Operating System Support for Parallel Numerical Program Development

R. Fößmeier
U. Rüde
Technische Universität München
Institut für Informatik

Abstract

This paper discusses an example of numerical software development. Standard tools of the UNIX® operating system are used to design a well structured, efficient implementation of two parallel linear systems solvers. It is shown that standard UNIX features are suitable for describing and implementing parallelism. The resulting codes are fully portable. Other software engineering and efficiency issues are discussed in detail.

Zusammenfassung


Resumo

Este trabalho apresenta um exemplo de evolução de cálculo de programação. Os ferramentas padrão do sistema UNIX são usadas para o desenvolvimento de solvers lineares paralelos de forma eficaz. Mostra-se que as características padrão de UNIX são adequadas para o descrever e implementar paralelismo. Os códigos resultantes são totalmente portáveis. Outras questões de engenharia de software e eficiência são discutidas em detalhe.

1. Introduction
2. Direct linear systems solvers
   2.1. Gauss/Cholesky elimination
   2.2. Orthogonal elimination (Givens)
   2.3. Parallelization
   2.4. Vectorization
   2.5. Comparison
3. Development of software for microcomputers with UNIX
   3.1. Fortran
   3.2. Software development with the UNIX system
   3.3. Pipelines and processes (filters)
   3.4. The shell language
   3.5. Parallelism using the UNIX system
4. Implementation of parallelism with processes and pipes
   4.1. The algorithm
   4.2. Formulation using processes and pipes
   4.3. Backward substitution
   4.4. Efficiency considerations
   4.5. Revised version
5. Intra-process optimizations
   5.1. Arrays and loops
   5.2. Processor dependent optimizations
   5.3. Process sizes
6. Parallel execution examples
7. Summary and Conclusions

References

UNIX is a trademark of Bell Laboratories.
1. Introduction.

In advanced projects numerical software is becoming more and more complex. A tremendous additional complexity is introduced by parallelism. In the past too little attention has been paid to controlling the complexity of numerical software.

Furthermore, it seems that a remarkable number of numerical methods remains unpopular because they are difficult to implement. In many cases this is not due to the algorithmic complexity in itself but to the unsuitability of the available program development tools.

Dynamic storage management is a case in point. Many advanced algorithms need dynamic memory allocation. If the programming language (e.g., Fortran) does not support dynamic storage allocation, the programmer has to implement it by himself. This is a great waste of time and also loss of program efficiency as the self-made simulation is usually less efficient than carefully designed library functions. All modern operating systems offer efficient dynamic storage handling. However these mechanisms cannot be accessed from languages that are based on pure static storage management. In this example a simple problem becomes difficult only because suitable tools cannot be used.

Additionally numerical software has special needs. Many of the more modern languages, like Pascal or Modula, do not support important data structures like double precision, complex numbers, matrices and vectors in a convenient form.

Does this now suggest that we need a new numerical superlanguage, combining the good parts of Fortran, Pascal and several other languages? We do not believe so. Maybe the strongest argument against another new language is a look on the junkyard of computer science: It is full of 'superlanguages' that turned out to be flops. Only a few languages are used in practice and even fewer are relevant for numerical programming.

We suggest a completely different approach. The main idea of this paper is to present the idea of heterogeneous hierarchical implementation of numerical software. As no language is well suited for all tasks we do not use one single language for the whole program. Different parts (refinement levels) of the program are implemented with different tools, i.e. different languages.

The basis for our investigation is the UNIX programming environment and the example for our presentation is the parallel solution of (full) linear systems of equations.

The highest level of implementation is the description of coarse grain parallelism and interaction of major modules, like elimination and substitution. For this top level we suggest to use the UNIX shell language, which has parallel constructs. However, the (Bourne) shell language is completely unsuitable for any numerical algorithm, so that another language is necessary. From the shell level, programs in classical languages can be called.

The next level are small modules written in a standard programming language. In our example these are the elimination (to bring a matrix into upper right triangular form) and the substitution to calculate the solution of the equations. A Cholesky
method is implemented for positive definite matrices and Givens rotations can be used for general systems. The substitution is the same for both cases and only one program is necessary. This already demonstrates the savings in development time: old codes can be easily reused.

On this level we use the programming language C which is available on all UNIX systems and is (with some restrictions) suited quite well for numerical programming (compare [26]). An extended C language is even used for the programming of vector supercomputers [12, 13]. However the choice of C as language is not important. We might have as well used Fortran or any other language. Furthermore different modules may be written in different languages.

This is very important because it makes all the old codes available without forcing the new developments to use the same language. It also makes it possible to interface to other (special purpose) languages, e.g. for the description and programming of advanced user interfaces or even artificial intelligence components. Unfortunately our example is too small to show the real usefulness of these possibilities.

On all implementation levels mentioned so far, operating systems, standard tools and compilers screen the user very effectively from machine dependencies. Yet, numerical software may have extreme efficiency demands, and it is often difficult or impossible to make use of the full performance of a machine and its facilities in a machine-independent way. Especially with decreasing hardware costs, the quality of compilers and optimizers cannot always keep up with processor performance, and so one may be forced (or at least tempted) to use assembly language. This may be especially true if one uses special (e.g. parallel) hardware that cannot be used optimally from a standard programming language. In some applications it may even be suitable to download special microcodes into the hardware, see e.g. the discussion in [1].

It is important to note that this influences only the lowest software level; moreover, efficiency is generally determined by comparatively small parts of the whole program, typically the innermost loops of an algorithm; so the work necessary to squeeze a reasonable performance from a machine is limited to a few spots. In our examples an overall gain of 20 to 30 per cent (on our particular computer) could be achieved by reprogramming the innermost loops in assembly language. This is even more true for parallel computers. On the lowest level fine grain parallelism may be exploited. Here the savings of carefully designing the inner loops for a special (e.g. pipelined) architecture may be even more important. A good optimizing compiler may be expected to automatically detect parallelism. Still it may be worthwhile doing some hand-optimization.

A very important aspect of software is portability, that is the ease (or difficulty) to transport it to other computers. Portability is one of the main reasons for using Fortran: it is available on almost all computers. To combine portability and efficiency on the lowest level of programming, standard library routines have been developed to perform frequently needed operations, e.g. the Basic Linear Algebraic Subprograms (BLAS [11]).

In contrast our approach depends on the UNIX system. Presently UNIX is available on many machines and the tendency is increasing. From our perspective using UNIX provides sufficient portability by itself. Furthermore it allows to treat necessity machine dependent parts, like our assembler routines in a reason way. All these parts are treated using conditional compilation so that on other machines the regular, portable, C version of the routines is compiled, and on our machine the equivalent assembler routine. Thus the same source code should compile on any UNIX system.

In the main body of the paper we first describe the (parallel) linear system solvers and discuss different models of parallelism. Then we describe the portion of the UNIX system which is essential for understanding our implementation. Based on that we go to implementation details and show finally how various aspects of our approach influence the efficiency of the resulting programs.

2. Direct Linear System Solvers
In this paper we will focus on direct linear system solvers. The basic idea of this class of algorithms is the decomposition of the system matrix by

\[ A = E \cdot R, \]

where the matrix \( R \) is an upper (right) triangular matrix. The algorithms differ in the properties and construction of \( E \). For details see e.g. [2, 25, 27, 28].

We shall use the following notation for the description of loops over all integer numbers between \( \mu \) and \( v \):

| for \( i \) from \( \mu \) to \( v \) do | The steps of the loop shall be executed in numerical order |
|forall \( i \) from \( \mu \) to \( v \) do | The steps of the loop are totally independent (and thus may be executed in parallel) |

A typical example for the second type is the linear combination of two vectors. The scalar product of two vectors is an instance of the first type; it cannot be fully parallelized, although a partial (hierarchical, "type 1", compare [17]) parallelization is possible. Between the two cases there is the case of mutually interfering steps which may be executed in any order, but not in parallel. Since compilation can hardly take advantage of this, no special notation seems to have evolved. We propose

| for \( i \in [\mu \ldots v] \) do | The steps of the loop may be executed in any order but not in parallel |
For the inner loops of the algorithms we shall also give a formulation by subroutines of the BLAS package \[11\]. We make use of the Fortran convention of passing an array argument by its first element. For generality's sake, we use the basic names of the routines only; for simplicity's sake, we omit vector increments (which would depend on the way the matrix is stored).

In the whole chapter we assume the matrix \(A\) of the linear system to be of dimension \(n \in \mathbb{N}\).

2.1. Gauss/Cholesky elimination

The most basic algorithm for matrix decomposition is Gaussian elimination. Here \(E\) is a lower (left) triangular matrix. The matrix is constructed by stepwise elimination of the corresponding elements in \(A\). It is well known that for general systems pivoting is necessary. Usually the elimination proceeds column-wise from the left. Before elimination in a new column is started the new pivot has to be determined, typically by determining the maximum element in the particular column. This prohibits the parallelization of the standard Gauss algorithm. For a positive definite matrix \(A\), however, it is well known that \(E\) can always be computed without pivot searching, i.e. without permuting rows or columns of \(A\):

<table>
<thead>
<tr>
<th>for i from 1 to n do begin</th>
<th>for i from 1 to n do begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>piv := a_{i,i}</td>
<td>piv := a_{i,i}</td>
</tr>
<tr>
<td>forall j from i+1 to n do</td>
<td>forall j from i+1 to n do</td>
</tr>
<tr>
<td>for all k from i+1 to n do</td>
<td>for all k from i+1 to n do</td>
</tr>
<tr>
<td>a_{k,k} := a_{k,k}</td>
<td>a_{k,k} := a_{k,k}</td>
</tr>
<tr>
<td>((\ast) := := denotes decrementation (\ast))</td>
<td>((\ast) := := denotes decrementation (\ast))</td>
</tr>
</tbody>
</table>

**Gaussian elimination**

without pivot search

<table>
<thead>
<tr>
<th>BLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXPY (n-i, (\ast), a_{i,j+1}, a_{i,j+i})</td>
</tr>
<tr>
<td>DOT (i-1, a_{i,i}, a_{i,i})</td>
</tr>
</tbody>
</table>

Compact Cholesky decomposition

2.2. Orthogonal elimination (Givens)

For general, i.e. not positive definite, matrices, parallel algorithms are usually based on orthogonal elimination (Givens rotation). This is a different elimination technique, where two rows of the matrix are rotated to render one element zero. The basic structure of the algorithm is:

<table>
<thead>
<tr>
<th>for i from 1 to n do begin</th>
<th>for i from 1 to n do begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>for j in ([i+1..n]) do</td>
<td>for j in ([i+1..n]) do</td>
</tr>
<tr>
<td>rotate ({(a_1, a_2, \ldots, a_n)})</td>
<td>rotate ({(a_1, a_2, \ldots, a_n)})</td>
</tr>
<tr>
<td>ROTG (a_{ij}, a_{ij}, c, s)</td>
<td>ROTG (a_{ij}, a_{ij}, c, s)</td>
</tr>
<tr>
<td>((\ast) to make (a_{ij}) zero (\ast))</td>
<td>((\ast) to make (a_{ij}) zero (\ast))</td>
</tr>
</tbody>
</table>

Orthogonal elimination by Givens rotations

Cholesky decomposition normally is performed as direct or compact decomposition (see \([2,25,28]\)) which differs from "non-compact" Gaussian decomposition by the order of operations. Its advantage is the possibility to accumulate the scalar products in extended precision.

The result \(L\) of the decomposition can be stored either in the upper right or in the lower left half of the matrix. If we keep the right hand side(s) of the linear system in additional columns to the right of the matrix \(A\) it is easiest to store \(L\) in the upper right half of \(A\). In this case the innermost loop of the algorithm is the scalar product of two columns of \(A\):

\[
\begin{align*}
&\text{for } i \text{ from 1 to } n \\
&\text{begin} \\
&\quad x := a_{i,i} \quad \text{begin} \\
&\quad a_{i,i} := \sqrt{x} \\
&\quad \text{for } j \text{ in } [i+1..n] \text{ do} \\
&\quad \text{begin} \\
&\quad \quad x := a_{ij} - [a_{i,j}] \quad \text{begin} \\
&\quad \quad\quad [a_{i,j}] \\
&\quad \quad a_{ij} := x/a_{ij} \\
&\quad \text{end} \\
&\quad \text{end} \\
&\quad \text{end} \\
&\end{align*}
\]
2.3. Parallelisation

If no permutations are necessary for the decomposition, several elimination steps may be executed in parallel (compare Sameh and Kuck [22]). Assume e.g. a sequential elimination starting in the first column working from bottom to the top. Elimination in the second column may start as soon as the elimination in the first column has proceeded far enough (that is two steps). Similarly the elimination in the third column may be started as soon as the elimination in the first column has proceeded four steps.

\[
\begin{array}{ccccccccccccccccc}
& x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\
9 & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\
8 & 10 & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\
7 & 9 & 11 & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\
6 & 8 & 10 & 12 & x & x & x & x & x & x & x & x & x & x & x & x & x \\
5 & 7 & 9 & 11 & 13 & x & x & x & x & x & x & x & x & x & x & x & x \\
4 & 6 & 8 & 10 & 12 & 14 & x & x & x & x & x & x & x & x & x & x & x \\
3 & 5 & 7 & 9 & 11 & 13 & 15 & x & x & x & x & x & x & x & x & x & x \\
2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & x & x & x & x & x & x & x & x & x \\
1 & 3 & 5 & 7 & 9 & 11 & 13 & 15 & 17 & x & x & x & x & x & x & x & x & x
\end{array}
\]

Fig. 1-a

Extending this idea we arrive at a parallel elimination algorithm; [22] gives fig. 1-a to show which steps can be performed simultaneously, each time rotating two adjacent rows. Since our method requires to store and pass the matrix rows in the order of elimination, we used the scheme of fig. 1-b which to some may seem more familiar and allowed us to store the matrix in traditional order, i.e. uppermost row first. We used a single pivot line (marked by ▶) for a whole column.

\[
\begin{array}{cccccccccccccccc}
\text{▶} & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\
1 & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\
2 & 3 & x & x & x & x & x & x & x & x & x & x & x & x & x & x & x \\
3 & 4 & 5 & ▶ & x & x & x & x & x & x & x & x & x & x & x & x & x \\
4 & 5 & 6 & 7 & ▶ & x & x & x & x & x & x & x & x & x & x & x & x \\
5 & 6 & 7 & 8 & 9 & ▶ & x & x & x & x & x & x & x & x & x & x & x \\
6 & 7 & 8 & 9 & 10 & ▶ & x & x & x & x & x & x & x & x & x & x & x \\
7 & 8 & 9 & 10 & 11 & ▶ & x & x & x & x & x & x & x & x & x & x & x \\
8 & 9 & 10 & 11 & 12 & ▶ & x & x & x & x & x & x & x & x & x & x & x \\
9 & 10 & 11 & 12 & 13 & 14 & ▶ & x & x & x & x & x & x & x & x & x & x
\end{array}
\]

Fig. 1-b

Thus the complete decomposition needs \(2n-3\) steps of at most \(n/2\) eliminations performed in parallel. This basic idea can be used on all elimination schemes without pivoting. So Cholesky decomposition as well as orthogonal elimination can be parallelized on the level of the second loop.

A special advantage of this method is the fact that all data is processed once for all. So data access control is trivial: When a process has finished on a datum (a matrix row) any other process is free to read it since it will never be changed again. We shall exploit this fact to use very simple techniques (producer–consumer synchronization) for process communication (see section 2). This is not possible for schemes with more parallelism (e.g. in [15]) which are, however, not strikingly superior to the standard scheme above.

2.4. Vectorization

Some algorithms may be parallelized further on a lower level by vectorizing their elementary steps (innermost loops). Here we find a crucial difference between orthogonal as well as non-compact elimination and compact Cholesky decomposition: Whereas the first group of algorithms compute the matrix elements as linear combinations of vectors and thus are suitable for type-0-parallelism, the steps of the compact algorithm, when sequentialized as above, are scalar products whose vectorization is not straightforward (type-1-parallelism) and not as efficient as type-0-parallelism. A type-0 parallel variant of compact Cholesky is:

\[
\text{for } i \text{ from } 1 \text{ to } n \text{ do} \\
\text{begin} \\
\text{for all } j \text{ from } i \text{ to } n \text{ do} \\
\quad a_{ij} := \left[ \begin{array}{c} a_{1i} \\ \vdots \\ a_{i-1,j} \end{array} \right] \\
\quad a_{ij} := \sqrt{a_{jj}} \\
\text{end} \\
\text{for all } j \text{ from } i+1 \text{ to } n \text{ do} \\
\quad a_{ij} := a_{ij}/a_{jj} \\
\text{end}
\]

**type-0 parallelizable variant of Cholesky algorithm**

While a scalar product cannot be accumulated fully in parallel, it is possible to compute all scalar products in the \(j\)-loop independently, making this loop the innermost one. Note that in this version of the algorithm the inner loop runs along matrix rows rather than columns. This might influence the storing of the matrix.
Here a complete matrix row is eliminated simultaneously, which yields even more parallelism than the parallelization above (n-1 versus 2n-3 steps). However, while the orthogonal elimination algorithm can be vectorized on the lowest level and parallelized as described on a higher one independently, here only one loop, the i-loop, can be parallelized. (If the i-loop is parallelized, all processes need simultaneous reading access to the matrix.) So on a machine which allows vectorized linear combinations and real parallel processes, non-compact Gaussian or orthogonal elimination is, with regard to execution speed, still preferable to compact decomposition.

If Cholesky decomposition is vectorized as above, the innermost loop (j) again runs across the rows of A, which supports row-wise storing of the matrix. If the machine requires array operands to be contiguous, row-wise storing is even compulsory.

The advantage of Cholesky decomposition versus Gaussian elimination is the smaller numerical effort of n^2/6 versus n^3/3 multiplications. The symmetry of the matrix can, however, be exploited also in non-compact Gaussian elimination (see [2]), a fact often neglected because of the numeric superiority of the compact Cholesky method with extended precision scalar products [25, 28]. We denote the elements of the lower left triangular matrix E by e_{ij} and the elements of the upper right triangular matrix R = D\cdot E^T by r_{ij} (of course, the ej need not be stored, as they are easily computed from R):

```
for i from 1 to n do
    begin
        d_i := r_i
        forall j from i+1 to n do
            begin
                t_j := r_j/d_i
                forall k from j to n do
                    t_k := t_k - t_j \cdot r_{kj}
                end
            end
    end
```

Symmetric non-compact Gaussian decomposition

This version of the algorithm also needs only n^2/6 multiplications and can be parallelized on the j/\bar{i}-level and vectorized on the k-level independently.

If the lowest program level, typically consisting of the innermost loops, uses standard routines like BLAS [11], vectorization can be left to the implementors of these routines, thus completely screening the user from machine idiosyncrasies.

2.5. Comparison

With regard to applicability, parallelization, vectorization and computational effort the algorithms considered compare as follows:

<table>
<thead>
<tr>
<th>method of elimination</th>
<th>conditions for system matrix</th>
<th>parallel elimination as described</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss with pivoting</td>
<td>(non-singular)</td>
<td>1 \cdot 1</td>
</tr>
<tr>
<td>orthogonal (Givens rotations)</td>
<td>(non-singular)</td>
<td>1 \cdot 1</td>
</tr>
<tr>
<td>Gauss (without pivoting)</td>
<td>no pivot search necessary</td>
<td>1 \cdot 1</td>
</tr>
<tr>
<td>compact Cholesky</td>
<td>positive definite</td>
<td>either, not both</td>
</tr>
<tr>
<td>symmetric Gauss</td>
<td>positive definite</td>
<td>1/6</td>
</tr>
</tbody>
</table>

3. Development of software for microcomputers with UNIX

In this section we discuss program development aspects for numerical algorithms, in particular those arising from parallelism.

3.1. Fortran

The classical tool for programming numerical algorithms, the Fortran language, is limited in many ways. It lacks data structuring features and control flow statements making development of good programs difficult. Worst, maybe, is the lack of dynamic storage handling and parallelism. Fortran reflects in its basic concepts the computer architecture and programming style of 1955.

On the other hand Fortran is still widely used with good reasons. It is the only major language supporting the basic data structures for numerical programming well (real, double, arrays). Most numerical standard software is written in Fortran. It is available on almost all machines. On the vector computers it makes this special type of parallelism available (compare [23]), though in an inconvenient form: One either has to use special subroutine calls or code the programs in a special (sequential!) style which can be vectorized by the compiler.

3.2. Software development with the UNIX system

Modern software development environments as the UNIX system provide much more than only compilers. There is a rich set of development tools. Within the UNIX system software can be modularized using separate processes (filters). Basically a filter is a function to be performed on a stream of data.
5.3. Pipelines and Processes (filters).

UNIX is a multiprocess system organizing a (pseudo-)concurrency of many separate processes. For the practical usefulness of this concept an efficient interprocess communication is essential. For this task, UNIX offers pipes.

A pipe is a conduit that enables a one way data stream between two processes. For the user pipes are virtual files. They operate on a strict first in – first out basis. While one process writes data into the pipe, another process can read the data out of the pipe. Pipes are functionally equivalent to real files, but are implemented as system buffers (usually something like 4 KByte large) in RAM storage (random access memory). No physical intermediate files are used. Thus large amounts of data can be passed efficiently between two processes. The overhead is not a write and successive read on a background storage system but just a transport of data in RAM, and some system time for switching between the processes. The system kernel is responsible for synchronizing the processes. It will stop the writing process and switch to the reading process when the pipe buffer becomes full, and switch back to the writing process when the buffer gets empty. Pipes can be used to connect several processes by several channels. It is the responsibility of the user to avoid deadlocks, however.

For the programmer, pipes are convenient, because they can be handled just like ordinary files, with only few restrictions. The same program can read its input from a file or get it directly from another process via a pipe.

5.4. The Shell Language

A further important UNIX tool, without which our project would have been impossible, is the command interpreter shell. The shell is programmable and supports many of the constructs of higher programming languages. It allows multi-process programming and makes pipes available as language constructs.

Usually all UNIX processes are associated with two files, stdin, the standard input, and stdout, the standard output. The first one represents the keyboard, the other the screen. The shell allows redirection of input and output. The command sequence

```
program_1 < a > b
```

will start program_1 with its input coming from file a, instead of from the keyboard, and its output going to file b, instead of to the screen.

In the shell language pipes are symbolized by the vertical bar |. The command

```
program_1 < a | program_2 > b
```

is functionally equivalent to the two commands

```
program_1 < a > intermediate_file
program_2 < intermediate_file > b.
```

The first version is more efficient, as it saves disk-I/O. The two processes program_1 and program_2 are run (pseudo-) parallel instead of sequentially.

Pipes are commonly used to structure complex tasks, by combining several steps of processing. A typical example, found in introductory books to the UNIX system, is

```
is | wc
```

This sequence solves the problem of counting the files in the current directory by using is, which lists all directory entries, and wc, which counts the number of words and lines of its input.

This idea of combining small and simple programs is part of the 'UNIX philosophy'. Instead of writing one complex program, the user is encouraged to solve a problem using several small ones and connect them via pipes. Programs used in such a way are called filters.

Ideally, the UNIX system itself should consist of a comparatively small number of general, but simple commands. Of course, as any UNIX user can see by just looking at the thickness of the manuals, this concept is not being followed purely. For a detailed discussion see e.g. [19].

5.5. Parallelism using the UNIX system.

A basic producer and consumer (pseudo-) parallelism is already expressed by building a pipeline. A different type of asynchronous, parallel execution can be started using the 'k' symbol. The program sequence

```
task1 & task2 &
```

means that the two processes (task1 and task2) are to be executed in parallel. Processes can be synchronized with the wait command and communication between processes can be performed with (most conveniently named-) pipes. The advanced UNIX V system offers still more concepts of parallel computing, like shared memory, messages and semaphores. However, (named) pipes and processes are sufficient to describe computing on a MIMD (multiple instruction, multiple data) computer with local (i.e. nonshared) memory, which is a very promising class of machines for large numerical problems. Typical machines of this class are e.g. the hypercube architectures [4, 3], the Supremnum [9, 14], the Navier–Stokes Computer [16] or also the Cray X–MP.

Alternative architectures use shared memory. Typical representatives are the Denelcor HEP [8], the Cedar supercomputer [10], the NYU ultracomputer [6] and the RPS [18]. This class of machines is already quite advanced and smaller computers with up to 16 processors (e.g. Sequent) are already commercially available. They offer more communication possibilities than the local memory computers. Thus a tighter coupling of parallel processes is possible. On the other hand the additional capabilities may be very difficult to program. If data is shared between (many) processes extreme care must be applied every time one process wants to modify the data. Several processors may want to write at the same time, copies of the data may be in cache storage that can become inconsistent, etc. If the possibilities are not either severely restricted or supervised by suitable tools, they seem to be almost too difficult to use in larger applications. A further disadvantage of these machines is the
complexity of the hardware. If there are \( n \) processors and \( m \) banks of memory then
\( n \times m \) different connections must be possible. The complexity of the switching network
basically grows with the square of the number of processors. For machines with many
processors a multi-stage switching network must be used, but then the memory access
time grows. In view of this discussion we restrict ourselves to the local memory
model. Of course our algorithms could be implemented on a shared memory
machine.

Though still most UNIX installations are single processor machines, the operating
system can be used for expressing parallelism in the execution. The expressive power
of the shell language (for parallel execution) is quite comparable to true parallel
programming languages like Occam. On the few true UNIX multiprocessor systems
using these features leads to real parallelism in the execution. As an alternative a
local network of single processor UNIX computers may be used for distributed parallel
processing.

Assume \( m_1 \) and \( m_2 \) are the names of two connected computers. Processes may
be started remote using the remote shell facility. With e.g.
\[
\text{rsh m}_2 \text{ prog ...}
\]
the program prog may be started on machine \( m_2 \). For this it is necessary that the
program prog is present on machine \( m_2 \). A remote shell command may get input via
regular pipes. Parallel execution can e.g. be programmed by installing programs \( p_1 \),
\( p_2 \) on machines \( m_1 \) and \( m_2 \). If \( m_1 \) is the local computer the command
\[
p_1 | \text{rsh m}_2 p_2
\]
executes the programs pseudo-parallel but
\[
p_1 | p_2
\]
starts \( p_2 \) on the remote computer and connects its input to the output of \( p_1 \) which
still runs on the local computer. Thus true asynchronous execution is achieved. The
two processes are still synchronized by input/output buffers.

In typical fast local area networks the overhead for starting such a connection is
comparatively high (in the order of several seconds). With the connection once
established the rate of data transferred is quite good compared to the processing speed.

There are of course more efficient ways for installing distributed systems using
lower level interfaces of the network software. On the other hand there are a number of
applications where it seems sufficient to use the slow remote shell mechanism for
parallelization. For an example see section 6.

The basic suitability of the UNIX system parallel computation is being used in
other projects, too. Multiprocessor UNIX systems are becoming commercially
available. The NYU Ultracomputer at the Courant Institute e.g. will even be a UNIX-
based parallel computer containing thousands of processors, see [6]. On this machine
it is planned that parallel languages are mapped to (extended) UNIX system calls.
For example a parallel loop
\[
\text{forall } i \text{ from } 1 \text{ to } v \text{ do ...}
\]
could be compiled to a spawn system call. spawn is a generalization of fork that
simultaneously creates \( v \) processes. It is semantically equivalent to a correspondingly
iterated fork. For this project the UNIX system is used as low level basis for parallel
computing. We are using UNIX as a high level language.

4. Implementation of parallel linear system solvers with processes and pipes.

This section introduces our implementation of parallel system solvers on a network of
microcomputers. We start with a parallel formulation of the solver using processes
d and pipes.

4.1. The algorithm

As was seen in section 2 two types of parallelism occur: fine grain parallelism that can
be exploited e.g. by vectorization and coarse grain parallelism that can be exploited by
processes. We will now focus on the second type parallelism. (Each process may possibly
exploit the fine grain parallelism using vectorization techniques, compare 2.5.)

Assume a solution scheme in which the elimination proceeds through the columns
from the left to the right. The obvious parallelization idea for all these schemes is
that the elimination of the second column need not wait for the first to finish, (assum-
ing no pivoting is necessary). As soon as the elimination in the first column has pro-
ceeded two rows the elimination in the second column may start. Similarly the elimi-
nation in the third column may start as soon as two rows have been completed in the
second elimination sweep. This continues to the last column.

So it is obvious that one may attribute the elimination of one column to one pro-
cess. Its input is a full matrix and its output is a reduced matrix with one column
less. Input and output are by rows. If such a process is programmed to output the
rows of the result as soon as they are computed, the second process can work in parallel
with the first.

As an additional advantage a single process need not store the whole matrix but
only two of its rows at any time. Note that each process gets a \( n \times n \) matrix as input
and produces an \( (n-1) \times (n-1) \) matrix as output. An additional output is one row of
the right triangular matrix \( R \).

For programming convenience the system’s right hand side(s) are added to the
matrix as extra columns, so that the matrices are rectangular, not square.
4.2. Formulation using processes and pipes

Assume now that elim0 is the name of a UNIX process taking a matrix as its standard input, produces the reduced matrix as its standard output and generates its row of R in a file to be specified as a parameter. Elimination of one row can then be invoked by

```
cfull_gls elim0 r1 > reduced_gls
```

and two elimination steps by

```
cfull_gls elim0 r1 | elim0 r2 > reduced_gls
```

Thus the elimination in n columns could be performed by a pipeline of n processes. The resulting matrix R would be contained row by row in n different files r1, r2 ... rn.

Note that this idea essentially implements the parallel scheme of 2.3. However there the elimination in different columns has been explicitly synchronized. Our implementation leaves the synchronization to the system. If the rows of the matrix are short, that is the matrix is small, then the complete matrix may fit in the pipe-buffer, and the processes are in reality executed sequentially. This of course makes sense. There is no need to parallelize, if the tasks are small. The loose synchronization is an advantage in practice.

4.3. Back substitution

Let us turn to the second step of the solution of the linear system. A process performing the backward substitution is designed. Assume back0 is a filter that, given a part of the solution and a row of the triangular matrix extends the solution by one more unknown. It performs just one step of the substitution process. The full substitution again consists of a pipeline of n back0 processes, which successively build up the solution.

Thus the solution of a linear system with n unknowns is performed by the following UNIX (shell) program

```
cfull_sys elim0 r1 | elim0 r2 | ... | elim0 rn
ccmpy back rn | back rn-1 | ... | back r1 > solution
```

In the substitution process there is some parallelism, too. As soon as one unknown has been determined it can be handed unchanged through all back0 processes. Each back0 process must read the part of the solution it gets and then write it to the next process, before it starts its calculation. Thus (conceptually) all substitution operations based on the mth unknown may be performed in parallel. However, the substitution is only a $O(n^2)$ process as compared to $O(n^2)$ of the elimination. Spending much effort on its parallelization is not worth the effort as long as the elimination dominates the work so much. The first elimination process is computationally more expensive than all back0 processes together.

4.4. Efficiency considerations

The processes in UNIX have all their private storage, and communication is done by copying data from one process to another with the pipe mechanism. In our formulation with n processes the matrix with size $O(n^2)$ must be copied n times, giving a total of $O(n^2)$ numbers to be copied. So the copy operations are of the same basic complexity as the arithmetic operations. This work cannot be neglected.

Furthermore many of the processes will do comparatively little work. The organization of parallelism must be expected to be expensive in any case and definitely is expensive in an UNIX environment. The creation and organization of a process does only pay, if this process does a substantial amount of work.

Our program with n processes for the elimination internally assumes that there are $O(n)$ processors. However, one cannot expect having a machine with as many processors as any linear system, one wishes to solve, has unknowns.

4.5. Revised version.

For a practical implementation it is necessary to modify the algorithm. Instead of the elim0 process a more general program elim is used. This program is similar to elim, except that it cannot only eliminate one column, but several in a single process. How many columns are to be eliminated can be specified as a parameter. So as the extreme cases elim can perform elimination of only one column just as elim0, or it can perform a complete elimination of the whole matrix. Usually it will be applied to perform something in between. Each elim process generates not only one row of the triangular matrix R, but a whole block of it.

Similarly the back0 process is extended so that it not only accepts single rows of R but blocks in the right form. The modified process back does not need a parameter specifying the size of the block as this information is already contained in the other data.

For an example for the application of the two modules elim and back see section 6. The following section now discusses how the processes are to be programmed efficiently.
5. Intra-process optimizations

Many of our tests were carried out on a microcomputer with INTEL 80286/87 processors (compare [20,17]), under a UNIX V.2 operating system. Important properties of this configuration were:

a) Two types of processes, called the small and large model respectively, differing by the length of addresses and, therefore, the available address space.

b) A fast numeric co-processor, which is, however, supported by the C-compiler only with respect to basic (“core”) computational instructions and to use of the hardware stack for the evaluation of expressions.

We tried to find out to which degree these features could be exploited to get some extra efficiency. To that purpose, we tried

a) structuring algorithms as well as data to get small parts which are easy to overlook and fast to access and

b) programming critical spots in assembly language, making heavier use of registers.

The results are, of course, hardware-dependent, and representative at most for microcomputers. They depend especially on the ratio between memory access time and instruction cycle. Interestingly, addressing problems quite similar to those encountered here are found also with supercomputers, where storage access times are quite critical and, especially for vector computers, operands have to be arranged carefully within storage pages or cache memories (compare [25]).

5.1. Arrays and loops

The INTEL 80286 processor offers faster addressing if all data are stored in one segment of 64 KB, in which case the segment register need not be altered during the run of the program. This advantage partly still holds if at least each single array is contained in one segment; in this case only short (16 bit) index arithmetic is needed if in a loop each array has a segment register of its own.

In algorithms where a matrix is accessed row by row it is even sufficient to keep only complete rows (or columns) together in a segment. So even with very long matrix rows, the orthogonal elimination algorithm, where only two matrix rows are accessed simultaneously, can still make use of efficient segment addressing: Each of the (possibly few per process) rows is attributed a segment of its own, the matrix becoming really an array of pointers. This kind of storage organization would even still be suited for parallel execution of a rotation step on an array processor.

We programmed the scalar products of the Cholesky algorithm to run along columns in the upper right half of the matrix; this suggested storing the matrix by columns. On the other hand, the parallelization scheme requires the matrix to be passed row by row between the processes, and since the UNIX low-level I/O routines read and write operate on contiguous pieces of memory this was a strong argument to store rows contiguously, since that's the way they are input/output. However, the additional effort for I/O with scattered or split rows is only of order O(n^2), n being the matrix dimension, whereas any savings in the computation of the inner loop are multiplied by n^2; so we tried both alternatives. The results of the tests are listed in the next paragraph.

5.2. Processor-dependent optimizations

Since our C compiler leaves most addressing optimization to the user (unlike many higher level language compilers, especially on mainframes) we tried several ways of squeezing more efficiency out of the algorithms. An obvious approach was the usage of the eight arithmetic stack registers of the numeric co-processor 80287 for storing intermediate results, to minimize memory access.

The innermost loop of the orthogonal elimination algorithm runs over the instruction sequence

\[
\begin{bmatrix}
\mathbf{s}_i \\
\mathbf{b}_i
\end{bmatrix} :=
\begin{bmatrix}
\mathbf{c}_i \mathbf{a}_i + \mathbf{s} \cdot \mathbf{b}_i \\
\mathbf{c}_i \mathbf{b}_i - \mathbf{s} \cdot \mathbf{a}_i
\end{bmatrix}
\]

Taking into account that on a single processor the two assignments cannot be done simultaneously and therefore require a temporary variable, we find 12 memory accesses and 6 arithmetic operations, neglecting index arithmetic. By using the 80287 registers memory access can be reduced to the absolute minimum of four (fetching and storing of \( \mathbf{a}_i \) and \( \mathbf{b}_i \)), \( \mathbf{c} \) and \( \mathbf{s} \) being kept in registers during the whole loop. This gave us a gain in speed of only about 20%; yet in larger machines with a better ratio of processor and memory speed the gain might be substantial.

There was hardly anything to be optimized with index arithmetic in this case, since the operation runs along the vectors \( \mathbf{a} \) and \( \mathbf{b} \), whence addressing can be done by two pointers being incremented in index registers.

This is different with Cholesky decomposition when implemented with row by row storing of the matrix and with scalar products at the innermost level (compare section 2). As matrix rows are scattered across the memory segments, index incrementation is not applicable, and index arithmetic is quite expensive: there were 20 machine instructions in the inner loop, and even when taking advantage of the fact that each step accesses two elements of the same row, there were still 19, among them only 5 floating point instructions, two of which could be saved by accumulating the scalar product in a register. With columnwise storing we could reduce the inner loop to seven instructions, among them three FP-instructions, accessing memory only twice.

This resulted in an overall gain of nearly 25% in speed.

With the matrix stored by rows but contiguously, not as an array of unrelated pointers, we could apply the index incrementation technique, which allowed us to calculate all addresses in registers and gave nearly the same savings as columnwise storing.
We list here the CPU times for a 75x75 matrix and the number of machine instructions in the inner loop for the four different versions of the Cholesky solver:

<table>
<thead>
<tr>
<th>matrix stored</th>
<th># instr.</th>
<th>time (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>by rows</td>
<td>20</td>
<td>516</td>
</tr>
<tr>
<td>common row address</td>
<td>19</td>
<td>513</td>
</tr>
<tr>
<td>by columns</td>
<td>7</td>
<td>548</td>
</tr>
<tr>
<td>contiguously</td>
<td>7</td>
<td>562</td>
</tr>
</tbody>
</table>

5.3. Process sizes

The INTEL 80x86 processors access main memory via two part addresses, the first part of which defines a memory segment of 64 KB. Segments are addressed via four segment registers. Because addressing is faster when the segment registers need not be changed, there are several types of processes differing by the number of segments accessible. A small memory model process has exactly one code (instruction) segment and one data segment. A large memory model process may have several code and data segments, but each procedure and each array must be contained in one segment. A huge memory model, without restrictions on arrays (and other data structures), was not available in our system version.

Since most of the segment address arithmetic, i.e. arithmetic on the upper 16 bits of an address, can be avoided in inner loops, not much difference was to be expected between the performance of small and large memory model programs. For orthogonal elimination the difference was practically nil, because thanks to the 80266's "extra" segment register the segment addresses of the two vectors a and b could be kept and incremented within the processor. Since a segment can, even in double precision, hold a vector (or matrix row) of length 8192, it did not seem worthwhile considering the "huge" model case, where arrays are bigger than segments. In this case, address arithmetic might take a considerable amount of extra time.

For a compactly stored matrix, the large memory model gives no real advantage, since the matrix cannot exceed the size of one segment. So for Cholesky decomposition, we give here the table corresponding to the above one for the first three alternatives only, with the small model times repeated for comparison:

<table>
<thead>
<tr>
<th>matrix stored</th>
<th># instr.</th>
<th>time (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>small</td>
<td>large</td>
</tr>
<tr>
<td>by rows</td>
<td>27</td>
<td>515</td>
</tr>
<tr>
<td>common row address</td>
<td>26</td>
<td>512</td>
</tr>
<tr>
<td>by columns</td>
<td>7</td>
<td>548</td>
</tr>
</tbody>
</table>

It is noteworthy that the advantage of the small memory model (single data segment) processes nearly vanishes when the program data are structured in the right way.

6. Parallel execution examples

As an example we solve systems with 125, 250 and 500 unknowns. Assume the system matrices are contained in the files m125.b, m250.b and m500.b, respectively. Each of these systems can be solved on a single machine (provided there is enough storage) by e.g.

```
elim 250 tri.b cm250.b >empty
back tri.b >empty >solution
```

With the available programs the task can also be split in separate processes. The objective should be to distribute the work evenly across the processes (assuming the same type of computers for each process). The program reads:

```
<cm250.b elim 20 tri.b | elim 20 r2.b | elim 30 i3.b | elim 40 r4.b | elim 140 i3.b >empty
<empty back r5.b | back r4.b | back i3.b |
back r2.b | back tri.b >solution
```

On a single microvax II the processes are only executed pseudo-parallel. The overall execution time is quite similar to the one-process elimination described above. There is some overhead for passing the data from one process to the other one, and for synchronizing the processes. However the overhead is small compared to the actual work. The version with separate processes has even a slight advantage as it needs a little less disk I/O. The triangular matrices tri.b and r1.b, r2.b, etc. are stored as rectangular matrices. Thus the second program needs less disk I/O, as the five matrices tri.b, r2,b, etc. all together are smaller than the single matrix tri.b.

The really interesting case is when the processes are executed by different machines in the network. One simple way to do this is to install programs p1, p2, etc. on the machines m1, m2, etc. Each program does its part of the elimination and starts the next program on the next machine for doing the remaining elimination.

Thus the program p2 e.g. looks like

```
elim 20 x.b | rsh m3 p3 y.b
back x.b y.b
```

It thus eliminates 20 columns and gives the remaining matrix to be treated by p3 on m3. It expects to get the corresponding part of the solution back to store it in y.b. This part of the solution is extended by 20 elements using back. The extended solution becomes p2's standard output, so that the program calling p2 can accept it (to store it in a file y.b). Note that all programs p2 may use the same names x.b and y.b as they are now stored on different machines. Thus not only the processing powers of several computers but also their storage capacities are combined. The approach reflects the non-shared memory model.

The invocation of a remote process with rsh is quite costly. Each time rsh is executed permissions are checked and an environment is built up. The time for an rsh execution is in the order of several seconds. Under these conditions a speedup for small examples cannot be expected to be very good.
We give the results for our experiments. The first table shows how the work is distributed among the different computers. For the larger matrices the numbers are scaled by a factor 2 or 4 respectively. The second table gives the computer times.

<table>
<thead>
<tr>
<th>Number of elimination for a 125 by 125 matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>number procs</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution time for Cholesky method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>number procs</strong></td>
</tr>
<tr>
<td><strong>size of matrix</strong></td>
</tr>
<tr>
<td>125</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution time for Givens method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>number procs</strong></td>
</tr>
<tr>
<td><strong>size of matrix</strong></td>
</tr>
<tr>
<td>125</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>500</td>
</tr>
</tbody>
</table>

The overall execution times for different numbers of processors compare as follows:

<table>
<thead>
<tr>
<th>Speedup factors for Cholesky method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>size of matrix</strong></td>
</tr>
<tr>
<td>125</td>
</tr>
<tr>
<td>250</td>
</tr>
<tr>
<td>500</td>
</tr>
</tbody>
</table>

Note the following:
- For small problems the overhead of using 4 times dominates the computer times. Only for the larger examples a good speedup can be observed.
- At least 3 processors are needed in order to obtain a speedup of 2, and 5 processors for a speedup of 3.
- The enormous time needed by the Givens method on one computer to solve the 500 by 500 example is due to lack of storage and a high paging rate.

Discussion

Obviously our parallelization does not exploit the available processors with 100\% efficiency. This is already a feature of the basic parallelization scheme of Sameh and Kuck [22]. In this algorithm the number of elimination steps performed in parallel varies between 1 and n/2. For our modification this means that if in the extreme case n processes are used, each eliminating one column, then at most n/2 of these processes will be busy at a time. Most of the time fewer will be busy.

In general, each process must wait for the previous process in the pipeline to supply its data. If the first process has to eliminate m columns, then the second (and all further) processes do not get any data before the first m/2 rows are finished by the first process. Even on the next rows the speed of the first process will limit the throughput. On these rows the second process has to perform 1, 2, 3, etc. elimination steps, respectively. On the other hand the first process has to perform m elimination steps in order to prepare the data for the second. Only later, depending on the setup, the second process may come to be the limiting factor.

In the following we use idealizing assumptions neglecting all overhead of communication, and also neglecting the preparation necessary before eliminating one element. One work unit is the operations for forming the linear combination of two row vectors in the elimination on one position, i.e. forming \( c \cdot a_k + s \cdot a_p \) and \( -s \cdot a_k + c \cdot a_p \) in the case of Givens rotations.

Under these assumptions we demonstrate the performance of the elimination of a 40x40 matrix using 4 processes. The following table shows after how many work units the elimination of the k-th row is finished in the p-th process. Asterisks (*) mark where the previous process has limited performance, not the local process itself. The assumed distribution of work is 5, 6, 8, 21 columns to be eliminated in the corresponding processes.
A sequential elimination for a system of the same size would need an execution time of 20540 work units. So in this case the speedup is 2.65 and the processor utilization is 66%. Here we have experimentally determined an optimal distribution of work.

Finding a general optimal strategy is quite difficult. Assume the time when process $p$ finishes eliminating row $m$ is $w_{p,m}$. The time process $p$ needs to perform the elimination of last $m$ is denoted by $t_{p,m}$. Further let $m_p$ denote the number of columns to be eliminated in process $p$. If $n_p = \sum_{i=p}^{q} m_i$ is the size of the system, then

$$w_{p,m} = \begin{cases} 0 & \text{for } p \neq 0 \text{ or } m \leq \sum_{i=q-p+1}^{q} m_i \\ \max \left\{ w_{p-1,m}, w_{q-m+1} + t_{p,m} \right\} & \text{else} \end{cases}$$

This formula recursively uses the max function so that optimization must be expected to be difficult.

Heuristically one can argue that each process must do approximately the same amount of work. As the elimination needs work proportional to $n^3$, one can demand

$$\left( \sum_{r=q-p+1}^{q} m_r \right)^3 = \frac{n^3}{q} 15p^3q,$$

where $q$ is the number of processes one wants to use and $n$ is the size of the system to be solved. This argument has been used to determine the distribution used in the above examples.

Another simple argument can be derived for the case when each process eliminates the same number $m$ of columns. In this case always the previous process limits the speed, that is just one branch of the maximum must be considered. In this case the total time is the same as for completing the elimination of the first $m$ columns plus the elimination of the last row, starting with the $(m+1)$st element. This gives

$$p = \frac{(n-1)n(2n-1) - (n-m-1)(n-m)(2(n-m)-1)}{6} + \frac{(n-m)(n-m-1)}{2}$$

for the total time with parallelization. Using

$$S = \frac{(n-1)n(2n-1)}{6}$$

for the sequential time we find that the speedup for $n$ large, $m$ fixed is

$$\text{speedup} = \lim_{n \to \infty} \frac{S}{P} = \frac{2n}{6m+3}$$

Taking into account that $n/m$ processors are used we have that asymptotically the efficiency must be better than

$$\text{efficiency} = \frac{2m}{6m+3} \leq \frac{2}{9}$$

As usually better load distributions than with constant $m$ are possible, this is just a lower bound for the efficiency.

This may seem unsatisfactory. There are two ideas how the performance may be improved.

1. Using the fact that the first processes terminate early one may try to assign the final processes to the same processors again. This would lead to a structure with a ring of processors. The data would be moving around in somewhat more than a full circle. Fig. 2 shows how in the example of fig. 1 processors are assigned to matrix rows: the trick described reduces the number of processors required from nine to five.
The approach supports a clean structuring of large projects and a clean definition of interfaces. Code reusability is improved. In fact, even for our small example problem, the modules for formatted I/O and some basic communication routines could be taken from a completely different project [21].

All this is achieved without sacrificing efficiency. Though the highest level has a comparatively large overhead, this is unimportant as the critical parts are implemented carefully. The innermost loop can even be implemented in assembly language so that the hardware is optimally used. It is quite important to note that this can be done without loss of portability. Using conditional compilation, C codes are compiled as a default, and only where the corresponding assembler modules are available, these are compiled.

The implementation of parallel execution is open to criticism. The remote shell mechanism is too slow. The overhead is tolerable only for large problems. Here a more efficient tool is necessary.

However the parallel programming model itself has a number of advantages. It is simple, so that it can be easily used. On the other hand it has quite a good expressive power. The shell constructs for asynchronous execution correspond closely to those of a typical parallel programming language, like Occam. So there should be the question whether parallelism justifies a new programming language. The concepts of parallelism are essentially orthogonal to those arising from classical sequential programming. This fact makes our approach of using a separate tool for parallel programming possible.

Clearly, there are limitations to our method. The concept basically reflects MIMD computers with non-shared memory. Thus one could use the special features of architectures with shared memory only by using the corresponding UNIX-V system calls within the C program. This would of course destroy the advantage of purely sequential (and thus safe and hardware-independent) programming of the modules. On the other hand our programs would, of course, run on a shared memory machine without problems, however not using all available hardware features.

For general parallel programming there may be efficiency problems. We had problems because we have partly been using an unsuitable hardware configuration (a general purpose network of microcomputers) and unsuitable software (like the rsh facility). Furthermore UNIX process generation and management is slow. This, however, depends on the system kernel and there more efficient implementations of the same functionality are possible.

Some restrictions are implicit in the approach. Communication by pipes means that all data to be transferred must be copied from one process' address space to the other's. This presents some overhead and there are problems that become difficult to parallelize if no other communication means are available. This is a price we have to pay for the simplicity of the basic concept.

---

2. As in all cases, where there is a large setup time, one may think of using the algorithm in a pipelined way for the solution of several linear systems. Then a new matrix of unknowns starts being eliminated in the first process, before the previous matrix has finished in the last one. The present codes could be easily adapted to such an extension. A possible application for such a linear-solver-pipeline could be the nested dissection algorithm (see e.g. [5]), where for solving a large sparse system of equations, many smaller but full ones have to be solved.

7. Summary and Conclusions.

In this paper we have shown an example of modern software engineering for parallel numerical algorithms. Our approach exploits the concepts of an advanced computer environment, the UNIX Operating System. Several tools and languages are combined to implement an efficient and elegant solution. Basic modules, programmed in a classical programming language, are combined to more complex applications using the shell language. Distributed computation is programmed using the shell, pipes and the remote shell communication software.

The C language is used to program the basic modules. However one very important advantage of our concept is the possibility to combine all languages. While old codes written in Fortran can still be used, new modules may be written in a more flexible language. For experimenting one may choose to use an interpreter language like Basic and other subproblems may best be solved by using one of the specialized UNIX tools.
Still it seems sufficient for quite a range of problems. Currently we are using the technique for implementing multigrid methods for the solution of 2D and 3D elliptic problems [21, 24]. The concept is also well suited to other numerical problems, e.g., in optimal control theory. Experiments to utilize a similar approach for optimal control software are currently made at the DFVLR.

Acknowledgements

We wish to thank Prof. Dr. Chr. Zenger for the support of this work and many interesting and constructive discussions. We are also indebted to Prof. Dr. Chr. Reinsch for valuable comments.

References

3. Chan, T. and Tuminaro, R., Implementation and Evaluation of Multigrid Algorithms on Hypercubes, Copper Mountain III.