Evaluation of Compiler Generated Parallel Programs on three Multicomputers

ROLAND RÜHL

Integrated Systems Laboratory, Swiss Federal Institute of Technology
Gloriastrasse 35, 8092 Zurich, Switzerland
E-mail: ruehl@iis.ethz.ch

Abstract

Distributed memory parallel processors (DMPPs) have no hardware support for a global address space. However, conventional programs written in a sequential imperative language such as Fortran typically manipulate few, large arrays. The Oxygen compiler, developed as part of the K2 project, translates conventional Fortran code, augmented with code and data distribution directives, into C programs including SEND/RECEIVE *communication primitives. The compiler directives, which are either supplied by the user, or for simple programs generated automatically, support a global name space through a run-time mechanism called data consistency analysis. We report in this paper the performance of seven parallel programs generated by Oxygen for three DMPPs, namely for a Parsytec Supercluster, an iWARP, and for the Fujitsu AP1000. All machines were configured as* 8×8 *tori.*

1 Introduction

Programming distributed memory parallel processors (DMPPs or *multicomputers*) with imperative, standard languages enhanced with compiler directives for data and program decomposition and distribution and for support of global name space is becoming a widely accepted approach. The ongoing effort to define the characteristics of "High Performance Fortran (HPF)" will increase the awareness of the users for this programming practice. HPF consists of Fortran 90 enhanced with compiler directives, and its definition is being worked out by a group including most major parallel processor manufacturers and some universities.

As part of the K2 project [1], we have developed the parallelizing compiler *Oxygen*. It generates parallel DMPP code from Fortran 77 with compiler directives; the directives allow data and code distribution and support a global name space. We have used Oxygen to generate code for the following platforms: (1) the DMPP simulator K9 [2], (2) a Parsytec Supercluster SC256 [3], (3) an iWARP [4], and (4) the Fujitsu AP1000 [5]. This paper summarizes results obtained from parallel programs generated for the latter three machines. A detailed description of the compiler and collection of simulated performance measurements can be found in [6, 7].

The paper is organized as follows: First we relate our research to similar ongoing efforts. Then we will give an overview of the compiler and describe the SC256, iWARP, and AP1000 machines, emphasizing on architectural parameters important for the compiler. Finally we will describe the benchmark programs and present the results.

2 Related work

Many research groups are currently developing parallelizing Fortran compilers, among them Parafrase II ([8], CSRD University of Illinois), PTRAN ([9], IBM), Parascope and the Fortran D compiler ([10], Rice University), the Kali compiler (ICASE), and SUPERB (University of Vienna).

Three of the above groups specifically address the problem of generating code for DMPPs. The user defines data distributions explicitly and the compiler then generates parallel code based on these distributions. Callahan and Kennedy [11] first defined Fortran directives for a virtual DMPP with an arbitrary number of processors and asynchronous message-passing. The physical platform used was an iPSC hypercube; directives (DISTRIBUTE and DECOMPOSE) determined the distribution of variables. Gerndt [12] defines partitions for nonlocal arrays and partition classes to solve aliasing problems with the SUPERB compiler which generates code for the Suprenum DMPP. Expressions accessing nonlocal array elements are *masked* statically according to the ownership of the elements. Koelbel at al. [13] define Kali Fortran which allows the explicit distribution of arrays and the parallelization of DO loops on message-passing architectures. The Kali compiler can compile parallel loops into an *inspector* and an *executor*. The inspector consists of a loop which checks the locality of data referenced inside a parallel loop body. An algorithm performing neighbor relaxation on an unstructured grid is compiled for the iPSC/2 and the NCUBE/7, and performance measurements are provided. Koelbel at al. conclude that run-time analysis is efficient for iterative algorithms which can reuse communication patterns after they have been generated in a first, expensive iteration.

Rogers and Pingali [14] present a method which, given a sequential program and its data partition, performs task partitions to enhance locality of references. Saltz and his colleagues [15] have also addressed the issue of testing locality of distributed array elements dynamically, when static array subscript analysis fails; a program is presented which

---

This research was supported in part by the Swiss Nationalfonds grant numbers 21-25244.88 and 21-30143.99 and in part by the European Institute of Technology.

ICS '92-7/92/D.C., USA
1992 ACM 0-89791-485-6/92/0007/0015...$1.50

generates communication patterns at run-time, from data structures produced at compile-time.

Several groups have described methods to estimate statically the quality of a given data distribution. Li and Chen [16] and Gupta and Banerjee [17] base their work on *pattern matching.* Balasundaram et al. [18] describe a training set of kernel routines to estimate costs of communications and computations, similar to (but more sophisticated than) the measurements done in section 4.1. Because of the assumed high communication latency, all three papers emphasize the generation of collective communication routines.

All of the above research is based on the following assumptions:

- The target DMPP uses *memory communication* (also called *message passing*) with high latency and transparent routing implemented in hardware.
- The data distribution is given at compile-time and the code distribution is derived implicitly from the data distribution.
- Only the owner of a data item is allowed to compute on that item.
- If a statement depends in its control-flow on non-local variables it is not allowed to access non-local variables. The control-flow dependence may be caused either explicitly with an `if` statement, or implicitly through non-local variables used in index expressions.

As an example let us consider the following loop:

```
      do 10 i=1,n
         a(i) = b(q(i))
10    continue
```

Similar code can be found in important scientific applications such as finite-element based algorithms. The above assumptions require a, q and the loop to be distributed with the same static partition and mapping. This is not efficient for many codes which have this structure, as for instance the PGMRES code presented in section 5. Therefore, we included in Oxygen a mechanism to determine ownership of distributed data dynamically (namely MULTICOPY variables as explained in the next section).

Most of the work on compilation for DMPPs featuring systolic communication has been done by H.T. Kung and his colleagues. Tseng, Lam, and Kung define a language extension to the Warp language W2 [19] called AL [20]; communication is hidden and the programmer chooses between different data and loop distribution primitives. Tseng's doctoral dissertation describes the implementation of AL on Warp using the DARRAY primitive for data distribution and the ALIGN primitive to distribute loops [21]. That research concentrates on static dependence analysis and on the generation of communication patterns that best exploit the topology and architecture of Warp. Parallelization happens strictly at compile-time and the automatic generation of the compiler directives is not tackled. Tseng's performance measurements of compiler-parallelized LU and QR decomposition, 2DFFT, different SOR methods, and various LINPACK routines are of absolute excellence [22]. However, it is not known how they would scale to a machine with more than the ten processors of Warp.

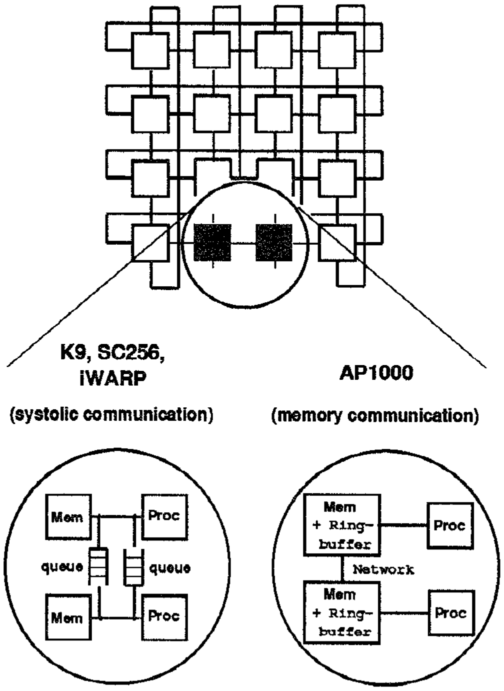

Figure 1: *The machine model of Oxygen is a two dimensional torus of processing elements communicating through send and receive primitives. Oxygen has been ported to two classes of DMPP systems: DMPPs featuring systolic communication (Parsytec SC256 and iWARP), and DMPPs featuring memory communication (Fujitsu AP1000).*

3 An overview of the compiler

In what follows, we will define a target DMPP model for Oxygen and introduce the term *data consistency analysis.* Then we will explain how the compiler is used and describe the programming paradigm implemented in Oxygen through directives.

3.1 The machine model

The machine model assumed by Oxygen is a two dimensional torus (see Fig. 1) of processing elements (PEs). Platforms such as the SC256, iWarp, the AP1000, and K9 implement this topology. Studies by Dally and Johnsson support the choice of a torus[23, 24] for a limited number of processors. To summarize their results, a torus outperforms other topologies when the number of PEs is no greater than 256 (Dally) or 64 (Johnsson).

When implementing Oxygen on the SC256, iWARP, and K9, we assumed *systolic communication*: PEs communicate through raw data *send* and *receive* primitives. No message-passing is involved: data are simply dispatched from a sender to a receiver using memory-to-queue (or register-to-queue) transfers, flow control is enforced by the queues connecting the PEs. PEs can only communicate to their four nearest neighbors. If information is to be sent along a route with more than one communication hop, explicit forward statements have to be executed by all PEs traversed by the route.

The AP1000 implements *memory communication* (also called *message passing*). A *routing controller* computes routes between message senders and receivers. PEs can not only communicate to their nearest neighbors, but to any processor in the system. Computation on intermediate PEs on a route is not disturbed by storing and forwarding messages, which is taken care of by the routing controller and done in parallel to the computation. Although this hardware message passing increases programming comfort for both the user and compiler implementor, it is generally associated with higher communication latency. Raw data communication can be implemented with a latency of only a few processor cycles. In message passing systems, time has to be spent on managing routing buffers and on computing the route; communication latency is therefore higher.

3.2 Data consistency analysis

Let us introduce the following definition:

> **Ownership** (of read/write shared variables): *A processing element (*PE*) in a* DMPP *is said to be the* OWNER *of a read/write shared variable if the variable's latest update is stored in the* PE*'s local memory. A* PE *in a shared memory multiprocessor is said to be the owner of a read/write shared variable if the variable's latest update is stored in the* PE*'s cache memory.*

On bus connected, shared memory multiprocessors, ownership is dynamically determined by the machine's cache coherency protocol. Parallelizing compilers for this class of systems (e.g. Parafrase-II) can therefore ignore the problem of determining the dynamically varying ownership.

Parallelizing compilers for DMPPs decompose the problem's domain, allocate the subdomains to the local memories of the PEs, organize the interprocessor communication activities, and enforce *data consistency* across the local memories. We have the following definition:

> **Data consistency analysis** *is the process of determining ownership in a* DMPP*. Enforcing data consistency requires run-time activities capable of dynamically tracking ownerships.*

On DMPPs, data consistency analysis becomes part of the parallelizing compiler responsibilities, and requires in the general case that extra run-time activities are generated by the compiler. This issue is independent of the machine interconnection and the interprocessor communication mechanism: it is the lack of a global address space that creates the need for consistency analysis. On Hypercubes like the iPSC/2 and NCUBE/2, systolic computers like WARP and iWARP, Transputer based DMPPs like the SC256, array processors like the Fujitsu AP 1000, or Thinking Machines CM5, a parallelizing compiler has to perform run-time data consistency analysis.

Oxygen directives allow the explicit partitioning and mapping of loops and arrays. However, any processor may access any element of a distributed data structure. Communication primitives (SEND/RECEIVES) are generated, transparently to the user. The compiler generates code to perform data consistency analysis at run-time.

3.3 User view of the compiler

Fig. 2 gives an overview of the compiler. The compiler consists of two major components:

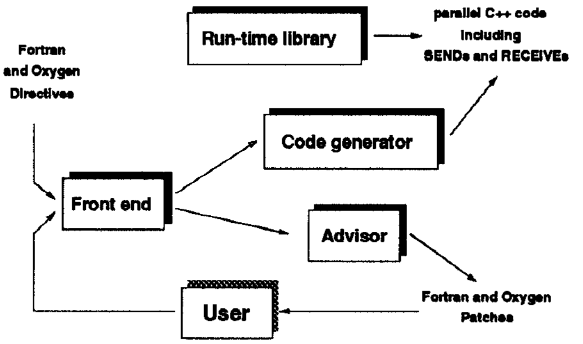
Run-time library
parallel C++ code including SENDs and RECEIVEs
Fortran and Oxygen Directives
Code generator
Front end
Advisor
Fortran and Oxygen Patches
User

Figure 2: *The front end generates an intermediate representation from the Fortran code which is then used by Oxygen's major two components: (1) the code generator which transforms the intermediate representation into C/C++ and send/receive statements, and (2) the advisor which, based on dependence information, suggests modifications to the Fortran source to increase performance.*

1. The code generator, which translates Fortran with Oxygen directives into parallel C programs including communication primitives. Before the code generator is invoked, the source is translated by the front end into an intermediate representation, consisting of a control flow graph and a symbol table.
2. The advisor, which suggests a collection of *source patches*, i.e., modifications to improve the performance of the generated parallel program. Given the intermediate data structures generated by the front end, a dependence analysis will be carried out to generate a dependence graph. The advisor uses this dependence information to detect parallel loops and tries to partition and map these loops and the main data structures used inside the loops onto the torus.

In addition to the tasks performed by a conventional compiler, Oxygen's code generator must also generate code to ensure data consistency at run-time. This will be explained in more detail in the next section. A detailed description of the advisor is out of scope of this paper. The interested reader is referred to [25]. At the moment the advisor correctly proposes patches for the three linear algebra benchmarks described in section 5.

3.4 Programming paradigm

Oxygen assumes that a Fortran program can be decomposed into a sequence of code blocks which by default run in parallel on all PEs. (Compiler directives may restrict the execution of a code block to one or more PEs.) Code blocks may be either "local" or "public." Local blocks, hereafter referred to as L_BLOCKS, may or may not run in parallel on the PEs, but in either case their computation is local and does not activate interprocessor communication. Public blocks, hereafter referred to as P_BLOCKS, always run in parallel and activate interprocessor communication because they operate on data structures allocated across PE boundaries.

The model of the program structure embedded in Oxygen is shown in Fig. 3. The leftmost column shows how the uniprocessor code is decomposed, via compiler directives, onto the various blocks. The center column shows the

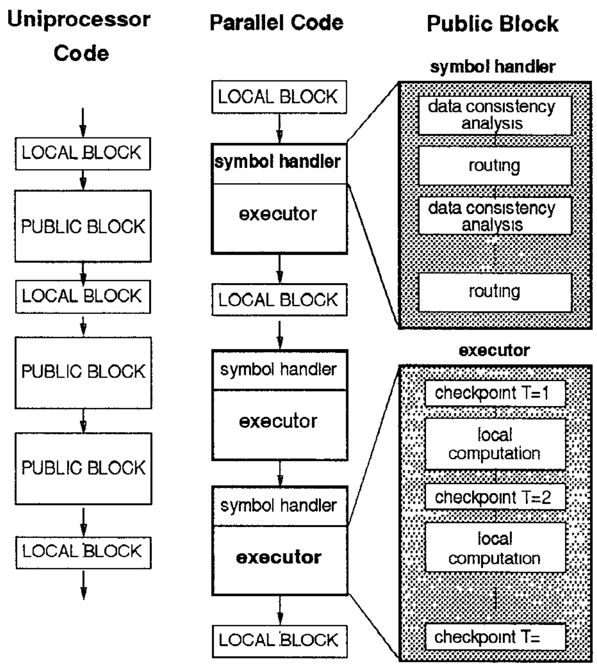

Figure 3: *The model of uniprocessor and parallelized code embedded in Oxygen. The uniprocessor code is decomposed, via compiler directives, into local and public blocks. Local blocks may run in parallel, but do not require any implicit interprocessor communication. The symbol handler performs data consistency analysis in public blocks, and is extra code inserted by Oxygen during compilation. The executor performs the part of the original public block computation present in the uniprocessor code and exchanges data with the appropriate processors based on the data consistency analysis of the symbol handler.*

structure of the source code—common to all PEs—generated by Oxygen. Every P_BLOCK is decomposed into a *symbol-handler* and an *executor*, whose structure is shown in the rightmost column; the symbol-handler is a sequence of data consistency analysis and routing phases to interrogate PEs about ownership of shared variables, while the executor is a sequence of computation phases intermixed with communication checkpoints. Data consistency analysis in the symbol-handler constructs the data structures that will be used by the executor to execute interprocessor communication primitives at the communication checkpoints.

The checkpoints that separate blocks of local computation within the executor do not synchronize the entire machine, but only the specific PEs involved in the data exchange. It follows that we can not restrict storage class qualifiers to be either "private" or "shared." An association of a data item to a storage class (declared through a compiler directive) should define the degree of locality, i.e: (1) which processor (the *owner*) allocates the item, and (2) which processor is allowed to access (fetch or update) it. We defined three storage class qualifiers:

- LOCAL data are only known to and may only be accessed by their owner.
- SINGLECOPY data. Their ownership is defined at compile-time as part of a directive and it remains the same throughout the lifetime of the program, and it refers to one PE. All other PEs may access and update the data through interprocessor communication.
- MULTICOPY data. The *ownership* changes dynamically. A PE becomes the owner when either it updates or it fetches a data item and no update of the same item is executed by other PEs at that time. In the latter case, an item may have multiple owners. PEs that are not owners can access the latest update of the variable through interprocessor communication. The latter is always performed toward the closest owner (in the topological sense).

3.5 The directives

Communication primitives are usually generated dynamically when nonlocal data are accessed inside P_BLOCKs. P_BLOCKs are enclosed between START PARBLOCK and END PARBLOCK directives. There are also a few collective communication primitives, to broadcast a local variable to all other processors, or to perform global reduction operations, for which communication primitives are statically generated.

A special directive (COPY) can be used to copy distributed arrays declared with the same Fortran statements but mapped and partitioned with different directives. This is useful when different algorithms used in an application operate on the same data structures but require different array distributions to perform efficiently. The 2DFFT benchmark of section 5.1 is such an application: first a two-dimensional matrix is scanned column-wise, than it is scanned row-wise.

Most directives refer to the decomposition of the data structures and to the decomposition of the program structure. The overall goal was to keep them as simple as possible in order to ease their automatic generation. Because of this, and because they are rather conventional, we skip most of the details in this paper. The interested reader is referred to [26].

Oxygen decomposition directives, which apply equally to data arrays and control loops, serve two purposes, namely *partitioning* and *mapping* of data elements or loop indices. Partition directives are SPLIT and SCATTER, mapping directives are RING, ROWWISE, and COLWISE.

Data distribution

As an example, let d^2 be the number of PEs, and $A(i)$ $(i = 1, 2, \ldots, n)$ an array variable to be decomposed on the torus. Let also m be the number of subdomains s_j $(j = 1, \ldots, m)$ into which the array index i must be partitioned. We have $m = d$ for ROWWISE or COLWISE, and $m = d^2$ for RING. SPLIT and SCATTER perform the following array index decomposition:

SPLIT

$$\begin{aligned} s_1 &= \{A(1), \ldots, A(k)\}, \\ s_2 &= \{A(k+1), \ldots, A(2k)\}, \\ &\vdots \\ s_m &= \{A((m-1)k+1), \ldots, A(n)\}. \end{aligned}$$

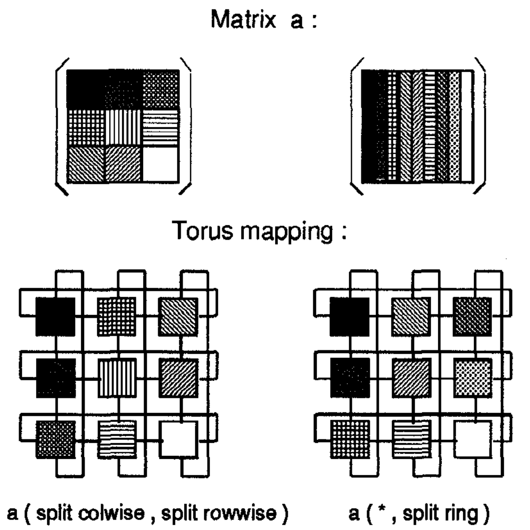

Figure 4: *Different domain decompositions via compiler directives. The same matrix* a *is decomposed and allocated onto a* 3×3 *torus in two different ways. The* $\star$ *indicates that the first dimension of* a *is not decomposed across* PE*s.*

SCATTER

$$
\begin{aligned}
s_1 &= \{A(1), A(m+1), \ldots\}, \\
s_2 &= \{A(2), A(m+2), \ldots\}, \\
&\vdots \\
s_m &= \{A(m), A(2m), \ldots\}.
\end{aligned}
$$

where $k = \lfloor n/m \rfloor + 1$. One of the three mapping directives can then be applied to the subdomains. Two domain decomposition examples together with their corresponding compiler directives are shown in Fig. 4 for a 3×3 torus and a two dimensional matrix a.

Code parallelization

Loop indices are treated similarly to array indices, i.e., they are partitioned and mapped using the same five directives. Index strides different from one are handled by Oxygen transparently. ROWWISE and COLWISE mapping directives can be used in two different contexts: (1) to map the indices of double nested loops on the torus, and (2) to map a simple loop on all rows (columns) of PEs. In the latter case all PEs in the same row (column) perform the same index computation.

Let us assume we want to map a double nested loop on a torus. ROWWISE applied to the outermost loop maps the same $m = d$ index subdomains onto the d PEs in the same torus column; COLWISE applied to the innermost loop maps the same $m = d$ index subdomains onto all PEs in the same torus row. Finally, rectangular blocks of the two-dimensional index space are decomposed on each PE with SPLIT:

```
c$      LOOP SPLIT ROWWISE
        do 10 i = 1, n
c$         LOOP SPLIT COLWISE
           do 20 j = 1, m
                        .
                        .
   20          continue
   10       continue
```

4 Characteristics of the target machines

4.1 The q parameter

To characterize the target machines, let us define the PE local communication/computation ratio q as follows:

$$q = \frac{t_{io}}{t_c},$$

where t_{io} is the time for a PE to send or receive a double-precision quantity to or from a neighbor PE, and t_c is the time the PE takes to perform a double-precision multiply-add operation as in DAXPY [27]. We have

$$t_{io} = \frac{t_{oh} + t_{dp}M}{M},$$

where t_{oh} is the communication start-up latency, t_{dp} is the time to transfer a double, and M is the message length (in doubles). Speedup obtained by parallelizing any non-trivial program on a DMPP will depend on the value of q on that machine, because the smaller q, the lower the communication overhead is. On an architecture with systolic communication we assume $M = 1$, and on an architecture with memory communication $M = \infty$ (i.e., communications are performed with maximum message length). Note however, that q is not an absolute figure of merit for a given architecture, it only compares local floating point performance to communication performance.

An exact value of q on a specific architecture can not be measured, because (1) execution time of programs is also influenced by other operations than communications or floating point multiply-additions, and (2) the time to access operands depends on where the operands are stored. We have computed an estimate of q by measuring the execution time of communication primitives and double precision floating point add and multiply operations. As shown in Fig. 5, t_c was estimated for both scalar and indexed computation.

4.2 Parsytec cluster

We have used the Parsytec Supercluster SC256 installation at the RWTH in Aachen, Germany. On such system,

```
/* scalar computation: */
double a, b, c;
for (i=NRIT-1; i >= 0; i--) {
    b = a + b * c;
}

/* indexed computation: */
double a, *b, *c;
for (i=NRIT-1; i >= 0; i--) {
    b[i] = b[i] + a * c[i];
}
```

Figure 5: *Loops to measure the computation speed.*

the user can configure arbitrary DMPP networks of transputers (with a maximum of four incomming and outgoing communication channels per processor). We have performed our measurements on a network configured as 8 × 8 torus of T800 processors. Each processor runs at 25 MHz clock speed and is configured with 4 MBytes dynamic memory. As backend C compiler we have used the INMOS toolset compiler version 1.0.

	t_c	$t_{io}(M=1)$	q
stack, code on SRAM, indexed	3.8	12.7	3.3
stack, code on SRAM, scalar	2.4	12.7	5.3
stack on SRAM, indexed	4.8	13.0	2.7
stack on SRAM, scalar	3.3	13.0	3.9
DRAM only, indexed	6.4	13.5	2.1
DRAM only, scalar	4.8	13.5	2.8

Table 1: *Measurements of* q *on 64 processors of a Parsytec cluster: the costs of a double precision multiply and add* (t_c) *are given for both vector and scalar operations in* μ*sec. Both communication costs* (t_{io}) *and computation costs* (t_c) *depend on the use of the internal memory of the T800.*

The T800 processor features 4 KBytes of fast on-chip RAM. The user can store data and instructions in this RAM. Table 1 summarizes estimates of t_c and $t_{io}(M=1)$ on the SC256 for different memory mappings. Programs compiled by Oxygen allocate the program stack in on-chip RAM.

4.3 iWARP

	t_c	$t_{io}(M=1)$	q
with binding, indexed	24	19	0.8
without binding, indexed	24	11	0.5
with binding, scalar	8	19	2.4
without binding, scalar	8	11	1.4

Table 2: *Measurements of* q *on iWARP: the costs of a double precision multiply and add* (t_c) *are given for both vector and scalar operations in processor cycles. Communication costs* (t_{io}) *for each double precision send or receive are given with and without gate binding.*

iWARP is a parallel architecture developed jointly by Carnegie Mellon University and Intel Corporation. The iWARP communication system supports two interprocessor communication styles: both memory communication and systolic communication. Although we would expect higher performance for some of our benchmark programs, when using iWARP's memory communication mechanism (which provides rather low communication latency), we have only used the systolic mechanism. Our results will show, that generated code performs efficiently even if the compiler handles route computation and message forwarding.

Our measurements have been collected on a 64 processor system installed at Carnegie Mellon. Each processor of that system runs at 16 MHz clock speed and is configured with 500 KBytes static memory. We have used the pre-release software version 2.5.2[1]

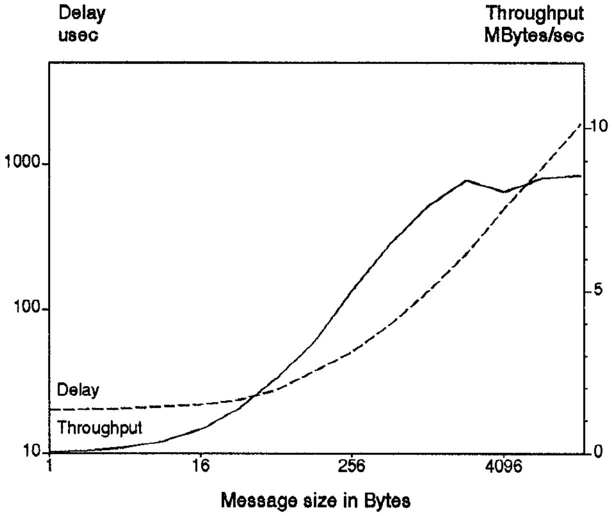

Figure 6: *Delay* (t_{io}) *and throughput* ($8 \times M/t_{io}$) *of neighbor to neighbor communication in the AP1000 for variable message size.*

Although an iWARP processor features four incomming and four outgoing physical communication channels, communications refer to two *Gates* (i.e. processor registers). The Gates must be bound explicitly to a channel. The execution of such *bind* instruction effects the communication bandwidth but can often be avoided (for instance when sending several times to the same channel). Table 2 summarizes estimates of t_c and $t_{io}(M=1)$ for iWARP.

4.4 The AP1000

	t_c	$t_{io}(M=\infty)$	q
in cache, indexed	1.15	0.94	0.8
out of cache, indexed	1.91	0.94	0.5
scalar	1.00	0.94	0.9

Table 3: *Measurements of* q *on 64 processors of an AP1000, under the assumptions that messages have infinite length. The costs of a double precision multiply and add* (t_c) *are given for both vector and scalar operations in* μ*sec. The vector operation speed is dependent on the operands being in cache or not. For long vectors a DAXPY iteration will be almost twice as slow as for short vectors stored in cache.*

The AP1000 is not a commercial system, but an experimental multiprocessor with 64 upto 1024 PEs. The AP1000 is hosted by a SUN4-330. Each PE consists of a 25 MHz SPARC with FPU, 16 MBytes DRAM (organized in four interleaved banks) and 128 KBytes direct mapped cache memory. An additional message controller (MSC) and a routing controller (RSC) manage interprocessor communication. Three communication networks are available: the

---

[1] This software works on an engineering prototype (*B-step*) which does not fully support Long Instruction Word optimization.

Torus network (T-net) for point-to-point communication between PEs, the Broadcast network (B-net) for host to PE communication, and the Synchronization network (S-net) for barrier synchronization. We used for all our measurements the AP1000 operating system CellOS1.1 and the SUN C compiler SC1.0 as backend for Oxygen.

Memory communication built in the AP1000 system has relatively higher startup latencies than the systolic mechanisms used on the other platforms. We measured in Fig. 6 communication delay (t_{io}) and throughput ($8 \times M/t_{io}$) of the AP1000 when doing point-to-point communication using the T-net. Throughput reaches its maximum, when messages are longer than 1024 Bytes. In the ideal case the compiler can pack information and communicate long messages. Table 3 summarizes t_{io}, t_c, and the q value for infinite message length.

5 Benchmark Programs and Results

Seven test programs have been used to evaluate the performance of the compiler: three dense linear algebra problems, two signal processing algorithms, one successive over-relaxation, and the solution of a sparse linear system of equations stemming from the simulation of semiconductor devices.

5.1 Benchmark description

Linear algebra

GAUSS. A linear system of equations is solved using Gaussian elimination with partial pivoting. The program uses double precision arithmetic.

ORTHES. A real square matrix is transformed through Householder similarity transformations to an upper Hessenberg matrix. The algorithm is used to compute all eigenvectors and eigenvalues of an unsymmetric matrix [28]. The program uses double precision arithmetic.

SVD. The EISPACK [29] singular value decomposition subroutine. All singular values of a rectangular matrix are computed using a QR iteration. The program uses double precision arithmetic.

Signal processing

GFFT. The generalized FFT processes complex vectors the size of which may not be a power of two, and is based on the prime factor decomposition of the vector size. The prime factor decomposition and the access pattern to the distributed vectors are therefore not known at compile-time. The program uses double precision complex arithmetic.

2DFFT We have implemented the 2 dimensional FFT of an image of $N \times N$ pixels. In a first step, an FFT of each column of the input matrix is computed. Then the matrix is transposed using the COPY directive, and another FFT is applied to each column of the transposed matrix. This algorithm is often used to compute a fast correlation of two pictures. The program uses single precision arithmetic.

SOR

In FLUID, the analysis of the behavior of airfoils in subsonic, transonic, and supersonic regime is carried out by solving a two-dimensional, steady-state transonic small disturbance equation. A finite difference method solves the problem iteratively with a successive over-relaxation (SOR) scheme. The results refer to 20 iterations of the SOR algorithm. Communications are generated dynamically in the first iteration and then reused in each iteration of the algorithm. The program uses double precision arithmetic.

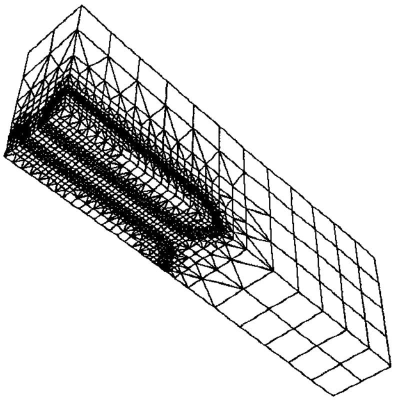

Figure 7: *Finite element grid representing a submicron DRAM cell with trench capacitors.*

PGMRES

Large, sparse systems of equations arise from the application of discretization schemes (e.g., finite elements) to partial differential equations, and may exhibit irregular sparsity patterns. While direct solution techniques based on Gaussian elimination have proved to be extremely robust and stable, they are no longer competitive with iterative methods for problems with large numbers of unknowns, as in three-dimensional applications [30]. The efficient parallelization of such iterative methods on DMPPs is still an open research topic; preliminary results are, however, promising [31].

The Fortran package SPARSKIT [32] has been implemented by Y. Saad to provide basic subroutines to handle sparse matrices. The package also includes the preconditioned iterative sparse system solver PGMRES [33]. We parallelized PGMRES with Oxygen directives and applied it to a system of equations stemming from the finite-element simulation of a semiconductor. The system of equations has 15,564 unknowns and 133,663 nonzeros. The finite element grid is shown in Fig. 7. As already mentioned in section 2, PGMRES was parallelized using MULTICOPY variables because of the low density of the system of equations. The program uses double precision arithmetic.

5.2 Results

Some large problems—among others PGMRES—could not be benchmarked on the iWARP system, because of their memory requirements. Note also, that for large data sizes, we had to estimate the serial execution time of the programs. This was done by measuring the execution time on one T800 (respectively iWARP or AP1000) processor for small data sizes, and then extrapolating these numbers using serial execution times of the same programs for larger problem sizes

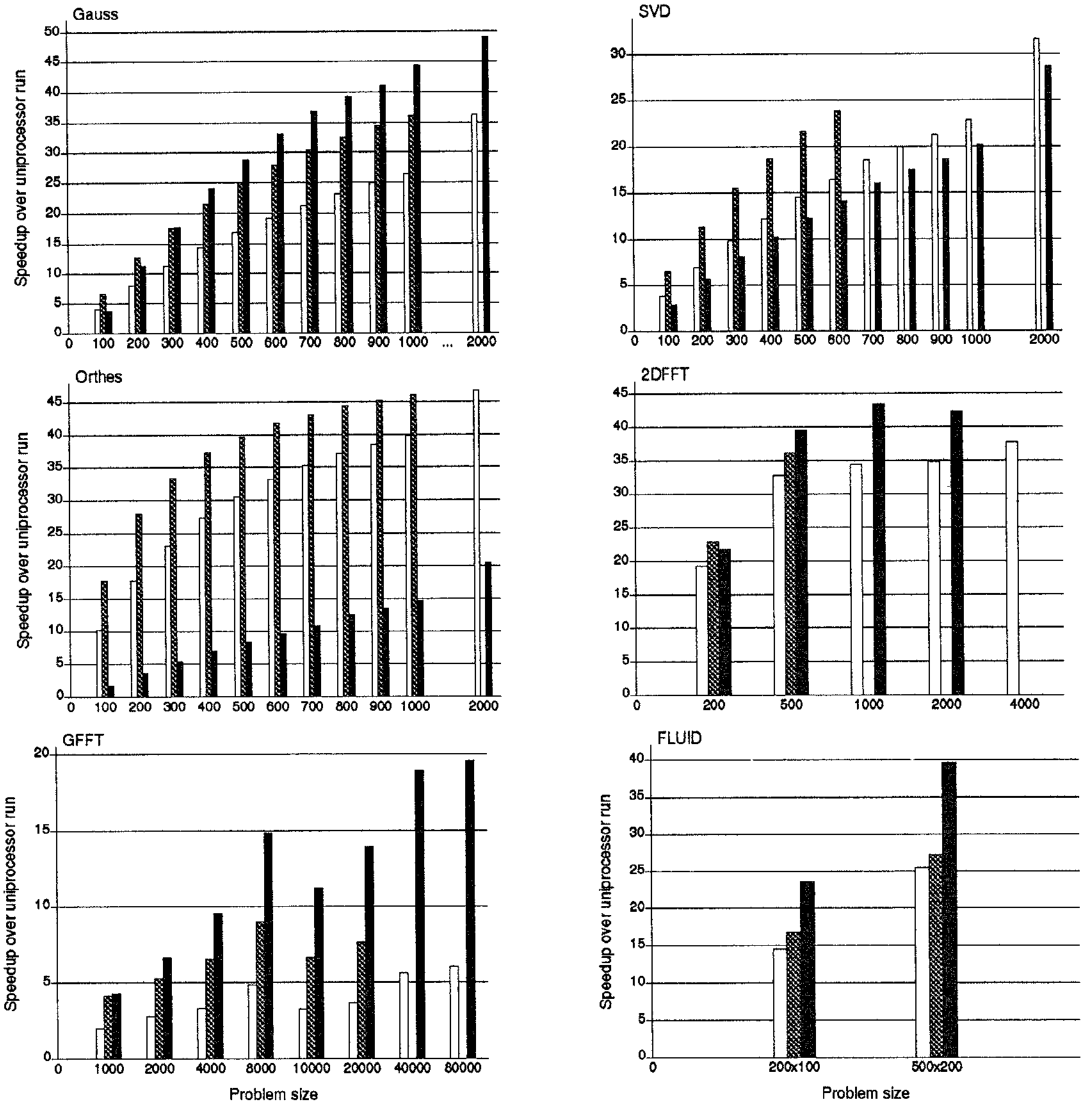

Figure 8: *Speedups of the 6 problems* GAUSS, SVD, ORTHES, 2DFFT, GFFT, *and* FLUID *running in parallel on a Parsytec T800 cluster configured as an* 8 × 8 *torus (white), an* 8 × 8 *iWARP system (grey), and an* 8 × 8 *Fujitsu AP1000 (black).*

on a workstation with more memory (a SUN Sparc Station 1+). Fig. 8 displays Speedups achieved when running the 6 first programs described above on the three systems.

In PGMRES the precision achieved depends on the number of iterations of the algorithm. Communications to fetch and update nonlocal elements of vectors are generated at runtime, at the beginning of the first iteration. All iterations reuse these communications. Therefore speedup depends on the number of iterations and the precision to be achieved. Table 4 summarizes our results for increasing precision of the solution computed. We also present simulated numbers in the right two columns of the table.

SC256 and iWARP

The difference in speedup between the SC256 and iWARP can be justified as mentioned in section 4.1 by the slower relative communication speed of the SC256. To further emphasize the effect of q on speedups, we have benchmarked all seven programs on our architectural simulator for DMPPs, K9. K9 is written in C++ and simulates C or C++ programs at the C statement level. We have used the PE model of K2, based on an AMD 29000 processor with floating point co-processor. We have simulated 7 different architectures which differ only in communication speed, i.e. $q = 0, 0.25, 0.5, 1, 2, 4, 8$. The results were reported in [7]; here, we will only show one example, namely speedup measurements for GFFT as shown in Fig. 9 for the 7 q values and different problem sizes. K9 simulates systolic communication. Therefore we can not immediately compare results achieved on the AP1000 with the simulation results. The iWARP measurements match fairly well with K9 simulations for $q = 2$, the SC256 measurements for $q = 4$. The difference in q is the main reason for the different measures on the two platforms.

precision	Speedups			
	SC256	AP1000	K9, $q=2$	K9, $q=4$
10^{-4}	**3.3**	**5.5**	7.8	4.9
10^{-8}	**3.9**	**6.7**	8.2	5.2
10^{-12}	**4.1**	**7.0**	8.3	5.3

Table 4: *Speedup of PGMRES for increasing precision of the computed solution. The third and fourth column shows measured speedups on the* SC256 *and the AP1000, and the rightmost two columns speedups measured with K9 simulations for* $q = 2$ *and* $q = 4$.

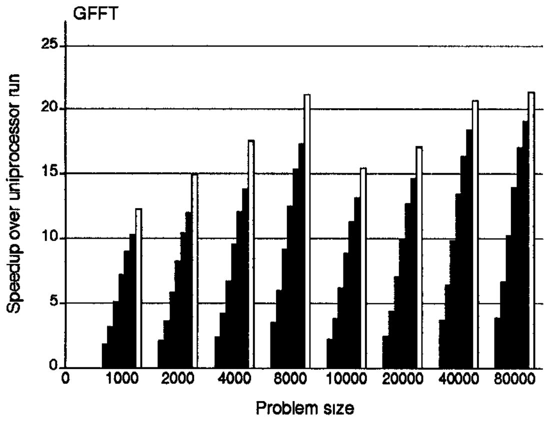

Figure 9: *Simulation of GFFT running in parallel on an* 8×8 *K2 machine; speedups are shown for 4 different problem sizes and 7* q *values, i.e.: (from right to left)* $q = 0, 0.25, 0.5, 1, 2, 4, 8$. .

AP1000

The communication mechanism built in the AP1000 system is different than that implemented in K9. The AP1000 uses memory communication instead of the systolic communication used in K9, the SC256, and iWARP. Therefore not all results achieved on the AP1000 in Fig. 8 can be immediately compared to K9 measurements. The speedups in 2DFFT, GFFT, FLUID, and PGMRES can be explained by the smaller q ratio of the AP1000. At least for medium and large problem sizes communication latency is negligible, because messages are packed into large enough chunks. Speedups in the GFFT case for instance are higher than what was measured on iWARP, because as shown in Table 3, $q \leq 1$ even if all data are in cache.

The three linear algebra problems use explicit communication directives, i.e. column broadcasting in both GAUSS and SVD, and global reduction operations in both ORTHES and SVD. Global reduction operations are much more expensive on the AP1000, than on a machine with systolic communication, as the results for ORTHES show. For large problem sizes vector broadcasting is quite efficient on the AP1000 as the results for GAUSS show, because such operation is supported by the routing controller.

Absolute measures

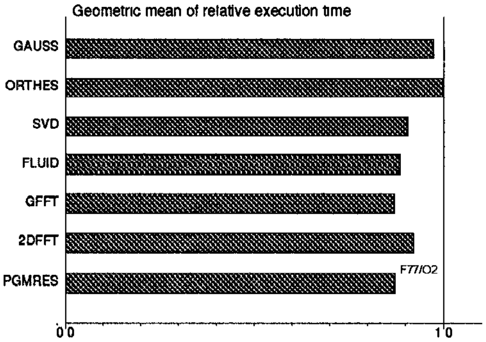

Figure 10: *Execution times of the 6 programs compiled with SUN f77 relative to the execution time of the same programs compiled with Oxygen for a one processor SUN Sparc Station 1+ workstation. For each program we show geometric averages for different problem sizes.*

So far we have only shown speedup figures to evaluate the performance of parallel programs generated by Oxygen. The parallel Fortran sources were constructed using the serial sources and adding compiler directives. Both the execution times of serial and parallel programs are affected by inefficiencies of Oxygen's code generator. Although the overall performance of both parallel and serial programs will gain when adding optimizations to the code generator, speedups will decrease, if those optimizations do not affect the parallelization overhead (e.g. communications and idle times). To quantify inefficiencies of Oxygen's code generator, we have used Oxygen to generate serial programs for a SUN Sparc Station 1+ and compared the execution times to the performance achievable with the SUN `f77` compiler under SUNOS 4.1.1. Fig. 10 shows the results. Serial code generated by Oxygen is between 0.2 and 13.0 % slower than code generated by `f77`.

6 Conclusions

The work on Oxygen has so far provided original contributions regarding the following aspects:

- Remote *updates* of non-LOCAL variables.
- Transparent (partly dynamic) generation of systolic communication primitives *and routes* (for the SC256and iWARP), and of message passing primitives (for the AP1000). Because of the underlying machine model, Oxygen manages the entire communication mechanism, i.e., fetching and updating of data shared between two PEs and (on the SC256and iWARP) forwarding between symbol-owning and symbol-requesting PE. Communication statements need no longer be "hardwired" in the Fortran source.
- General solution to the problem of generating a symbol-handler. Oxygen generates a correct symbol-handler even for P_BLOCKs in which the control-flow depends on non-LOCAL variables. This control-flow dependence may be arbitrarily nested. In related projects ([13, 15]) accesses to non-LOCAL variables should not be control-flow dependent on non-LOCAL variables.
- MULTICOPY symbols. Oxygen provides a mechanism to share data among PEs with dynamically defined ownerships. Without this mechanism, programs like

PGMRES—that can be parallelized on shared memory multiprocessors—could not be executed efficiently.

The results have shown that a DMPP compiler that includes above four features can generate parallel code of satisfactory performance, even for codes more complicated than those considered in the literature presented in section 2. The communication channels of the target DMPP should feature a neighbor to neighbor communication bandwidth comparable to the local computation speed. As in [7] we have measured acceptable performance for $q \leq 4$ on platforms supporting systolic communication. For the relatively high latency memory communication architecture of the AP1000, q was an important parameter to estimate the performance of most of our benchmark programs. On this platform, two of the seven programs could have benefited from hardware supporting global scalar reduction operations.

Acknowledgements

This project would not have been possible without the continuous support of W. Fichtner and M. Annaratone. We thank D. Haupt from RWTH in Aachen, T. Gross from Carnegie Mellon University, and T. Horie from Fujitsu Laboratories LTD, for kindly supporting our work on the SC256, iWARP, and AP1000 systems, respectively. The comments of the anonymous reviewers helped us to improve the quality of the paper.

References

[1] M Annaratone, M. Fillo, M. Halbherr, R Rühl, P. Steiner, and M. Viredaz. The K2 distributed memory parallel processor: Architecture, compiler, and operating system. In *Proc. Supercomputing 91*, Albuquerque, November 1991. ACM-IEEE.

[2] P Beadle, C. Pommerell, and M. Annaratone. K9: A simulator of distributed-memory parallel processors. In *Proc. Supercomputing 89*, Reno, Nevada, November 1989. ACM-IEEE.

[3] Parsytec. Megaframe – SuperCluster series, technical overview. Technical report, Aachen, 1988

[4] S Borkar et al. Supporting systolic and memory communication in iWarp. In *Proc. 17th Symposium on Computer Architecture*, pages 70–81, Seattle, May 1990. ACM-IEEE

[5] H. Ishihata et al. An architecture of highly parallel computer AP1000. In *Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 13–16. IEEE, May 1991.

[6] R. Rühl and M. Annaratone. A parallelizing compiler for distributed memory parallel processors. Technical report, Swiss Federal Institute of Technology Zurich, Integrated Systems Laboratory, 1991. Also to appear in IEEE Trans. on Par and Dist Syst.

[7] M Annaratone and R. Rühl. Balancing interprocessor communication and computation on torus-connected multicomputers running compiler-parallelized code. In *Proc. SHPCC 92*, Williamsburg VA, March 1992. IEEE

[8] C. Polychronopoulos et al. Parafrase-II: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. Technical report, University of Illinois, CSRD, 1989.

[9] M. Burke et al. Automatic discovery of parallelism: a tool and an experiment. In *PPEALS*, pages 77–84, New Haven, July 1988 ACM/SIGPLAN.

[10] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR90-149, Rice University, February 1991.

[11] D. Callahan and K. Kennedy Compiling programs for distributed memory multiprocessors. *The Journal of Supercomputing*, Vol 2:151–169, 1988.

[12] M. Gerndt. Array distribution in SUPERB In *Proc. of the third International Conference on Supercomputing*, pages 164–174, Crete, Greece, June 1989. ACM SIGPLAN.

[13] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared memory data structures on distributed memory architectures. In *Second Symposium on Principles and Practice of Parallel Programming PPoPP*, page 177, Seattle, Washington, March 1990 ACM SIGPLAN.

[14] A. Rogers and K Pingali. Process decomposition through locality of reference. In *Conf. Programming Language Design Implementation*, pages 69–80, Portland, OR, June 1989 ACM-SIGPLAN

[15] J. Saltz, K. Crowley, R Mirchandaney, and H. Berryman. Runtime parallelization and execution of loops on message passing machines *J. on Par. and Dist. Comp.*, April 1990. Also available as ICASE Rep. 89-7, Ja. 1989.

[16] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE transactions on parallel and distributed systems*, 2(3):361–375, July 1991.

[17] M Gupta and P. Banerjee. Compile-time estimation of communication costs in multicomputers. In *IPPS'92*, Beverly Hills, California, March 1992. ACM-IEEE.

[18] V. Balasundaram, G. Fox, K. Kennedy, and U Kremer. A static performance esitimator to guide data partitioning decisions. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP*, SIGPLAN Notices, 26(7):213–223, April 1991.

[19] T Gross and M Lam. Compilation for a high-performance systolic array. In *Proc. SIGPLAN 86 Symp. Compiler Construction*, pages 27–38 ACM SIGPLAN, June 1986

[20] P. S Tseng, M Lam, and H. T Kung. The domain parallel computation model on Warp In *Proc. SPIE*. SPIE, 1988.

[21] P.S Tseng. *A parallelizing compiler for distributed memory parallel computers.* PhD thesis, Carnegie Mellon University – School of Computer Science, May 1989 (also available as technical report number CMU-CS-89-148).

[22] P S. Tseng. Compiling programs for a linear systolic array. In *Conference on Programming Language Design and Implementation*, page 311. ACM SIGPLAN 90, June 1990.

[23] W J Dally Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans. on Computers*, 39(6):775–785, June 1990.

[24] S. L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4 133–172, 1987

[25] M Neeracher and R Rühl. Automatic Parallelization of LINPACK Routines on Distributed Memory Parallel Processors. Technical Report 92/7, Swiss Federal Institute of Technology Zurich, Integrated Systems Laboratory, 1992

[26] R. Rühl. Oxygen—A short tutorial Technical Report 91/9, Swiss Federal Institute of Technology Zurich, Integrated Systems Laboratory, 1991.

[27] C. L. Lawson, R J Hanson, D R Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage *ACM Transactions on Math. Softw.*, 5(3):308–323, September 1979.

[28] R. S. Martin and J H Wilkinson. Similarity reduction of a general matrix to Hessenberg form. In J H Wilkinson and C. Reinsch, editors, *Handbook for Automatic Computation. Vol. 2 (Linear Algebra)*, pages 339–358. Springer-Verlag, New York, 1971.

[29] B.S. Garbow, J M. Boyle, J.J. Dongarra, and C.B. Moler *Matrix Eigensystem Routines - EISPACK guide Extension.* Lecture notes in Computer Science Springer-Verlag, 1977.

[30] G Heiser, C. Pommerell, J. Weis, and W. Fichtner. Three dimensional numerical semiconductor device simulation: Algorithms, architectures, results. *IEEE Trans. on CAD*, 1991.

[31] C Pommerell, M. Annaratone, and W. Fichtner. A set of new mapping and coloring heuristics for distributed-memory parallel processors. *SIAM J. on Sci. and Stat. Comp.*, 1991.

[32] Y. Saad. SPARSKIT A basic tool kit for sparse matrix computation Technical Report CSRD Report no. 1029, University of Illinois, CSRD, August 1990.

[33] Y. Saad. Krylov subspace methods on supercomputers. *SIAM J. Sci Stat. Computing*, 10(6):1200–1232, November 1989