COMMISSION OF THE EUROPEAN COMMUNITIES

****

ESPRIT III

PROJECT NB 6756

****

CAMAS

COMPUTER AIDED MIGRATION OF APPLICATIONS SYSTEM

****

CAMAS-TR-2.1.2.1

BLAS Progress Report

****

Date : SEPTEMBER 1993

Rev. 1.0

ACE - U. of AMSTERDAM - ESI SA - ESI GmbH - FEGS - PARSYTEC -
U. of SOUTHAMPTON.

Technical Report CAMAS TR 2.1.2.1

BLAS

M. Bergman and P.M.A. Sloot
University of Amsterdam
September 1993

1. Introduction

Finite Element (FE) methods can be employed for a large range of applications; many problems in fields like material science, aerodynamics, high energy physics and engineering are solved with FE methods. CAMAS concentrates on one class of applications: structural mechanics. In particular the PAMCRASH program, a simulation of the crashworthiness of vehicles, plays an important part within CAMAS. The program is an example of an application within the structural mechanics that exploits a FE method.

FE methods can be divided into explicit and implicit methods [1, 2]. The type used in the PAMCRASH code is an explicit method. With this type the time steps in the program should be small in order to get reliable results. In each time step new results are obtained based on the results of the previous time step. For this reason there is no need for a factorisation of a stiffness matrix. Inherent to explicit methods is the fact that all intermediate results are calculated. Obviously, this will affect efficiency as a large number of steps have to be calculated. Implicit FE methods, on the other hand, allow larger time steps, so reducing the number of steps to run a simulation. However, in each time step the stiffness matrix is involved in the computation of intermediate results. In order to compute the intermediate results a system of equations of the form $\mathbf{Ax} = \mathbf{b}$, with $\mathbf{A}$ the stiffness matrix, has to be solved. It is this kind of additional numerical problems that is addressed by the Basic Linear Algebra Subprograms (BLAS). In this particular application an implicit FE method benefits the efficiency of the program. For many other applications in steady state physics there is no need for small time steps. For these, implicit FE methods will be preferable to explicit methods.

One of the goals of the CAMAS project is to give a prediction of the performance of a program that is to be parallelised for various platforms. The workbench that is under development in the project comprises a set of tools that can be used to predict the performance of a program on an existing or virtual parallel machine. Two of these tools are SAD and PARASOL. The SAD tool gives an abstract time complexity description of the program. In general this description will contain elements that can be expressed in terms of a number of (system) parameters and elements that cannot be determined a priori. In SAD the latter are formulated in terms of probabilities and stochastic variables. The aim of PARASOL is to isolate a number of machine parameters, such as communication, processor and cache parameters, that can be used in combination with the results of SAD. Depending on the input parameters a prediction for the performance of the program will be given.

In this paper the time complexity of the algorithm, that will be used for the parallel solver of

the system of equations, will be the central issue. This solver will turn out to be the most critical part of implicit FE methods. As the time complexity of the solver will be completely determinable, it is well-suited for validating the results as obtained from SAD and PARASOL. This design strategy of comparing predicted results with results that are obtained from a complex but completely determined problem can also be found in [3].

The rest of this paper is organised as follows. In the next section a brief description of FE methods will be given, followed by an inventory of linear solvers that can be exploited for these methods. After the selection of a linear solver the chosen solver will be explained in detail in section 3. In section 4 parallelisation methods for the solver will be discussed and an initial description of the time complexity of the algorithm wil be formulated.

2. FE methods and linear solvers

Typical FE programs consist of four independent modules [4]: 1) a pre-processing step including the generation of the element stiffness matrices and force vectors; 2) Assembly of the global system of equations given as **Ax** = **b**, where **A** is the stiffness matrix, **b** is the known force vector and **x** is the unknown vector of displacements; 3) solution of the global system for the nodal point displacements; 4) a post-processing step which generally includes the calculation of stress and strain quantities within each element.

The overall performance of a FE program is determined by the costs associated with each of the steps. To illustrate that the most critical step is the third one, consider a general and regular $n \times n$ mesh. When the nodal points are numbered from left to right starting at the lower left corner, each point in the mesh can be uniquely identified with a number in the range from 1 to n^2. The stiffness matrix consists of the interactions between all points, i.e. the matrix element k_{ij} describes the interaction of points i and j. In general, only the neighbouring nodal points are of interest to determine the displacement of a particular point. With the above described numbering the stiffness matrix will then be symmetrical and banded with a bandwidth n. The order of the matrix equals the total number of mesh points n^2. In figure 1 the upper triangle of the stiffness matrix is shown, where $N = n^2$ is the order of the matrix.

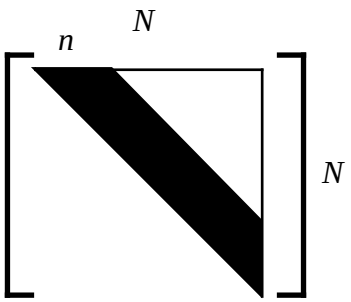

Figure 1: stiffness matrix for a general $n \times n$ mesh

The computational costs for the generation of the stiffness matrix, the assembly of the global system and the calculation of the element strains and stresses are all directly proportional to the number of elements. Thus, for each of these steps the computational costs will be of order N.

The general system of linear equations **Ax** = **b** can be solved in many ways. When for example Gaussian elimination is used, the cost for the factorisation of the matrix is proportional to N^2 and for the forward and backward substitution proportional to $N\sqrt{N}$. Other linear solvers will need in general even more computing effort to come to a solution. For these the cost of the

solution of the system might even be proportional to N^3. This is also the case when the numbering of the nodal points is not optimal and the stiffness matrix is not banded.

From the above it is clear that for an increasing number of grid points the cost of the solution of the system of equations will dominate, even if the nodal points are numbered optimally. It is for this reason that the choice and implementation of a linear solver is crucial.

To solve linear systems either direct or iterative methods can be employed. Direct methods, such as Gaussian elimination, require in general $O(N^3)$ floating point operations to find a solution for a $N \times N$ matrix. Only for some special linear systems the number of floating point operations can be reduced somehow. Another drawback of direct methods is that the complete (factorised) matrix should be kept in memory. Iterative methods, on the other hand, require only N^2 floating point operations per iteration and do not need keeping the whole matrix in memory. In figure 2 a number of properties of a direct method and several iterative methods are listed [5].

	Gaussian elimination	Jacobi iteration	Gauss-Seidel iteration	Conjugate Gradient
Stability	++	+	+++	++++
Parallelization	++	++++	++++	++
Convergence	+++	++	+++	++++

Figure 2: properties of several linear solvers

The stability of Gaussian elimination and Jacobi iteration is not very good. Round off errors may have a great impact on the solution and will lead to incorrect results. Gauss-Seidel iteration and the Conjugate Gradient method, on the other hand, are much more stable. The parallelisation of Jacobi and Gauss-Seidel iteration is straightforward and can be easily accomplished. For the other two methods parallelisation is not trivial. The best convergence of the four listed linear solvers is attained by the Conjugate Gradient method. For this method it can be proved that the number of iterations to come to a solution is at most N. However, if the spectral radius and the condition number of the system matrix satisfy some special criteria, the number of steps to find an acceptable solution (i.e. a solution satisfying some accuracy criterion) will be much smaller than N.

For many problems that are solved by means of FE methods the stiffness matrix will be very large: values of N of order 10^4 or 10^5 are no exceptions. When the problem size is this large direct methods to solve the linear system will fail for the above mentioned reasons. Therefore iterative methods should be used instead. The reliability and the speed of convergence of the Conjugate Gradient method are factors making this iterative method preferable to the others. In the next section the Conjugate Gradient method will be discussed extensively.

3. The Conjugate Gradient method

The Conjugate Gradient method is a very powerful method for solving large systems of linear

equations. Usually the method is applied to systems with large sparse matrices, like the ones arising from discretisations of partial differential equations. Formally spoken it is not correct to speak of *the* Conjugate Gradient method. Instead, a number of iterative solvers based on a common principle are referred to as Conjugate Gradient methods. The original Conjugate Gradient method of Hestenes and Stiefel [6] (the CGHS method in the taxonomy of Ashby [7]) is only valid for Hermitian positive definite (hpd) matrices. Although this method can be applied directly to solve the system of linear equations it will not be adopted here. Instead we will use the Preconditioned Conjugate Gradient (the PCG method in the taxonomy of Ashby) method. This method is very similar to the original Conjugate Gradient method, except that it uses preconditioning. The algorithm for the PCG method is shown below:

The PCG algorithm

Initialise:	Choose a start vector $\mathbf{x}_0$ and put	
	$k = 0$	
	$\mathbf{r}_0 = (\mathbf{b} - \mathbf{A}\mathbf{x}_0)$	calculate the residual vector
	$\mathbf{p}_0 = \mathbf{s}_0 = \mathbf{C}^{-1}\mathbf{r}_0$	the first direction vector
Iterate:	**while** $\lvert\mathbf{r}_k\rvert \geq \varepsilon \lvert\mathbf{b}\rvert$	iterate until the norm of the residual vector is small enough
	$\alpha_k = \dfrac{(\mathbf{r}_k, \mathbf{s}_k)}{(\mathbf{A}\mathbf{p}_k, \mathbf{p}_k)}$	
	$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$	calculate new iterate
	$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$	update residual vector
	$\mathbf{s}_{k+1} = \mathbf{C}^{-1}\mathbf{r}_{k+1}$	
	$\beta_k = \dfrac{(\mathbf{r}_{k+1}, \mathbf{s}_{k+1})}{(\mathbf{r}_k, \mathbf{s}_k)}$	
	$\mathbf{p}_{k+1} = \mathbf{s}_{K+1} + \beta_k \mathbf{p}_k$	calculate new direction vector
	$k = k + 1$	
	stop $\mathbf{x}_k$ is the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$	

The number ε is the stopping criterion.The vector $\mathbf{r}_k$ is the residual vector of the k-th iteration, the vector $\mathbf{p}_k$ is the direction vector, and **C** is the preconditioning matrix. The iterations stops if the norm of $\mathbf{r}_k$ is smaller than the norm of **b** multiplied by ε.

The purpose of preconditioning is to transform ill-conditioned system matrices to a well conditioned form, thus increasing the convergence rate of the Conjugate Gradient method. The preconditioning matrix **C** must approximate the system matrix **A** as closely as possible but still allow a relative easy calculation of the vector $\mathbf{s}_k$. A good preconditioner decreases the total execution time of the Conjugate Gradient process. This means that a good parallel preconditioner not only decreases the total number of floating point operations, but also possesses a high degree of parallelism. A good preconditioner depends both on the system matrix *and* the parallel computer. For instance, the incomplete Cholesky factorisation preconditioner is very successful on sequential computers, but performs not as good on vector- and parallel computers.

Polynomial preconditioners [8] are very well suited for parallel computers [9], and experiments have shown that, implemented on a distributed memory computer, they can be much more effective than incomplete factorisation preconditioners [10]. Therefore the concept of polynomial preconditioning is adapted and we put

$$C^{-1} = \sum_{i=0}^{m} \gamma_i A^i$$

The choice of m and γ_i is topic of active research, but is beyond the scope of the CAMAS project. Here we concentrate on parallelisation of the PCG method and the time complexity of the algorithm. We take the von Neumann series as the polynomial preconditioners.

$$C^{-1} = \sum_{i=0}^{m} N^i,$$

where **N** = **I** - **A**.

4. Parallelisation of the PCG method

4.1. Formal aspects of parallelisation

Parallelisation of a program can be achieved in two ways: by task decomposition or by data decomposition. For task composition the problem is divided into a number of subtasks that can be executed in parallel. Data composition implies dividing the data into groups (grains) and executing the work on these grains in parallel. For both decompositions parallel parts of the problem are assigned to processing elements. In general it is necessary to exchange information between the parallel parts. This can only be accomplished when the processing elements are interconnected in some way. Basically, parallelising a problem boils down to the following questions: what is the best decomposition for the problem and what is the best processor interconnection scheme.

For the PCG method task decomposition cannot be employed. The core of the algorithm consists of a big loop. Most intermediate results in a loop cycle depend on previously calculated results. This makes it impossible to execute the loop in parallel. Hence, parallelisation of the algorithm can only be realised through data decomposition.

The choice of decomposition and processor interconnection scheme depends on the kind of problem. In order to measure the quality of a specific decomposition and interconnection scheme a metric is needed. The metric that will be adopted here is the total execution time of the parallel program T_{par}. With this metric, the decomposition and interconnection scheme must be chosen in such a way that T_{par} is minimised. In the case of data decomposition T_{par} depends on the following parameters:

$$T_{par} \equiv T_{par}(p, n; \tau_{calc}, \tau_{startup}, \tau_{comm}, \text{topology}).$$

In this dependency p denotes the number of processing elements; n is a measure of the data size; τ_{calc} denotes the time to perform one floating point operation; $\tau_{startup}$ is the startup time in a communication step and τ_{comm} the time to send one byte. The topology parameter expresses the fact that total execution time depends on the selected interconnection scheme.
T_{par} can be expressed in the following terms:

$$T_{par} = \frac{T_{seq}}{p} + T_{calc.np} + T_{comm}, \tag{4.1}$$

with

$T_{seq} \equiv T_{par}(p=1, n)$;
$T_{np} \equiv T_{np}(p, n; \tau_{calc}, \text{topology})$;
$T_{comm} \equiv T_{comm}(p, n; \tau_{startup}, \tau_{comm}, \text{topology})$.

T_{seq}, the execution time of the sequential algorithm, is to be defined as the execution time of the parallel algorithm on one processor, and not as the execution time of the fastest known sequential implementation. T_{np} describes all computations that cannot be executed completely in parallel. For example load imbalance will become visible in this term. T_{comm} is the total communication time of all cycles of the parallel program. This term will include both startup times and actual communication times.

4.2 Decomposition of the PCG method

In one step of the PCG algorithm there are three vector updates ($\mathbf{x}_{k+1}$, $\mathbf{r}_{k+1}$, $\mathbf{p}_{k+1}$), three vector inner products ($(\mathbf{r}_k, \mathbf{s}_k)$, $(\mathbf{Ap}_k, \mathbf{p}_k)$ and $(\mathbf{r}_k, \mathbf{r}_k)$) and m + 1 matrix vector products ($\mathbf{Ap}_k$ and m for the polynomial preconditioning). Table 1 gives the execution times on a single processor for these operations:

Operation	T_{seq}
vector update (vu)	$T_{seq}^{vu}(n) = 2\,n\,\tau_{calc}$
vector inner product (vi)	$T_{seq}^{vi}(n) = (2\,n-1)\,\tau_{calc}$
matrix vector product (mv)	$T_{seq}^{mv}(n) = (2\,n^2 - n)\,\tau_{calc}$

Table 1: Execution times of the three basic operations

The vectors in the table are of length n, whereas the matrix is of size n x n. All numbers are real. The total execution time for one iteration step is:

$$T_{seq}(n) = \left(2(m+1)n^2 + (11-m)n - 3\right)\tau_{calc},$$

where m is the degree of the polynomial preconditioner and n the degree of the matrix. Here the times to calculate the square root (norm for $\mathbf{r}_k$) and the divisions for α_k and β_k have been left out of consideration.

Examining the PCG algorithm two observations can be made. In the first place there are three different types of data in the algorithm: scalars, vectors and matrices. In the second place, the only data types on which operations are performed are scalars and vectors. Thus, the matrices **A** and **C** remain unchanged. For this reason both matrices can be decomposed statically. The advantage of a static decomposition may be clear: now it is not necessary to send large portions of a matrix to other processing elements. Later in this chapter three different matrix decompositions will be discussed: the row-block decomposition, the column-block

decomposition and the grid decomposition.

It is also possible to decompose the vectors in the algorithm. However, unlike the matrix decomposition, the vector decomposition cannot be static. For example the vector decomposition does not always have to match the matrix decomposition. As a consequence, parts of the vector will have to be sent to other processors during execution of the parallel PCG. Another distinction from the matrix decomposition is the fact that there is only one possible decomposition for vectors: the division of the vector in equal parts.

4.2.1 The parallel vector update

For a scalar that is known to every processor the vector update operation can be performed completely in parallel. The result of the operation will be a vector that is evenly distributed over the processors. The new vector can either be used as input for a matrix vector product (in case of $\mathbf{r}_{k+1}$ and $\mathbf{p}_{k+1}$) or is further processed after the algorithm has terminated (in case of $\mathbf{x}_{k+1}$). Figure 3 shows the parallel vector update.

$$\begin{bmatrix} \mathbf{1} \\ \vdots \\ \mathbf{p} \end{bmatrix} + \textit{scalar} \times \begin{bmatrix} \mathbf{1} \\ \vdots \\ \mathbf{p} \end{bmatrix} \Rightarrow \begin{bmatrix} \mathbf{1} \\ \vdots \\ \mathbf{p} \end{bmatrix}$$

Figure 3: the parallel vector update

The computing time for this operation is

$$T_{par}^{vu}(p,n) = 2\left\lceil \frac{n}{p} \right\rceil \tau_{calc} = \frac{T_{seq}^{vu}(n)}{p} + 2\left(\left\lceil \frac{n}{p} \right\rceil - \frac{n}{p} \right) \tau_{calc},$$

with $\lceil x \rceil$ the ceiling function of x. The computing time for the parallel vector update is of the form of equation 1. When this operation is implemented as shown in figure 3 there is no need for communication between the processors. In this case $T_{comm} = 0$. The non-parallel part is a pure load balancing effect:

$$T_{np}^{vu}(p,n) = 2\left(\left\lceil \frac{n}{p} \right\rceil - \frac{n}{p} \right) \tau_{calc}.$$

If the vector length n is not a multiple of the number of processing elements p the length of the vector parts will not be the same for all processors. Now some processors will handle parts of size $\lceil n/p \rceil$ and others parts of size $\lceil n/p \rceil - 1$. In this way a load imbalance is introduced in the system the effect of which is described by the T_{np} term.

4.2.2 The parallel inner product

The parallel inner product is a two step operation. In the first step a partial inner product is calculated in every processor. The result of this step is a partial sum decomposition of scalars. The second step consists of accumulating and summing the partial sums in every processor.

The sum of the distributed partial sums will be the outcome of the parallel inner product and will be known to every processor. Notice that the second step cannot be omitted: the result of a parallel inner product (the scalars α_k or β_k) needs to be known to every processor to perform a parallel vector update. Figure 4 depicts the parallel inner product.

$$\begin{bmatrix} \mathbf{1} \\ \vdots \\ \mathbf{p} \end{bmatrix} \times \begin{bmatrix} \mathbf{1} \\ \vdots \\ \mathbf{p} \end{bmatrix} \Rightarrow \begin{bmatrix} \mathbf{1} \end{bmatrix}, \cdots, \begin{bmatrix} \mathbf{p} \end{bmatrix} ; \begin{bmatrix} \mathbf{1} \end{bmatrix} + \cdots + \begin{bmatrix} \mathbf{p} \end{bmatrix} \Rightarrow \textit{scalar result}$$

Figure 4: the parallel inner product

The total time for the parallel inner product is

$$\begin{aligned} \mathrm{T}_{par}^{vi}(\mathrm{p},\mathrm{n}) &= \left(2\left\lceil \frac{\mathrm{n}}{\mathrm{p}} \right\rceil - 1\right)\tau_{calc} + t_{sa.calc} + t_{sa} \\ &= \frac{\mathrm{T}_{seq}^{vi}(\mathrm{n})}{\mathrm{p}} + \left(t_{sa.calc} - \left(1 - \frac{1}{\mathrm{p}}\right)\tau_{calc}\right) + 2\left(\left\lceil \frac{\mathrm{n}}{\mathrm{p}} \right\rceil - \frac{\mathrm{n}}{\mathrm{p}}\right)\tau_{calc} + t_{sa}, \end{aligned}$$

where t_{sa} is the total communication time to send the partial inner products resident on each processor to all other processors. The $t_{sa.calc}$ is the computing time introduced by summing the partial inner products after (or during) the scalar accumulation. For this parallel vector inner product T_{comm} and T_{np} can be expressed as

$$\mathrm{T}_{comm}^{vi}(\mathrm{p},\mathrm{n}) = t_{sa},$$

and

$$\mathrm{T}_{np}^{vi}(\mathrm{p},\mathrm{n}) = \left(t_{sa.calc} - \left(1 - \frac{1}{\mathrm{p}}\right)\tau_{calc}\right) + 2\left(\left\lceil \frac{\mathrm{n}}{\mathrm{p}} \right\rceil - \frac{\mathrm{n}}{\mathrm{p}}\right)\tau_{calc},$$

It follows that the non-parallel part consists of two terms. The second term is, as in the parallel vector update, the load imbalance term. The first term comprises the summation of the partial results that cannot be performed completely in parallel. The exact form of this term depends on the topology of the processor network. This can also be said for T_{comm}.

4.2.3 The parallel matrix vector product

As mentioned before, there are three different matrix decompositions that will be discussed here. Each of the decompositions prescribe how the input vector and the result vector have to be decomposed to perform the parallel matrix vector product. As the vector decomposition often will not match the matrix decomposition this operation requires pre- and post-processing of the vectors involved.

<u>The row-block decomposition</u>

For row-block decomposition the matrix is divided in blocks of rows. Every block contains $\lceil n/p \rceil$ or $(\lceil n/p \rceil - 1)$ consecutive rows of the matrix. In order to perform the matrix vector product the complete input vector should be present in every processor. As this vector is always the result of a vector update or a prior matrix vector operation it is distributed over the processors. Consequently the first step of this operation consists of gathering the argument vector for every processor. Once the complete argument vector is assembled the matrix vector operation can be performed. The operation does not need any post-processing. The result of the matrix vector product will already be distributed among the processors for further calculations. Figure 5 shows the matrix vector product for a row-block decomposed matrix.

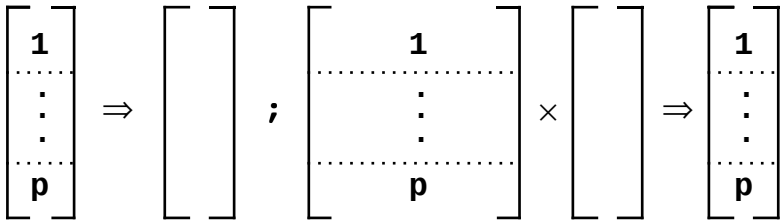

Figure 5: the parallel matrix vector product for a row-block decomposed matrix

The total execution time for the row-block decomposed matrix vector operation is

$$\begin{aligned} T_{par}^{mv;rb}(p,n) &= \left\lceil \frac{n}{p} \right\rceil (2n-1)\tau_{calc} + t_{vg} \\ &= \frac{T_{seq}^{mv}(n)}{p} + (2n-1)\left(\left\lceil \frac{n}{p} \right\rceil - \frac{n}{p} \right)\tau_{calc} + t_{vg}. \end{aligned}$$

In this expression t_{vg} is the time needed for the vector gather operation. The non-parallel time T_{np} only consists of a load imbalance term, and T_{comm} is completely determined by the vector gather operation t_{vg}.

<u>The column-block decomposition</u>

For column-block decomposition the matrix is divided into blocks of columns. Every block contains $\lceil n/p \rceil$ or $(\lceil n/p \rceil - 1)$ consecutive columns of the matrix. In contrast with the matrix vector operation for row-block matrices, there is no need for pre-processing. The distributed argument vector matches the decomposition of the matrix, so the matrix vector operation can be performed immediately. However, the result of this operation will be a partial sum decomposition that needs to be accumulated and scattered over all the other processors. As the result vector will serve as input for another matrix vector operation or a vector operation the vector must be evenly distributed among the processors. This implies that post-processing is required which includes communication and some floating point operations to evaluate the partial sums. Figure 6 shows the matrix vector product for a column-block decomposed matrix.

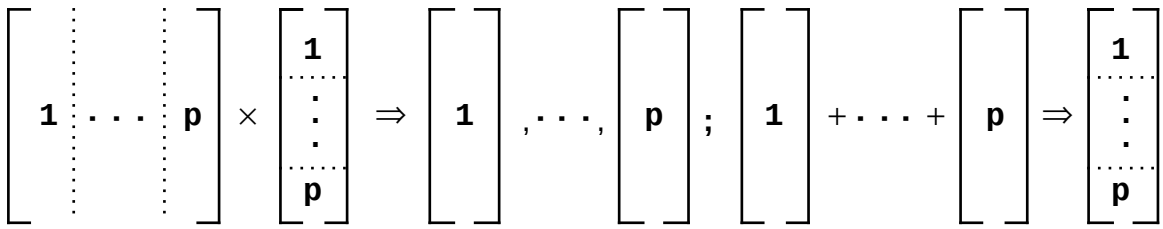

Figure 6: the parallel matrix vector product for a column-block decomposition

The total time for this operation is

$$\begin{aligned} T_{par}^{mv;cb}(p,n) &= \left(2n\left\lceil\frac{n}{p}\right\rceil - n\right)\tau_{calc} + t_{va.calc} + t_{va} \\ &= \frac{T_{seq}^{mv}(n)}{p} + \left[t_{va.calc} - n\left(1-\frac{1}{p}\right)\tau_{calc}\right] + 2n\left(\left\lceil\frac{n}{p}\right\rceil - \frac{n}{p}\right)\tau_{calc} + t_{va}, \end{aligned}$$

where t_{va} is the time to accumulate and scatter the resulting vector, and $t_{va.calc}$ the time to evaluate the partial sums. T_{np} consists of two parts: a term for the summations in the partial vector gather operation (the 2nd term) and a load imbalance term (the 3rd term). The sole contribution to T_{comm} is the communication time for the partial vector accumulate operation t_{va}.

<u>The grid decomposition</u>

Both decompositions described above are combined in the grid decomposition. Now the matrix is decomposed in p square blocks that are distributed over the processors. The argument vector in the product is divided into $\sqrt{p}$ equal parts which are assigned to $\sqrt{p}$ processing elements. To perform the matrix vector product the input vector is scattered among the processors containing a row of blocks of the matrix. The result vector is scattered among the processors in the block column direction, partially sum decomposed among the processors in block row direction. As easily can be seen both the argument vector and the result vector do not have the format that can be used for further vector or matrix vector operations. Therefore the matrix vector operation for grid decomposed matrices needs both pre-processing and post-processing that both require communication. Figure 7 illustrates the matrix vector product for grid decomposed matrices.

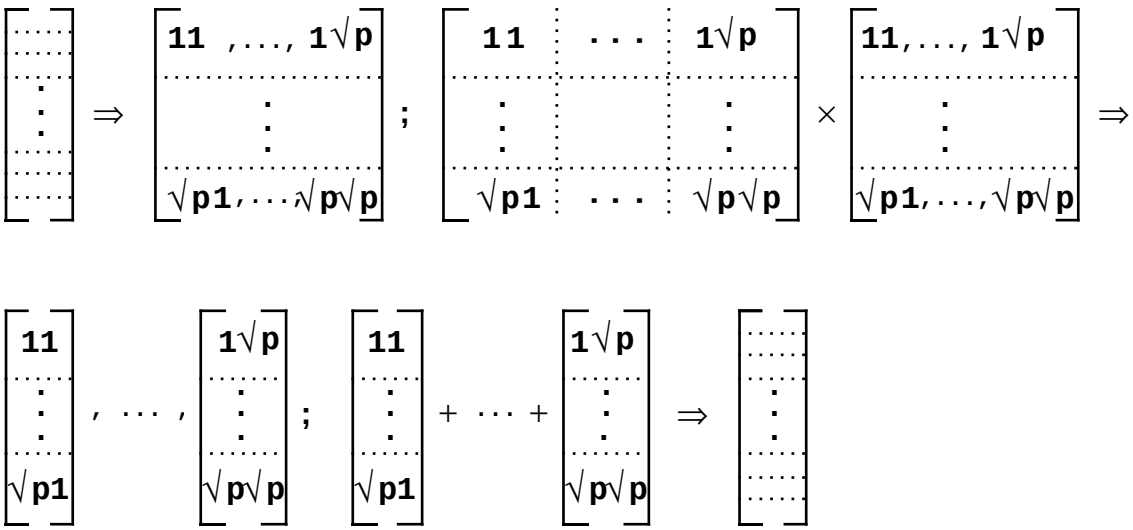

Figure 7: the parallel matrix vector product for a grid decomposed matrix

The execution time for this operation is

$$
\begin{aligned}
T_{par}^{mv;g}(p,n) &= \left\lceil \frac{n}{\sqrt{p}} \right\rceil \left(2\left\lceil \frac{n}{\sqrt{p}} \right\rceil - 1 \right) \tau_{calc} + t_{pva.calc} + t_{pvg} + t_{pva} \\
&= \frac{T_{seq}^{mv}(n)}{p} + \left[t_{pva.calc} - n\left(\left\lceil \frac{n}{\sqrt{p}} \right\rceil \frac{1}{n} - \frac{1}{p} \right) \tau_{calc} \right] \\
&\quad + 2\left(\left\lceil \frac{n}{\sqrt{p}} \right\rceil^2 - \frac{n^2}{p} \right) \tau_{calc} + t_{pvg} + t_{pva}
\end{aligned}
$$

where t_{pvg} is the time for the partial vector gather operation, t_{pva} the time for the partial vector accumulate, and $t_{pva.calc}$ the time to evaluate the partial sums after or during the vector accumulation.

Once more T_{np} consists of a load imbalance term and a term enclosing floating point operations that are not performed completely in parallel. T_{comm} is determined by the time needed in the communication of the pre- and post-processing steps, i.e. the sum of t_{pvg} and t_{pva}.

Future work

The time complexity expressions that were derived in the previous section are still not completely determined. They contain terms that still need to be refined. For example, the communication parts are not fully specified as they will depend on the processor topology. Thus, the next step is to give the time complexity of the PCG method for various processor topologies. Besides, an evaluation of the three decompositions that were derived in the previous section, should be made. This should lead to the selection of a specific decomposition.

When the time complexity of the PCG method is fully specified it will be possible to see if the results as predicted by SAD agree with what is suggested by the time complexity the PCG method.

References

[1] Th.J.R. Hughes, The Finite Element Method Linear Static and Dynamic Finite Element Analysis, Prentice-Hall, 1987.

[2] K.J. Bathe and E.L. Wilson, Numerical Methods in Finite Element Analysis, Prentice-Hall, 1976.

[3] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, A Static Performance Estimator in the Fortran D Programming System, Languages, Compilers and Run-Time Environments for Distributed Memory Machines (1992), 119-138.

[4] W.T. Carter, Finite Element Analysis on a Transputer System, Transputer Research and Applications **4** (1990), 88-96.

[5] G.H. Golub and Loan, Charles F., Matrix Computations second edition, (The John Hopkins University Press , Baltimore and London, 1989).

[6] M.R. Hestenes and E. Stiefel, Methods of Conjugate Gradients for Solving Linear Systems, Nat. Bur. Standards J. Res. **49** (1952) 409-436.

[7] S.F. Ashby, T.A. Manteuffel, and P.E. Saylor, A Taxonomy for Conjugate Gradient Methods, Siam J. Numer. Anal. **27** (1990) 1542-1568.

[8] O.G. Johnson, C.A. Micchelli and G. Paul, Polynomial Preconditioners for Conjugate Gradient Calculations, Siam J. Numer. Anal. **20** (1983) 362-376.

[9] Y. Saad, Practical Use of Polynomial Preconditionings for the Conjugate Gradient Method, Siam J. Matrix Anal. Appl. **12** (1991) 865-881.

[10] C. Tong, The preconditioned Conjugate Gradient Method on the Connection Machine, International Journal of High Speed Computing **1** (1989) 263-288.

[11] 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).