the Netherlands* , 1989).

4] P.M.A. Sloot and C.G. Figdor, "Elastic light scattering from nucleated blood cells: rapid numerical analysis," Applied Optics **25**, 3559-3565 (1986).

5] P.M.A. Sloot, A.G. Hoekstra and C.G. Figdor, "Osmotic Response of Lymphocytes Measured by Means of Forward Light Scattering: Theoretical Considerations," Cytometry **9**, 636-641 (1988).

6] A.G. Hoekstra, J.A. Aten and P.M.A. Sloot, "Effect of aniosmotic media on the T-lymphocyte nucleus," Biophysical Journal **59**, 765-774 (1991).

7] P.M.A. Sloot, M.J. Carels, P. Tensen and C.G. Figdor, "Computer-assisted centrifugal elutriation.I. Detection system and data acquisition equipment," Computer Methods and Programs in Biomedicine **24**, 179-188 (1987).

8] P.M.A. Sloot, A.G. Hoekstra, H. van der Liet and C.G. Figdor, "Scattering matrix elements of biological particles measured in a flow through system: theory and practice," Applied Optics **28**, 1752-1762 (1989).

9] E.M. Purcell and C.R. Pennypacker, "Scattering and absorption of light by nonspherical dielectric grains," The Astrophysical Journal **186**, 705-714 (1973).

10] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, (Springer-Verlag, 1980).

11] M.R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for solving linear systems," Nat. Bur. Standards J. Res. **49**, 409-436 (1952).

12] J.J. Modi, *Parallel Algorithms and Matrix Computation,* (Oxford University Press, 1988).

13] A.G. Hoekstra and P.M.A. Sloot *To be published*. .

14] W. Hoffmann and K. Potma *Experiments with Basic Linear Algebra Algorithms on a Meiko Computing Surface*, Tech. Rept. CS-90-02, Department of Mathematics and Computer Science, University of Amsterdam, 1990.

15] M.J. de Haan, *Parallel implementation of the standard Conjugate Gradient method on a ring of transputers.,* Master's thesis, Technical Highschool Alkmaar, 1991.

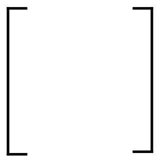
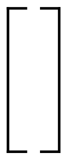
16] D.A.P. Mitchell, J.A. Thompson, G.A. Manson and G.R. Brookes, *Inside the Transputer,* (Blackwell Scientific Publications, 1990).

17] H.A. van der Vorst and J.B.M. Melissen, "A Petrov-Galerkin type method for Solving Ax=b, where A is symmetric complex," IEEE Trans. Mag. **26**, 706-708 (1990).

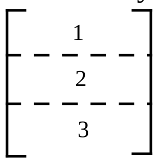
18] S.F. Ashby, T.A. Manteuffel and P.E. Saylor, "A taxonomy for conjugate gradient methods," Siam J. Numer. Anal. **27**, 1542-1568 (1990).

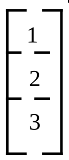
Appendix A] Symbolic notation to visualize the data decomposition

A matrix and vector are denoted by a set of two brackets:

matrix:  ,and vector:  .

The decomposition is symbolized by dashed lines in the matrix and vector symbols, as in



and


.

The matrix and vectors are divided in equal parts and distributed among the processing elements. The decomposition is drawn for three processing elements, but is extended to p processing elements in a straightforward way. The decomposition of the matrix is static, it remains the same during the computation. The decomposition of the vectors can take three distinct forms, depending on the calculation that was or has to be performed:

1] The vector is known in every processing element: $\left[\begin{array}{c} \\ \\ \\ \end{array}\right]$;

a special case is a scalar known in every processing element, which is depicted by [ ] ;

2] Parts of the vector are distributed among processing elements: $\left[\begin{array}{c}1\\ \hline 2\\ \hline 3\end{array}\right]$;

3] Every processing element contains a vector, which summed together give the original vector. This "decomposition" is referred to as partial sum decomposition. This partial sum decomposition usually is the result of a parallel matrix vector product;

the partial sum decomposition: $\left[\begin{array}{c}1\end{array}\right] + \left[\begin{array}{c}2\end{array}\right] + \left[\begin{array}{c}3\end{array}\right]$.

A special case is a partial sum decomposition of scalars, which is the result of parallel innerproducts. This will be depicted by [1] + [2] + [3]. Furthermore a mix of [2] and [3] is possible. The '-> ' denotes a communication, e.g.

$$\left[\begin{array}{c}1\\ \hline 2\\ \hline 3\end{array}\right] \text{->} \left[\begin{array}{c} \\ \\ \\ \end{array}\right]$$

represents a vector gather operation, that is every processing element sends its part of the vector to all other processing elements. The '=>' denotes a calculation, e.g.

$$\left[\begin{array}{c}1\\ \hline 2\\ \hline 3\end{array}\right] + \left[\begin{array}{c}1\\ \hline 2\\ \hline 3\end{array}\right] \Rightarrow \left[\begin{array}{c}1\\ \hline 2\\ \hline 3\end{array}\right]$$

is a parallel vector addition.