AT&T
Bell Laboratories

Numerical Analysis Manuscript 93–12

Compiler Assisted Distributed Memory Parallelization of an Iterative Solver for Irregular Sparse Linear Systems

Claude Pommerell
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA,
pommy@research.att.com

Roland Rühl
CSCS–ETH, Swiss Scientific Computing Center, CH-6928 Manno, Switzerland,
ruehl@cscs.ch

April 29, 1993

Compiler Assisted Distributed Memory Parallelization of an Iterative Solver for Irregular Sparse Linear Systems

Claude Pommerell and **Roland Rühl**

AT&T Bell Laboratories
600 Mountain Avenue, Room 2C-548A
Murray Hill, NJ 07974-0636
USA
pommy@research.att.com

CSCS–ETH
Swiss Scientific Computing Center
CH-6928 Manno
Switzerland
ruehl@cscs.ch

April 29, 1993

Abstract

Distributed memory parallel processors (DMPPs) can deliver high peak performance comparable or higher than vector supercomputers while promising a better cost-performance ratio. Programming, however, is significantly harder than on traditional vector systems, especially when problems necessitating irregularly structured solution methods are considered. A class of such applications, with large resource requirements, is the numerical solution of highly non-linear partial differential equations on irregularly refined three-dimensional finite-element discretizations. Porting such an application from vector (and shared memory parallel) machines to DMPPs involves some fundamental algorithm changes (for instance, grid decomposition, mapping, and coloring strategies). In addition, no standardized language interface is available to ease the efficient parallelization and porting among DMPPs and between vector computers and DMPPs.

We have ported PILS—an existing package for the iterative solution of large irregularly sparse linear systems of equations on vector computers—to DMPPs, using the parallelizing Fortran compiler Oxygen. Two DMPPs, namely Thinking Machines' CM5 and Fujitsu's AP1000, were used to evaluate the performance of the generated parallel program quantitatively. The results indicate how an application should be designed to be portable among supercomputers of different architecture. We describe several language and architectural features that are essential for such a porting process and ease the parallelization of similar applications drastically.

1 Introduction

Although the first distributed memory parallel processors (DMPPs) were built more than 20 years ago [1], recent development in VLSI technology has made DMPPs attractive alternatives to vector supercomputers. These architectures offer high peak performance for relatively low investment in computer hardware: powerful (and cheap) microprocessors can be used to build a DMPP processing element (PE) with only few integrated circuits. By connecting many such PEs together in a communication network, it is easy to achieve at least the same peak performance as featured by today's most powerful vector supercomputers.

Unfortunately, DMPPs are still much less comfortable to program. Programming comfort can not be quantified as precisely as peak performance, but it is clear that more user efforts are required to implement efficient and portable programs, especially if not only portability among DMPPs is considered, but also portability between DMPPs and vector supercomputers. While early research in scientific parallel processing concentrated on isolated parallel algorithms and specialized architectures, the goal of many current projects is to mitigate this lack of general programmability, either through new software technology (new parallel algorithms, operating systems, and parallelizing compilers), or through new hardware designs, in particular the implementation of more powerful DMPP interconnection networks.

The distinction between the two classes of supercomputers is far less rigid then it may appear from the above. Pipelined vector operations and parallel computations on distributed memory are two orthogonal architectural concepts to exploit parallelism. Distributed memory parallel vector processors or superpipelined processors with large caches on a shared-memory bus are examples of hybrid combinations featured by commercial supercomputers available today. If an application proves to be portable from single vector processors to distributed memory scalar parallel processors, one can safely assume that it will also achieve reasonable efficiency on a supercomputer of hybrid architecture. For the sake of clarity, we distinguish between single vector processors (which we call vector computers) and distributed memory scalar parallel processors (which we call DMPPs), keeping in mind all variations in between.

The interest of the research community in DMPPs has led most major vector supercomputer vendors to develop and announce new supercomputers that are based on distributed memory architectures. However, for many applications, DMPP operating systems and compilers are still incapable of assisting the user in obtaining a reasonable fraction of the machine's peak performance, in a way comparable to what can be expected on vector computers. Closing this gap faces many unsolved problems. Among others, the following three questions have yet to be answered:

- How difficult is the migration of existing vector supercomputer application codes to DMPPs ? By what techniques can parallelism expressed for vector operations be translated into parallel computations on data structures in distributed memories ?

- How much can the system software (operating system and compiler) ease the above migration ? How efficiently could a standardized software interface (for instance, a programming language like High Performance Fortran (HPF) [2]) support the implementation of portable scientific applications, portable not only among DMPPs but among all kind of high performance platforms, including traditional vector supercomputers ?

- How should large scale applications be designed to achieve high-performance both on vector and parallel supercomputers in a portable way ?

It is easier to answer the questions above for applications requiring only highly structured computations, like dense matrix linear algebra, or finite element and difference methods applied to regular grids. In fact several research groups have defined Fortran-like languages and implemented compilers to allow portable coding of these applications. The definition of HPF is a try to standardize these efforts. The portability on a whole spectrum of architectures has been proven for some of these languages (see for instance [3]).

In this paper, we are concerned with more irregular computations, like linear algebra on general sparse matrices, or finite element or finite volume methods applied to irregularly refined grids. For this application class, static compilation is insufficient, and some software support must be provided at run-time. Standard compile-time dependence analysis techniques fail when control-flow depends on run-time information, which is typical for this type of application. If we want to avoid "manual" intervention in the application code, run-time support should either be provided by the operating system or by the compiler's run-time library. The three problems above have been discussed by only few research groups [4, 5].

Strictly speaking, most applications do not rigorously require such irregularity in the solution method. Almost any irregularly structured solution method to a given scientific problem could be substituted by a more resource intensive regularly structured, but more easily parallelizable method. With many real-world problems, however, using a less powerful method induces a severe loss that cannot be offset even by massive parallelism. For instance, if the grid density requirements in a 3-D domain vary by one order of magnitude, the number of redundant grid points when discretizing with a (regular) tensor-product grid with non-uniform spacing can be as much as three orders of magnitude higher than the number of grid points in an irregular finite-element grid. While there are many regularly and irregularly structured options to solve a particular problem, the only *efficient* solution method may require irregular computations. Even though parallelization speedup is generally higher for regularly structured computations, the irregular method often stays the most efficient approach. Many publications reporting close to peak performance on some problem use in fact outdated regularly structured methods, and it would be faster to solve the same problem with a state-of-the-art irregularly structured method running at workstation performance.

This paper tries to answer the three questions above for the existing vector supercomputer code PILS [6], a software **P**ackage of **I**terative **L**inear **S**olvers targeted to the solution of very large, irregularly sparse, unsymmetric, ill-conditioned systems of linear equations. The package has been used in several finite element based applications, including several semiconductor device simulators. We have parallelized PILS using the compiler Oxygen [7], a parallelizing Fortran compiler developed as part of the K2 project [8]. The performance of the generated parallel program is evaluated on two DMPP systems, namely the CM5 from Thinking Machines [9] and the AP1000 from Fujitsu [10].

The paper is organized as follows. After a more precise definition of the project goal, we will introduce the hardware platforms used, and the two software packages PILS and Oxygen. We will then discuss additions to PILS that enable an efficient parallelization with Oxygen, and conclude with a summary of our measurements on both the CM5 and the AP1000.

2 Goal of this project

This project is a step towards the full parallelization of existing applications that are based on large and irregularly structured two- and three-dimensional finite element discretizations. Our target applications are already portable on a range of sequential and vector supercomputers, and we want to extend this same portability to DMPPs and among DMPPs.

A fully automatic parallelization of such a code would be impossible, just like today's highly mature vectorizing compilers are incapable of achieving high performance on a naive sequential implementation of such an application. Instead, carefully designed data structures together with additional hints to the compiler (usually in the form of directives) are necessary to exploit the architectural features of vector computers. We try to exploit the same restructuring effort in a highly vectorized application and substitute a parallelizing compiler for DMPPs, together with some additional tuning. Our self-imposed restrictions include the following:

- Modifications to the original application are kept to a minimum, so that it stays fully compatible and compilable on vector computers. Except for the performance gain, the user should not be able to distinguish whether the application runs sequentially on the host or in parallel on the DMPP.
- Most communications are generated transparently by the parallelizing compiler which hides the underlying message passing architecture as far as possible.

Our project should give an indication on how to design portable finite element based applications for DMPPs and how to support such applications within the language extensions of a parallelizing compiler. In addition, it should highlight features of the underlying machine that work in favor or against the efficient parallelization of such applications.

Many recent studies investigate DMPP implementations of iterative solvers for linear systems. It is important to emphasize that almost all of these publications aim at a much simpler goal. Most authors consider only regularly structured problems, like 5-point finite difference discretizations of simple partial differential equations. The resulting matrices do not need a general sparsity data structure, all operations on them can be treated much more efficiently as addition, scaling, and positional shifts on smaller dense matrices. Furthermore, most authors do not consider preconditioning, except possibly for diagonal scaling. Good preconditioners often increase the convergence speed of an iterative method more drastically than parallelization, and many real-world ill-conditioned problems just cannot be solved by unpreconditioned iterative methods. On the other hand, good preconditioners, like approximate factorizations, are more difficult to parallelize than any other operation in iterative solvers. Finally, many authors consider only symmetric positive-definite systems, for which Conjugate Gradients, but also weaker methods like Symmetric Successive Over-Relaxation (SSOR) or Jacobi iteration converge well. Despite significant new developments in the last few years, no single unsymmetric variant of Conjugate Gradients can be seen as universally successful on ill-conditioned unsymmetric linear systems.

3 Background

3.1 Target platforms

DMPPs can be classified according to the architecture of the local PE, the network architecture, and the communication paradigm. Both the CM5 and the AP1000 feature local SPARC processors. Results of this study can also be applied to DMPPs featuring other RISC processors, but using the same processor on both machines eases a comparison of the two communication architectures. Although the network topologies of the two machines are different (a *fat tree* in the CM5 and a torus in the AP1000) both machines can emulate a two-dimensional communication network efficiently. In *systolic* DMPPs, PEs can only communicate to their nearest neighbors in the network. In *message-passing* DMPPs, PEs can communicate to any processor in the system; a routing controller computes routes between message senders and receivers, transparently to the user. Message passing provides more comfort to the user at the expense of higher communication startup latencies. The CM5 and the AP1000 are typical message passing systems.

As will be explained in more detail in Section 4 we use extensively the host/node programming paradigm. Both machines have a similar host: the CM5 a SUN4/75 and the AP1000 a Sun SPARCsystem 300. The host features full UNIX support (SUNOS 4.1.2 on the CM5 and SUNOS 4.1.3 on the AP1000), while the PEs are only operated with a microkernel. All PE Input/Output is performed via explicit host/PE communications.

3.1.1 Thinking Machines CM5

The CM5 can be configured with up to 16K PEs. Each PE is a SPARC processor with 64 Kbytes cache used for instruction and data caching. The SPARC operates at a clock speed of 32 MHz and has four optional vector processors. Thus, each PE is capable of peak 128 MFLOPS. The PEs can be organized into a single partition or multiple partitions. Each partition has a manager which governs the allocation of parallel resources.

The CM5 has two internal networks that support interprocessor communication: a control network and a data network. The control network supports operations that require global communication such as global reduction operations and global processor synchronizations. The data network supports point-to-point communication. Its topology is a fat tree as shown in Fig. 1.

Both data and control network have a peak (hardware) bandwidth of 20 Mbytes/s. The data network guarantees a system-wide (hardware) bandwidth of 5 Mbytes/s no matter where the data is being sent in the system. We used Thinking Machines' message passing library (CMMD) for most of our experiments.

The current software release allows access to the vector processors only in data parallel Fortran programs written in Connection Machine Fortran (CMF), through the use of the Thinking Machines' scientific library routines (CMSSL), or by explicitly programming the vector units in assembler. Oxygen generates C programs with calls to communication library routines. Therefore all results presented in this paper refer to the use of the SPARC processor only, without vector units.

For all our measurements we used the CM5 software release `7.2_beta1.1-P4` with the SUN C compiler SC1.0 as back end for Oxygen.

Oxygen assumes that the underlying DMPP network has torus topology. When mapping a torus to the fat tree network featured by the CM5, one has to try to map torus neighbor-to-neighbor communication links into subtrees at the fat tree's lowest level. As shown in Fig. 1 this is done by recursively decomposing the two-dimensional network required by Oxygen.

3.1.2 Fujitsu AP1000

The AP1000 is not a commercial system, but an experimental multiprocessor with 64 to 1024 PEs. A detailed description of the system can be found in [10]. 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 Torus network (T-net) for point-to-point communication between PEs, the Broadcast network (B-net) for host

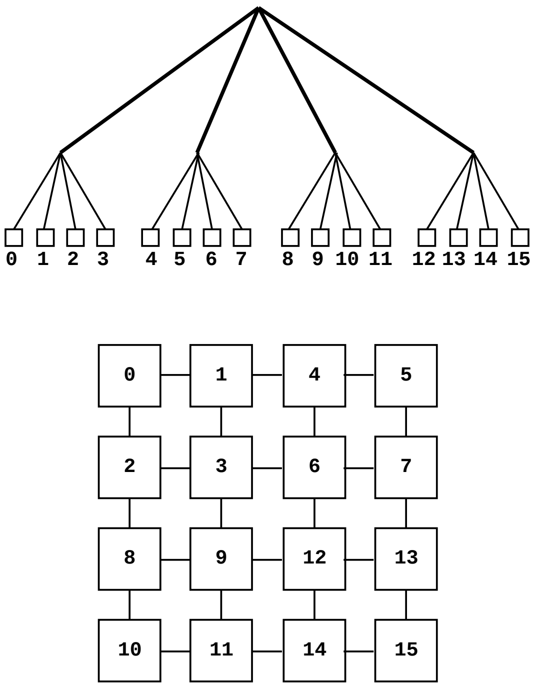

Figure 1: *The fat tree network of the CM5 is mapped onto the two dimensional network required by Oxygen by recursively decomposing the two-dimensional torus network.*

to PE communication, and the Synchronization network (S-net) for barrier synchronization. The T-net implements wormhole routing.

Whenever a user starts execution of a parallel program, the AP1000 PEs are locked for other users. Parallel jobs can be queued in batches.

For all our measurements we used the AP1000 operating system CellOS1.1, with the SUN C compiler SC1.0 as back end for Oxygen.

In contrast to the AP1000, the CM5 allows several users running jobs simultaneously on one system. Memory is shared physically, and parallel jobs are synchronously context switched using the CM5 synchronization network. Timings are in principle not affected since synchronizations are fast.

3.1.3 Basic measurements

We showed in [11, 12] that the speedup of manually parallelized DMPP programs strongly depends on the *communication/computation ratio* T_{io}/T_c, where T_{io} is the time to communicate a double precision floating-point quantity (of size 8 bytes), and T_c is the time the PE takes to perform a double-precision multiply-add operation as in DAXPY [13].

To measure communication performance, we benchmark the loop depicted in Fig. 2. Execution time of neighbor-to-neighbor communications in the (virtual) torus topology is measured for various message sizes. Fig. 3 compares such measurements on CM5 and AP1000. Both machines are configured as 4×8 tori. The native message passing library on the CM5 features high startup

```
char message[MSIZE];
for (j=NRIT; j>0; j--) {
    SEND(RIGHT,message,MSIZE);
    RECEIVE(LEFT,message,MSIZE);
}
```

Figure 2: *The above loop is used to measure communication latency and throughput of the communication networks of the two MPPs.*

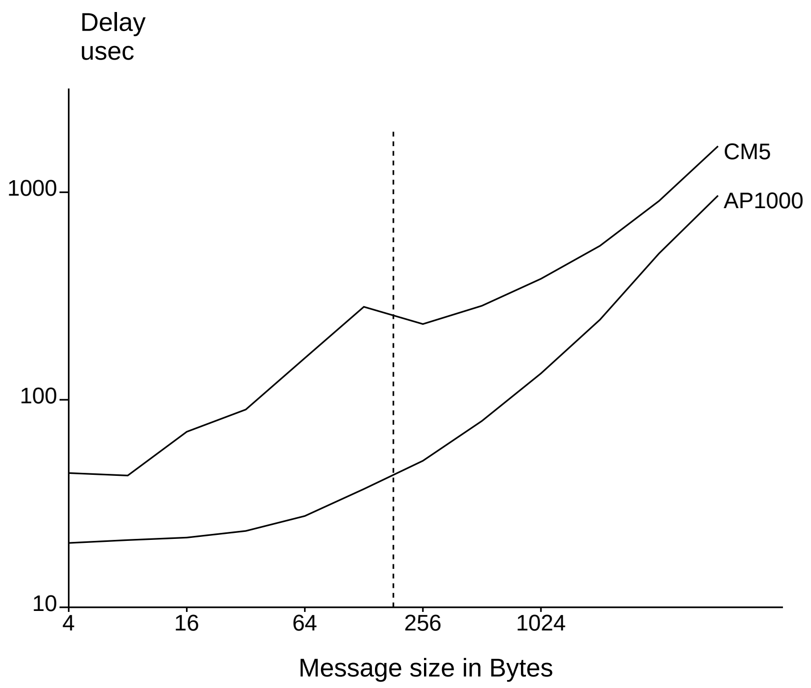

Figure 3: *Comparison of network delay on the CM5 and AP1000. Both systems are configured as* 4×8 *tori.*

latencies when non-blocking sends are used (in the order of 100 μs). In addition, the same library supports the so-called *active message communication paradigm* [14] with much lower communication latencies. In order to reduce startup latencies, we have replaced the native CMMD library calls with our own communication routines (implemented using active messages) for message sizes smaller than 129 bytes. As shown in Fig. 3 startup latency is reduced to about 45 μs.

We have observed an occasional breakdown of communication performance on the CM5, especially when communicating larger messages. To demonstrate this behavior, we have done 6 successive measurements of the communication bandwidth using the loop of Fig. 2. The bandwidth reaches, for large message sizes, a maximum of more than 5 Mbytes/s (which corresponds to the expected network performance when non-blocking sends are used). However, as shown in Fig. 4, occasionally a sudden drop in network bandwidth to less than 1 Mbytes/s occurs.

It is the task of Thinking Machines' hardware and communication library architects to find a

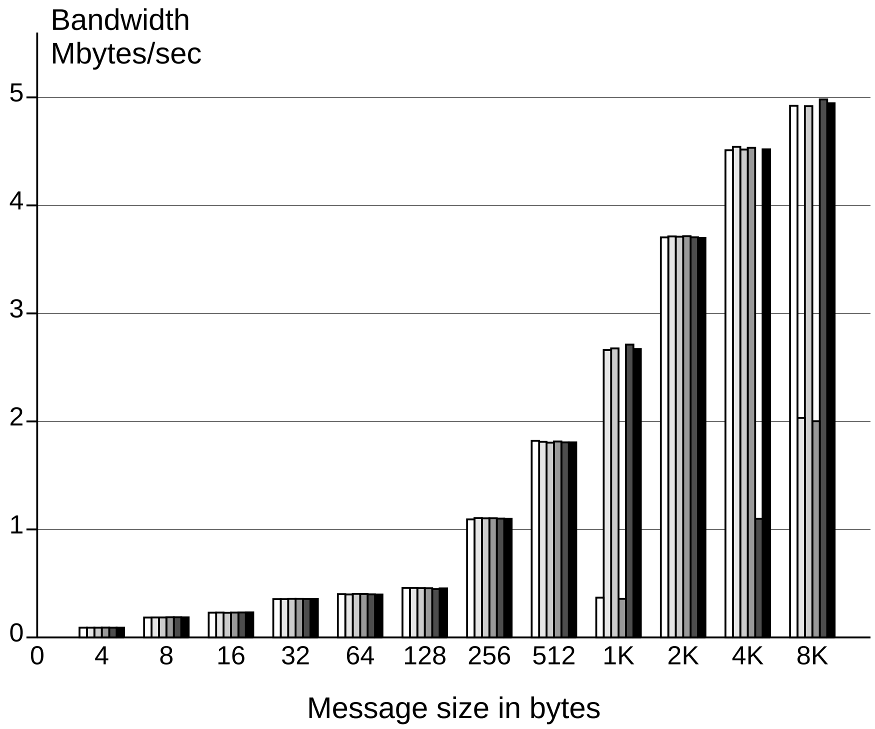

Figure 4: *Six successive measurements of communication bandwidth in a 32 processor CM5 for varying message size. Note the occasional breakdown in performance (in this measurement occurring for message sizes 1 Kbytes, 4 Kbytes, and 8Kbytes).*

sound explanation of this problem. We believe the most probable sources of this performance bug being either due to the time-sharing mechanism or due to network saturation. Note that although we benchmarked the CM5 in time-shared mode, we chose for both the CM5 and the AP1000 times of the day when nobody else was using the system.

The performance of floating point operations is difficult to predict on complex computer architectures, as it depends strongly on the location of the operands in the memory hierarchy. We have therefore measured T_c (the time required to compute a double-precision multiply add operation as in DAXPY, see also Fig. 5) for register operands, in cache operands, and out of cache operands. Table 1 shows the results for the two DMPPs.

3.2 PILS

PILS is a software **P**ackage of **I**terative **L**inear **S**olvers targeted to the solution of very large, irregularly sparse, unsymmetric, ill-conditioned systems of linear equations. It implements a large number of iterative methods, preconditioners, and other variants for iterative solvers, and provides a high degree of flexibility, like automatic adaptation to a more robust preconditioned iterative method whenever the solver that is usually fastest fails to converge. It is integrated into a number of applications, including several semiconductor device simulators, where the solution of hundreds of ill-conditioned linear systems of similar, irregular sparsity structure dominates the overall execution

```
/* scalar computation: */       /* indexed computation: */

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

Figure 5: *The above two loops are used to measure the computation performance of the three MPPs.*

	T_c (μs)				
	CM5	AP1000	SS1+	SS2	SS10
scalar	0.27	1.00	0.76	0.35	0.15
indexed in cache	0.65	1.15	1.93	0.87	0.24
indexed out of cache	0.96	1.91	2.25	1.18	0.58

Table 1: *Measurements of computation speed on the two systems. The time T_c to execute a DAXPY iteration on one processor (i.e. 2 double precision floating point operations) depends on the location of the operands. For comparisons we also include T_c measurements for several SPARC based workstations, namely SUN SparcStations 1+, 2, and 10–41. Serial performance of the AP1000 is close to the performance of a SS1+, while one processor of a CM5 (without vector units) is comparable to a SS2.*

time. PILS and its client applications have been used regularly over two and a half years now, at several dozens of academic and industrial sites.

PILS runs on a number of different platforms, from workstations to supercomputers, including Convex, Cray, Fujitsu, and NEC vector computers. The sparse matrix data structures in PILS are optimized for the use of vector computers.

This section describes those features of PILS that were relevant for our project. Interested readers are referred to [6] for an overview of algorithms and applications, [15] for the embedding in device simulation, [16] for a performance analysis, and [17] for ample detail on algorithms and implementation.

3.2.1 Colored jagged diagonals based on a partitioning by matchings

Nonzero entries of the sparse matrix are stored in so-called "jagged diagonals" [18], a data structure in which matrix-vector products can be computed with a few long vector operations, even if the number of nonzeros per row or column is small. Every operation on a row-oriented jagged diagonal (where row indices are given implicitly by the storage order, and column indices are stored explicitly in an index vector) accesses one of the argument vectors (on the right-hand side of an assignment) through indirect addressing, using a vector gather routine.

Column-oriented jagged diagonal operations update a vector by addressing it indirectly (both on the left- and the right-hand side of an assignment), requiring vector gather and vector scatter routines. To avoid write conflicts in this scatter, the entries in the index vector must be unique. This is obtained through a reordering of the nonzeros resulting from a partitioning by matchings in the associated graph. PILS uses a combination of row- and column-oriented jagged diagonals, to save storage on structurally symmetric parts of the matrix, and to make transposed matrix-vector multiplication as efficient as matrix-vector multiplication without transposition. Transposed matrix-vector products are important in some of the iterative methods, and are inefficient in many other sparse matrix formats.

The rows of the sparse matrix are further symmetrically permuted into blocks, such that the blocks comprising the diagonal are zero everywhere except on the diagonal itself. This allows for a vectorized solution of sparse triangular systems, which is important for fast no-fill incomplete factorization preconditioners. Such a block structure is obtained after coloring the associated graph of the matrix into sets of independent vertices. Every color block of each triangle of the sparse matrix is stored in separate jagged diagonals.

3.2.2 Language usage for portable vectorization

To meet the requirement for the data structuring and flexibility, PILS was mainly implemented in C++. Because of the existence of a translator to C [19], C++ is available on all machines with a C compiler, including all vector supercomputers. Unfortunately, the code generated by vectorizing compilers for C or C++, if available at all, is generally slower than code generated by vectorizing Fortran compilers. For this reason, the most time-critical, vectorizable parts of PILS are written in Fortran. These Fortran parts consist of 46 simple and four nested loops. Although they account for only six percent of the total code size, these loops take over 99 percent of the run-time in a typical application, and are fully vectorized. The compilation system inserts for each machine the appropriate compiler directives, allowing the vectorizing compiler associative rearrangements of global summations, or asserting that index vectors used on the left-hand side of an assignment are unique and safe for scatter operations.

In particular, all linear algebra operations are handled by these critical loops. Addition and scaling of vectors, inner products, copying and zeroing vectors are among the simple Fortran loops[1]. Sparse matrix operations are supported through several more complicated loops that use indexed addressing and deeper nesting.

The C++ part of PILS constructs data structures, interprets specifications to set up the correctly parametrized solution method, and interfaces with the outside world (e.g., a client application), converting external formats into internal data structures if needed. Within the solution process in a preconditioned iterative method, C++ controls the program flow, calls linear algebra routines, and checks convergence through vector norms, but never accesses directly the entries of any of the vectors involved in the iteration. In the computation of sparse matrix-vector products, as well as in the application of no-fill incomplete factorization preconditioners, a C++ program segment guides through the data structure, but the final computation involving the argument and result vectors is performed by Fortran segments.

3.2.3 Typical application

Typical client applications for PILS require the solution of many linear systems in which the matrices have different numerical values, but only one or a few sparsity structures. These matrices arise, e.g., in the discretization of nonlinear partial differential equations (PDEs) into a system

[1] Note that the collection does not consist only of BLAS1 routines [13]. Most critical loops perform combined operations that allow a more efficient usage of caches, vector registers, and chainable pipelines.

of nonlinear equations. Every iteration of a Newton solver for this nonlinear system requires the solution of a linear system of equations. Similarly, the solution of time-dependent PDEs calls for the solution of one or more linear systems at each time step. Semiconductor device simulation, our target application, requires the solution of up to several hundreds or even thousands of linear systems within the modeling of a system of time-dependent non-linear PDEs on irregularly refined two- or three-dimensional finite element grids. For large simulations, the time to solve linear systems accounts for more than 95 percent of the overall execution time.

As the sparsity structure of the matrices is re-used many times, the initialization cost for data structures is amortized over several numerical solver runs. The time used in the inherently sequential algorithms to build vectorizable data structures is negligible in comparison to their benefits.

The linear systems occurring in device simulation can be very ill-conditioned. Occasionally, iterative methods preconditioned by standard fast no-fill incomplete factorization preconditioners fail to converge. In such case, PILS has the capability to adapt to more robust preconditioners, like approximate factorizations with numerical dropping of fill entries based on a threshold tolerance. Unfortunately, these robust preconditioners offer little room for vectorization or parallelism, and run essentially in scalar mode.

3.3 Oxygen

Oxygen [7] is a parallelizing Fortran compiler for distributed memory parallel processors. It accepts Fortran 77 and compiler directives and generates parallel C code with communication primitives for execution on DMPPs. Originally designed as part of the K2 project [8], the compiler was ported to several platforms: the DMPP simulator K9 [20], the transputer based Parsytec SC256, the Intel iWARP and Paragon machines, the Fujitsu AP1000, and Thinking Machines' CM5. Performance of compiler generated parallel programs were reported in [21]. The compiler's benchmark suite consisted of several linear algebra, signal analysis, sorting, and successive over relaxation algorithms. It also included an attempt to parallelize the iterative solver PGMRES included in SPARSKIT [22] which we applied to a finite element problem stemming from semiconductor device simulation.

Oxygen is implemented in C++. It generates DMPP C code with either systolic communication primitives (for iWARP and transputer based machines) or with message passing primitives (for the Paragon, AP1000, and CM5). The generated parallel code is supported by a run-time library implemented in C++.

Several compilation systems with similar (or more) functionality than Oxygen have been developed recently. Systems, such as for instance the CRYSTAL [23], Fortran D [24], KALI [25], PARADIGM [26], and SUPERB [27] compilers, provide on the one hand programming comfort and ease program portability on DMPPs; on the other hand they target automatic parallelization of existing "dusty deck" sequential Fortran programs. As a result of those efforts the High Performance Fortran (HPF) standard has evolved [2], which essentially defines a set of Fortran annotations to distribute data and parallelize loops on DMPPs. In all the above systems, distributed data can be accessed through what is commonly referred to a *global name space*. That is, any processor's reference to an element of a distributed data structure is handled by the compiler transparently to the user; the compiler inserts communication primitives in the generated DMPP program to allow such remote data references.

For this study, Oxygen serves only as a tool to ease the parallelization of PILS, without making use of its automatic parallelization capabilities. Although efficient automatic parallelization of simple programs, for instance dense linear algebra codes such as LINPACK [28], is possible, it is well known that strong user interaction is required to parallelize PILS-like codes even on simpler architectures such as shared-memory parallel computers.

In contrast to HPF and the other above mentioned compilation systems, Oxygen provides the following features which support parallelization of irregularly structured problems:

- A global name space is supported at run-time through a mechanism called *data consistency analysis*. References to distributed variables can be arbitrarily nested. HPF's global name space is resolved at compile-time.
- Oxygen allows fetches and *remote updates* of distributed data. HPF is based on the "owner-computes rule", i.e. only the processor which allocates a data item is allowed to modify it.
- Oxygen generated parallel programs do not use collective communication routines, which would imply at least one global synchronization at the end of all parallel loops. Instead, processors synchronize implicitly by using local communications (non-blocking sends and blocking receives) as required by data exchanges.
- The sequential part of the program execution is duplicated by Oxygen on all processors. The other above mentioned systems compile the sequential part of a parallel program into code to be executed on a single processor or the host of the DMPP.

3.3.1 Programming paradigm

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 a global name space. In regularly structured computations the global name space can be enforced at compile time, but to be able to parallelize irregular codes like PILS, run-time support is required. We call the process of determining ownership in a DMPP at 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.

Oxygen decomposes a Fortran program 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 may or may not run in parallel on the PEs, but in either case their computation is local and does not activate implicit interprocessor communication. Public blocks always run in parallel and activate interprocessor communication because they operate on data structures allocated across PE boundaries.

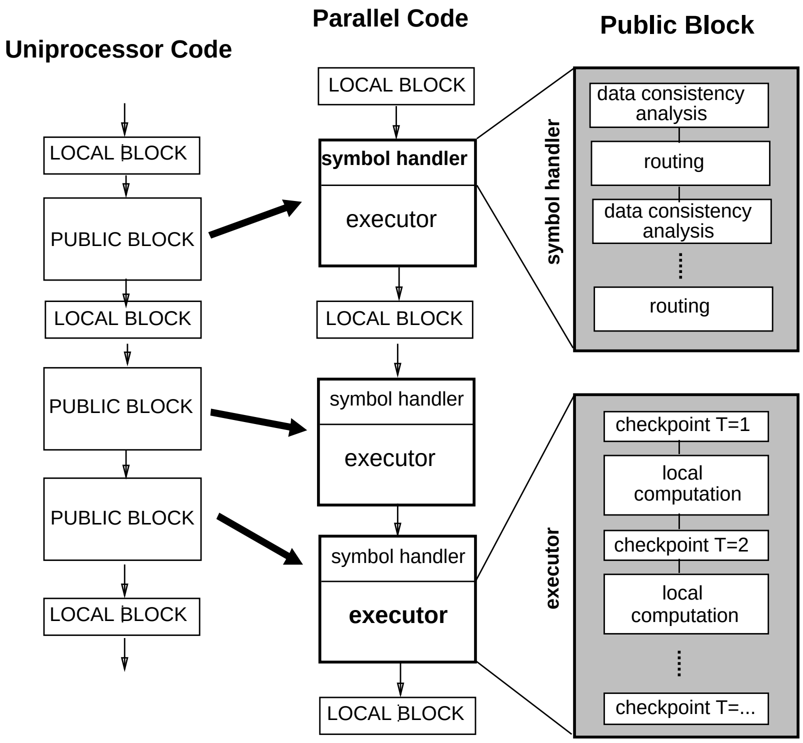

Figure 6: *The model of uniprocessor and parallelized code embedded in Oxygen. The uniprocessor code is decomposed, via compiler directives, into local and public blocks.*

The model of the program structure embedded in Oxygen is shown in Fig. 6. The leftmost column shows the uniprocessor code and the center column shows the structure of the source code—common to all PEs—generated by Oxygen. Every public 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 alternating data consistency analysis and routing phases to interrogate PEs about ownership of shared variables. The executor is a sequence of computation phases with communication checkpoints in between. Data consistency analysis in the symbol handler constructs the data structures that will be used at the communication checkpoints to execute interprocessor communication primitives. Note that on systolic DMPPs the routing phases must establish communication on all PEs placed on a given route between data accessing and owning PE. On message passing DMPPs the owner is only notified during the symbol-handler's execution of a remote data access.

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 These are only known and may only be accessed by their owner.

SINGLECOPY data Ownership is defined as part of a directive, remains the same throughout the lifetime of the program, and refers to one PE. All other PEs may access and update the data through interprocessor communication. A SINGLECOPY array may be distributed and its elements may be owned by different PEs. Typically, large arrays in a Fortran program will be declared SINGLECOPY, together with the specification of a directive to define the distribution i.e. partitioning and mapping of the array's index domain. Distributions are mostly given statically (for instance block-wise or cyclic partitioning). For the parallelization of PILS, however, it is important to be also able to define irregular SINGLECOPY distributions at run-time. That is, a distribution is initialized once at the beginning of program execution before the actual parallel computation starts, but after some run-time information important for an efficient distribution has become available.

MULTICOPY data Ownership changes dynamically. A PE becomes the owner when it either updates or fetches such data, and no update of the same item is executed by other PEs at that time. If two fetches with no intervening update occur, an item may have multiple owners. PEs that are not owners of an item can access its latest update through interprocessor communication. The run-time computation of ownership for MULTICOPY data is expensive both in terms of execution time and in terms of memory requirements. Therefore, this qualifier should only be used when no information is given on how to efficiently distribute a given data structure. As will be explained in Section 4, we explicitly included partitioning and mapping strategies in PILS and MULTICOPY directives were not required for its parallelization.

Communication primitives are usually generated dynamically when nonlocal data are accessed inside public blocks. Public 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. The directives SUMVER and SUMHOR, for instance, can be used to globally sum up local variables across all PEs of all columns or rows of the underlying torus topology.

4 Porting PILS with Oxygen

If the linear solver is the main time-consuming operation within an application implemented using PILS, we can obtain a notable benefit by parallelizing only PILS, without touching the client application. The applications would fit neither in the limited memory nor in the scarce operating system support of a single PE. The PILS interface to the outside world should therefore run on the host, so that the client application can run on the host and does not see a difference between a sequential or a parallel version of the PILS library.

Along with this, the mostly sequential parts of PILS that construct data structures run preferably on the host. Parts with significant parallelism are already clearly marked in PILS as vectorizable parts. These critical loops, mentioned in Section 3.2.2, are parallelized by Oxygen.

The clean separation of numerically intensive, time-consuming part of PILS coded in Fortran, and the more complex control and interfacing part coded in C++ led us to the following parallelization strategy:

- All C++ code is executed on the DMPP host.
- All Fortran routines are annotated with parallelization directives and compiled by Oxygen into parallel subroutines to be executed on the PEs.
- A remote procedure call interface is used for communication between host and PEs. This interface requires an additional software layer between C++ and Fortran source which determines which parameters to send from host to PEs on subroutine invocation, and which parameters to return at the end of the routine's parallel execution.

We will now describe the parallelization of the numerically intensive time-consuming Fortran source on the PEs, and then explain how the remote procedure call interface was optimized to avoid expensive host/PE communications.

4.1 Parallelizing the Fortran part

The numerically intensive part of PILS consists of linear algebra operations on vectors and sparse matrices. As mentioned in Section 3.2.1, matrices are stored in colored jagged diagonals. The part of this data structure that describes the sparsity structure, consisting mainly of indices for nonzero entries, stays constant while an application is solving many linear systems with the same sparsity structure. The numerical part of the data structure, with one particular instance of values for the nonzero entries, stays constant over the whole iterative algorithm that solves one particular system of linear equations. The average number of nonzeros per row in the matrices ranges typically between five and twenty-two, and there are typically between five and twenty vectors involved in the iteration process. Both matrices and vectors thus have to be distributed for memory and execution time efficiency. During the solution of one linear system, however, only the distributed vectors are write-accessed.

Therefore we have chosen two different mechanisms to distribute the vectors and the matrix. On the PEs vectors are declared as SINGLECOPY Fortran arrays, and remote accesses to vector elements are supported by Oxygen's global name space. The matrix instead, is declared as LOCAL array, since no changes are to be expected once the array elements have been initialized by the host as part of the host/PE interface explained in more detailed in Section 4.4. Both vector and matrix distributions are computed by the host at run-time.

All vectors are declared SINGLECOPY with the same dynamic distribution in all 50 subroutines. A distribution vector specifying ownership of vector elements (i.e. which element is allocated on which processor) is defined in the SINGLECOPY declaration of the vectors. The distribution vector is initialized by the host/PE interface when the solver starts executing.

The same dynamic partitioning and mapping is also used for simple loops. This makes efficient parallelization of many Fortran routines defining vector-vector operations rather simple, as all linear operations on vectors do not require any communication. Since Oxygen does not enforce global synchronization at the end of loops, PEs compute fully asynchronously on these operations. Only in few such routines (for instance, the computation of a dot product), additional directives have to be inserted to efficiently implement global reduction operations. All vector-vector operations are therefore specified as local blocks.

The indirect addressing within sparse matrix-vector products constitutes the major challenge. The same operations that require vector gather routines on a vector computer (see Section 3.2.1) need remote memory fetching on a DMPP. Oxygen handles these transparently by generating appropriate messages for accesses to the global name space. As the indirect addresses are not known at compile-time, the communication pattern can only be generated at run-time, within a public block. As the same patterns are required many times (there are one or two matrix-vector products per iteration, and several tens of iterations to solve a linear system), the executor part of a public block can be re-used, and the more time-consuming symbol handler is only performed once.

Vector computer scatter routines writing to an indexed array correspond to remote updates on a DMPP. These operations can be parallelized efficiently only because Oxygen does *not* restrict computation to the owner of a data item. The generation of remote update messages is just as transparent and efficient as remote fetching.

4.2 Mapping for locality

Any simple-minded assignment of vector entries to processors would degrade performance disastrously, as almost all indirectly addressed entries would result in remote accesses, and every processor would have to communicate with every other processor. The *mapping problem* for finite-element meshes consists of partitioning vectors (and matrix rows and columns) in a way that each processor owns an approximately equal number of entries (for load balancing), a maximum number of accesses are local, a minimum number of other processors are accessed for remote fetching or updating, and these processors are at minimum distance in the communication network (thus reducing transfer latency and network contention). The development of heuristics for this (NP-hard) mapping problem is currently a field of active research, and beyond the scope of a parallelizing compiler, but the possibility to define a dynamic mapping using a distribution list allows us to add problem-specific mapping heuristics to our code.

We selected a two-dimensional geometric mapping heuristic that we had good experience with in an earlier related project [29] and that is well adjusted to the torus-like network topology of the AP1000 and our embedding of a torus in the fat tree of a CM5. The heuristic proceeds as follows on n vertices mapped to a $p_v \times p_h$ torus or mesh of processors: First, vertices are sorted according to their x-coordinate in the physical discretization grid. This sorted list of vertices is partitioned by $(p_v - 1)$ vertical cut lines into p_v sets of n/p_v vertices each, and each of these sets is assigned to one column of the mesh of processors. Then each set is sorted again according to the y-coordinate and partitioned by $(p_h - 1)$ horizontal cut lines into p_h sets, and each of these sets is assigned to one processor of this column. This algorithm has a time complexity of $O(n \log n)$ and can be used in the sequential part on the host, as it does not dominate over other algorithms in the construction of the colored jagged diagonal data structure. Figure 7 shows how the discretization grid of the smallest of our test cases was mapped on 16 processors arranged in a 4×4 mesh.

Recently, new mapping heuristics such as recursive spectral bisection [30] have been shown to reduce the total number of edges that are cut by processor boundaries. We refrain from such techniques because the eigenvalue computations required in these algorithms would clearly dominate over all other operations in the construction of the PILS data structures. The normal procedure

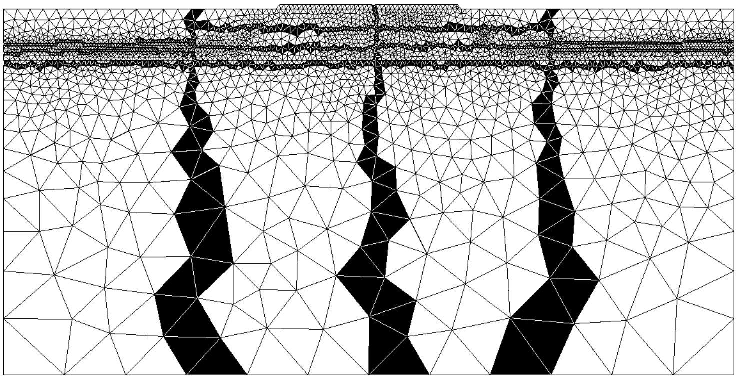

Figure 7: *Two-dimensional geometric mapping of an irregularly refined discretization grid to a* 4×4 *processor interconnection network. All elements (triangles) whose corners are assigned to different processor are drawn in black. White lines crossing black strips correspond to graph edges that require communication; local edges are drawn as black lines.*

would be to move these computations off-line to a separate mapping program whose result is read from file before starting the parallel program. This would conflict too much with our goal to parallelize PILS transparently from its client application.

4.3 Coloring

Incomplete factorization preconditioners rely on the solution of sparse triangular systems of linear equations. In the case of fast preconditioners with no fill (like ILU or its faster variant D-ILU), the sparsity structure of the triangular matrices for the preconditioner coincides with the corresponding triangle of the system matrix.

The solution of triangular systems imposes higher sequentiality than matrix-vector multiplication. In a lower triangular system $Lx = b$, for instance, the i-th entry of the solution vector, x_i, depends on the prior calculation of the previous entries, x_j with $j < i$. Apparently, this imposes a synchronization point (with data exchange through communication) for each row. Sparsity in L reduces the dependencies: x_i depends only on x_j if ℓ_{ij} is nonzero. *Coloring* is a technique to exploit this sparsity in order to construct higher parallelism in triangular system solvers.

Coloring is already used in PILS to vectorize triangular system solvers (see Section 3.2.1), and is integrated into the colored jagged diagonal data structure. The solution of a triangular system with c colors now results in $(c-1)$ self-synchronizing data exchanges, adding up to $(2c-2)$ communication points for a D-ILU preconditioned matrix-vector product. The typical number of colors is around 7 for our 2-D grids and around 12 for our 3-D grids.

Even if perfect load balance in vector-vector operations and unpreconditioned matrix-vector multiplication is achieved through mapping an equal number of vector entries to each processor, the coloring introduces some imbalance within the processing of each single color. The *balanced greedy coloring heuristic* [29] achieves an equal distribution of each color, at the expense of increasing the total number of colors[2].

4.4 The host/PE interface

The host/PE interface replaces Fortran subroutine calls in the C++ PILS source code with a remote procedure call (RPC) interface. It first passes all parameters required for the parallel execution of the subroutine to the PEs, then initiates the start of the corresponding compiler parallelized routine on the PEs, and finally, after the parallelized routine's execution, the interface collects results necessary on the host for further execution of the solver. Parameters to be passed include scalars, vectors, and sparse matrices.

Correct parallel execution of PILS does not require the matrix to be loaded from host to PEs for each parallel matrix-vector operation. As already mentioned above the matrix is only read-accessed and can therefore be loaded only once. Host-to-PE distribution of vectors jeopardizes efficient parallel execution of any vector operation, if read access to a vector during such operation requires the host to distribute the vector to the PEs.

In iterative solvers most time is spent in modifying vectors. Because the PILS source was optimized such that most time-consuming code was implemented in Fortran, most vector accesses take place *in the Fortran code*, i.e. the host very seldomly write-accesses larger parts of any vector during the iterative solver's execution. Most dependences from the Fortran subroutines to the controlling C++ code are due to data items involved in control flow decisions, such as the computation of a convergence criterion.

The above observation allows for heavy optimization of the host-to-PE communication protocol: if indeed a vector is only modified by the Fortran subroutines, it must be distributed to the PEs only once; any future subroutine accessing the same vector will operate on the same data kept resident in the PE memory after the subroutines finish execution.

However, without major changes in the controlling PILS C++ source, it is difficult to decide statically whether any valid vector copy exists on the PEs or whether such a vector should be (re-)initialized by the host/PE interface. We will try to give evidence in Section 5 that such decision can be made efficiently at run-time by the host/PE interface. As part of that interface we have implemented on the PEs a *host/PE cache*, which dynamically allocates and downloads from the host distributed copies of vectors accessed in parallelized Fortran subroutines. Whenever the cache finds a valid PE-memory resident copy of a vector it does not download the vector from the host.

For result vectors and scalars that are used in the host code after they were modified by a parallelized Fortran subroutine, the PEs are forced to invalidate the corresponding entry in the host/PE cache Such a decision is done statically as part of the host/PE interface code.

On some platforms host/PE communications are rather expensive. On the AP1000, for instance, during the host program execution, two UNIX processes run on the host: the host program itself, and a monitoring process which interfaces host program and the AP1000 B-net. Therefore any host/PE communication invokes at least one UNIX process switch between host program and

[2] A compensating reduction of the number of colors by ignoring dependencies between vertices mapped to the same processor, as used with a compressed row sparse matrix data structure in [29], could not be used without modifying the code handling the jagged diagonal sparse matrix data structure.

	LDDH	DR15E	BIPOL3D20KH	BP25E	DR15C	BP25C
Grid dimension	2-D	3-D	3-D	3-D	3-D	3-D
# PDEs	1	1	1	1	3	3
# unknowns	2674	15564	20412	25642	46692	76926
# nonzeros	18614	143710	263920	234436	986042	1618414
matrix density	7.0	9.2	13.0	9.1	21.1	21.0

Table 2: *Problem characteristics.*

monitor, and host-to-PE communication latency escalates to several milliseconds. Because many Fortran subroutine calls are required for a single iteration of the solver (even for moderate problem sizes several hundreds), the host/PE interface has to be optimized even further to reduce PE idle times due to the host/PE communication latency.

The following two strategies are chosen to further optimize the interface:

- Only few parallelized Fortran subroutines actually return data to the host. Therefore the execution of most subroutines can be *pipelined*, which reduces PE idle times due to host-to-PE latency *and* PE idle times due to interprocessor communication. Remember that no global synchronizations are introduced at the end of most parallel loops.

- To reduce protocol overhead between host and PEs, the host/PE cache is also implemented on the host. The host computes itself which data need to be distributed; in parallel the same decision is done on the PEs to compute which data need to be received from the host.

5 Experimental results

5.1 Test environment

Our benchmark suite consists of six test problems, linear systems extracted from real simulations within several semiconductor device simulators. These problems were selected to display a typical variety of problem sizes and complexities within typical large device simulations. Most of our problems are based on three-dimensional discretizations, as two-dimensional device simulation problems are often small enough to be solved more efficiently with sparse direct solvers. Standard device simulation solves the drift-diffusion equations, a set of three partial differential equations (PDEs) in three variables. Depending on the operating mode, the discretized equations are solved separately ("Gummel iteration"), involving one unknown per discretization grid point, or all three PDEs are assembled in a coupled system with three unknowns per grid point. Grid dimensionality and number of PDEs influence the density of the sparse matrix and the connectivity for partitionings as those used in the mapping. Table 2 summarizes the characteristics of our test problems. Matrix density is conveniently expressed by the number of nonzero entries per row.

Unless otherwise noted, all benchmarks were done using Bi-CGSTAB as the iterative solution method, preconditioned by a D-ILU preconditioner in split position. In our experience, this combination of an iterative method and a preconditioner has proven to be fastest in most problems of device simulation. Other methods and preconditioners may achieve higher MFlops performance, but usually require more total execution time to converge to the solution.

	LDDH	DR15E	BIPOL3D20KH	BP25E	DR15C	BP25C
serial execution						
	56	71	80	30	65	102
parallel execution						
CM5 4×8	57	71	81	30	65	95
CM5 8×8	32	71	80	30	75	95
AP1000 4×4	55	71	69	30	65	n.a.
AP1000 4×8	58	71	82	30	65	n.a.
AP1000 8×8	54	71	80	30	65	89
AP1000 8×16	54	71	80	30	65	96

Table 3: *Number of iterations required to achieve precision* 10^{-10}.

5.2 Asynchronous reduction operations and convergence

Iterative solvers are sensitive to rounding errors in the computation of vector dot products. Any parallel implementation of such a reduction operation must modify the summation order with respect to the sequential operation. Since floating-point addition is not associative, even a slight change in the summation order induces a different rounding of the result. In an iterative solver, these rounding variations propagate up to variations in the total number of iterations required to solve a given problem (for very sensitive ill-conditioned problems, such variations may even result in divergence or algorithmic breakdown [17]).

Vector dot products are executed in parallel on a DMPP by summing up globally local dot products on the local partial vectors. Oxygen substitutes tuned library calls for these global summations. Unfortunately, these manufacturer-provided library routines work non-deterministically, so that the actual order of summation may not be exactly the same from one identical run to another. Note that this phenomenon is not limited to DMPPs, we have had similar experiences with various shared-memory multiprocessors.

We have collected in Table 3 the number of iterations required for both serial and parallel solver runs. Due to the different (and even non-deterministic) sequences of summations when computing vector dot products, the required number of iterations to achieve a given precision (here 10^{-10}) with the parallelized solver may be both smaller or larger than for the serial program run. In order to avoid misinterpretations due to this effect, whenever the measurements presented in this section should demonstrate the efficiency of our parallelization strategy, we refer to only one iteration of the iterative solver. Total execution times are shown when different solution methods are compared.

5.3 Execution time and speedups per iteration

The total execution time of PILS can roughly be decomposed into three components: (1) execution time of the compute intensive Fortran code, (2) distribution of the matrix from host to PEs, and (3) overhead of the host/PE RPC interface. The first component is depicted in Table 4; measurements of the second and third component are summarized in Table 5.

Table 4 summarizes execution times and speedups measured for one iteration running on one

	LDDH	DR15E	BIPOL3D20KH	BP25E	DR15C	BP25C
serial execution times (s)						
CM5	0.14	0.76	1.23	1.23	*4.3*	*7.1*
AP1000	0.20	1.46	*2.4*	*2.5*	*8.6*	*14.2*
SparcStation 1+	0.16	1.15	1.89	1.98	6.72	n.a.
SparcStation 2	0.09	0.69	1.13	1.15	4.03	6.68
parallel execution times (s)						
CM5 4×8	0.044	0.126	0.247	0.502	0.552	1.913
CM5 8×8	0.056	0.101	0.165	0.295	0.444	0.976
AP1000 4×4	0.035	0.188	0.326	0.474	1.059	n.a.
AP1000 4×8	0.025	0.104	0.182	0.295	0.571	n.a.
AP1000 8×8	0.019	0.067	0.109	0.176	0.346	0.750
AP1000 8×16	0.017	0.045	0.072	0.127	0.213	0.500
Speedup						
CM5 4×8	3.2	6.0	5.0	2.5	*7.8*	*3.7*
CM5 8×8	2.5	7.5	7.5	4.2	*9.7*	*7.3*
AP1000 4×4	5.7	7.8	*7.4*	*5.3*	*8.1*	n.a.
AP1000 4×8	8.0	14.0	*13.2*	*8.5*	*15.1*	n.a.
AP1000 8×8	10.5	21.8	*22.0*	*14.2*	*24.9*	*18.9*
AP1000 8×16	11.7	32.4	*33.3*	*19.7*	*40.4*	*28.4*

Table 4: *Execution times per iteration in seconds and speedup of the PILS Fortran code. Table entries denoted with n.a. (not available) could not be filled due to the memory requirements of the largest problems. Numbers in italics have been extrapolated using SparcStation 2 measurements.*

CM5 respectively AP1000 processor, on both a SparcStation 1+ and a SparcStation 2, and running in parallel on CM5 and AP1000 systems of different sizes. The parallel execution times refer to the Fortran code only, i.e. no host/PE communication overhead is included. On the one hand, parallel performance depends on the structure of the problem: larger systems like DR15C achieve high speedups, because the relative importance of the communication overhead decreases. On the other hand different communication performance of the underlying DMPP strongly influences speedup measurements. As also shown in Fig. 3, the AP1000 features higher communication performance than the CM5, both in absolute and (more importantly) in relative terms (i.e., the ratio of communication to computation performance is approximately 4 times smaller on the AP1000 than on the CM5, independently of the message size). It is therefore not surprising that the highest speedups could be measured on the AP1000. Note that due to the different communication performance of the two DMPPs, an 8×8 AP1000 system outperforms a CM5 of the same size, although PEs of the AP1000 are less powerful.

The overhead of the host to PE matrix distribution and of the host/PE RPC interface are shown in Table 5. Since the algorithm implemented for the matrix distribution is based on broadcasts, time for the matrix distribution does not depend on the machine size. Consequently only measurements for 8×8 configurations are shown. The RPC overhead is largely independent of both the machine

	LDDH	DR15E	BIPOL3D20KH	BP25E	DR15C	BP25C
	Host execution time (s)					
CM5 4×8	0.110	0.223	0.360	0.598	0.738	2.218
CM5 8×8	0.130	0.183	0.270	0.404	0.625	1.187
AP1000 4×4	0.103	0.258	0.402	0.523	1.160	n.a.
AP1000 4×8	0.100	0.174	0.259	0.349	0.672	n.a.
AP1000 8×8	0.094	0.139	0.187	0.237	0.442	0.844
AP1000 8×16	0.094	0.117	0.151	0.187	0.310	0.589
	Time for matrix distribution (s)					
CM5 8×8	0.331	2.521	4.973	3.176	16.462	21.363
AP1000 8×8	0.557	3.347	5.937	4.270	21.957	28.903
	Host/PE bandwidth for matrix distribution (Mbytes/s)					
CM5 8×8	.190	.203	.195	.262	.228	.288
AP1000 8×8	.113	.152	.164	.195	.171	.213

Table 5: *Average execution time per iteration (measured on the host) and total overhead for the matrix distribution; the distribution overhead is given both as time (in seconds) and as host/PE bandwidth (in Mbytes/s).*

size and the problem size.

Both RPC interface and matrix distribution overhead significantly contribute to the overall execution time. However, at least on the AP1000, we expect a drastic performance improvement of the host/PE interface in the near future [31]. As already explained in Section 4.4, any host/PE communication on the AP1000 traverses a monitoring process and at least two expensive UNIX process switches are required on the host. We have been promised new host/PE communication primitives that avoid such process switches in the AP1000's next system software release.

One reason for the measured differences on the CM5 and the AP1000 is the different communication performance of the two systems. However, to fully explain the results presented in Tables 4 and 5 we have to come back to the occasional drop in CM5 communication performance measured in Section 3.1. Both Tables 4 and 5 show *average* execution times of one iteration of the solver. Table 6 shows how much this average differs from both maximum and minimum execution time of an iteration benchmarked on the host of both an 8×8 CM5 and AP1000 system. While a relatively large maximum can be explained through occasional UNIX system load on the time-shared host, we would expect minimum and average execution time to only hardly differ. The measurements on the AP1000 confirm this expectation. However, on the CM5 the averages are up to twice as large as minimum execution times. We can only explain this phenomenon through the occasional drop in communication performance, which contributes more to the average than to the minimum execution time.

In order to quantify how important low communication startup latencies are for the efficient distributed memory parallel execution of PILS we have depicted in Fig. 8 the total number of PE-to-PE messages communicated in an 8×8 CM5 as a function of the message size, for the complete solution of each of the 6 problems. From this figure it is obvious that very often only

Problem	CM5			AP1000		
	min.	max	av.	min.	max	av.
LDDH	0.12	0.18	0.13	0.08	0.23	0.09
DR15E	0.15	0.67	0.18	0.13	0.27	0.14
BIPOL3D20KH	0.20	0.76	0.27	0.18	0.34	0.19
BP25E	0.33	0.87	0.40	0.23	0.35	0.24
DR15C	0.49	1.07	0.63	0.43	0.65	0.44
BP25C	0.60	2.13	1.19	0.83	0.98	0.84

Table 6: *Minimum, maximum, and average of execution time of one iteration measured on the host (in seconds) of the two DMPPs configured as* 8×8 *tori. Note the large difference between average and minimum on the CM5, whereas minimum and average on the AP1000 hardly differ. Note also, that in many cases the minimum iteration time measured on the CM5 host is smaller than the average execution time of the PILS Fortran code measured on the PEs.*

Problem	with sync.	without sync.
LDDH	0.021	0.017
DR15E	0.053	0.045
BIPOL3D20KH	0.088	0.072
BP25E	0.149	0.127
DR15C	0.235	0.213
BP25C	0.551	0.500

Table 7: *PE execution times per iteration with and without global synchronizations introduced at every checkpoint of the parallelized program running on an* 8×16 *AP1000.*

small messages are communicated. Since the compiler is able to optimally pack messages between communication checkpoints, this distribution only depends on the partitioning strategy and the resulting data dependencies.

5.4 Introducing global synchronizations

All HPF–like compilation systems mentioned in Section 3.3 generate collective communication statements at the end of each parallel loop. Therefore at any such checkpoint an implicit synchronization of all PEs takes place. When an Oxygen generated parallel program is executed, only PEs that need to exchange data implicitly synchronize through the use of blocking receive statements. Note that this feature also requires the sequential code being duplicated on all PEs; HPF–like compilation systems restrict the execution of serial code to one single PE. To demonstrate what impact global synchronizations have on the overall performance we compare in Table 7 PE execution times per iteration on the one hand running without synchronizations on an 8×16 AP1000, and on the other hand running the same code with an additional synchronization at every checkpoint. Note that the AP1000 features very fast global synchronizations (about 5 μs latency). The approximately 10 % loss in performance shown in Table 7 are not due to the latency of a synchronization

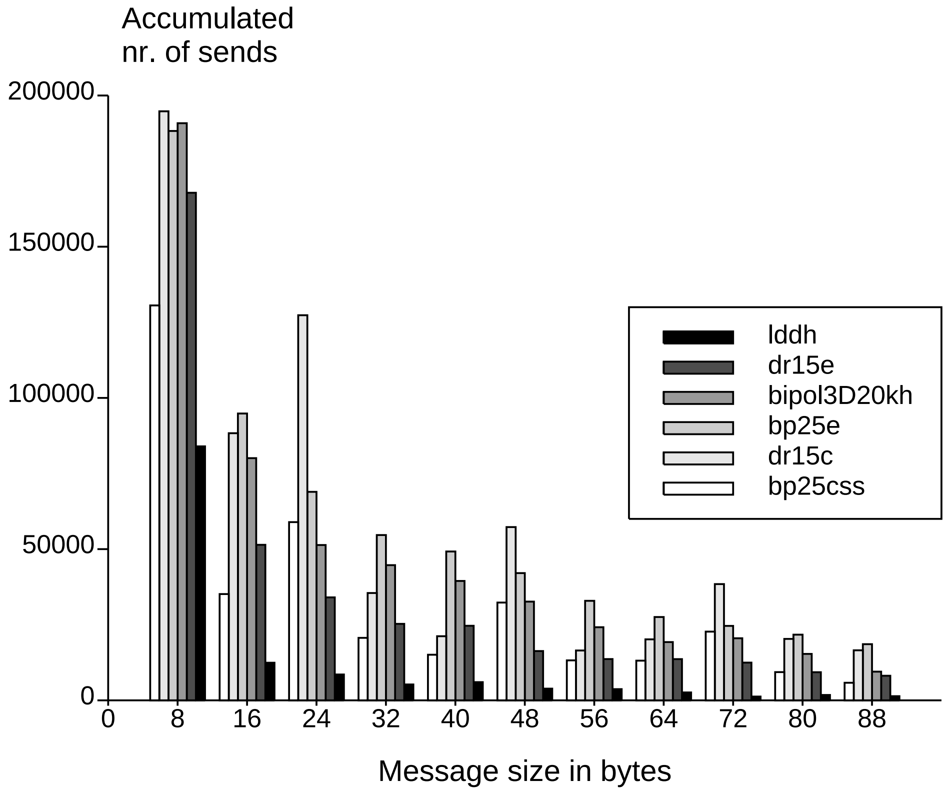

Figure 8: *Distribution of message sizes for parallel execution on an 8×8 CM5 system.*

statement but due to additional idle times which are hidden when PEs execute asynchronously. On many other machines the penalty paid for synchronization introduced at every checkpoint would be much higher. If a DMPP does not feature a separate synchronization network (as for instance the Intel iPSC and Paragon machines), performance of synchronizations degrades to multiples of the network latency.

5.5 Effect of coloring

As mentioned in Section 4.3, coloring is an essential technique to parallelize incomplete factorization preconditioners. The number of colors determines the number of communication points within a preconditioned matrix-vector multiplication.

The greedy algorithm in PILS usually finds a coloring with a low number of colors (note that finding the minimum number of colors is NP-hard). Unfortunately, coloring can induce some load imbalance. While the complete set of entries in each vector is equally distributed to all processors, the set of entries of each single color may be less balanced. In all our examples, greedy coloring induced an imbalance factor (expressed as the sum of the sizes of the largest color subset mapped to any processor, for each processor, over the problem size) of between 1.09 and 2.78.

A better load balance can be achieved by using more colors. While this optimization reduces the pure computation time, it also increases the communication time, so that a trade-off between

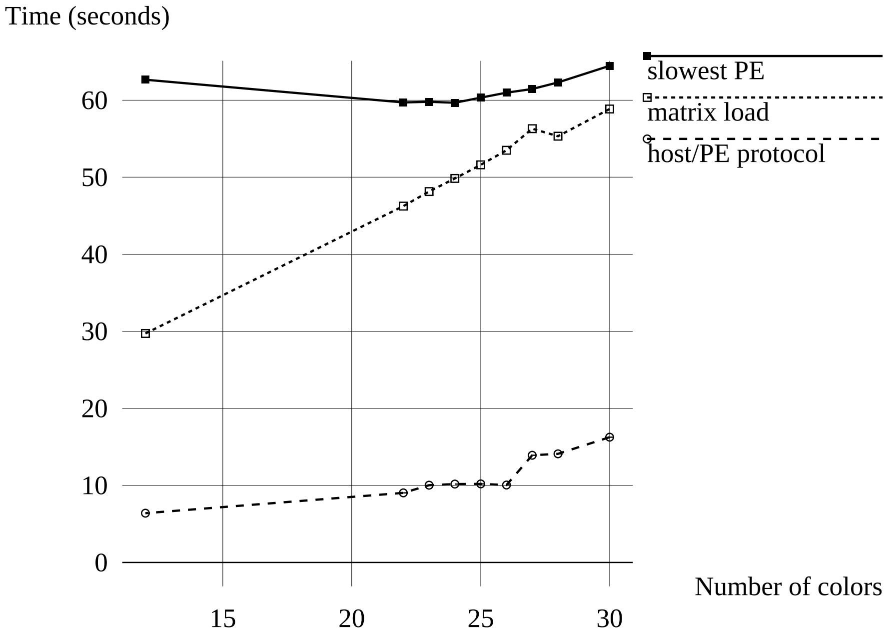

Figure 9: *Effect of the number of colors on components of the execution time.*

load balance and number of computation points has to be made. Figure 9 illustrates an experiment with balanced coloring, running on the BP25C problem on an 8×8 AP1000.

The greedy coloring heuristic produces 12 colors on this matrix, leading to an imbalance factor 2.07. By varying a maximum color imbalance parameter in the balanced coloring heuristic, we are able to generate several colorings with between 22 and 30 colors. The imbalance factor is less than 1.05 on all these balanced colorings. As can be seen in the upper curve of Figure 9, the gain through balancing the colors is quite moderate. The execution time on the slowest (that is, most loaded) PE could not be reduced by more than 5%. Furthermore, the matrix load time increases as the matrix is subdivided into smaller chunks and the host/PE protocol takes up more time because the computation is partitioned over a higher number of individual Fortran routines.

Note that the introduction of a second coloring heuristic actually conflicts with our goal to keep modifications to an absolute minimum. We added this analysis here only to emphasize on the crucial role of preconditioning in the parallelization of iterative solvers. A lexicographic ordering of the points in a 3-D grid would induce $O(\sqrt[3]{n})$ such communication points (where n is the problem size), assuming that level scheduling [18] is used to extract parallelism. On the other hand, unpreconditioned iterative methods are usually unable to solve linear systems of this kind.

Coloring modifies the preconditioner and thus affects the convergence speed. The number of iterations with the balanced colorings varied by up to ± 10 %, in comparison to the original coloring. Certain colorings appear to deteriorate the convergence speed on some regularly structured problems [32]. There is no such evidence, however, with coloring on irregular structures, and there

Method	*Precon-ditioner*	*Iter-ations*	*Time (seconds) on slowest PE*	*Time (seconds) for host-PE protocol*	*PE speedup over SS1+*
reference					
Bi-CGSTAB	split D-ILU	65	23.1	6.4	18.9
other methods					
BiCG	split D-ILU	130	45.9	10.7	19.4
CGS	split D-ILU	78	27.2	7.5	19.9
GMRES(10)	split D-ILU	388	87.4	42.1	19.3
GMRES(∞)	split D-ILU	89	41.6	37.4	24.0
other preconditioners					
Bi-CGSTAB	right D-ILU	67	33.4	8.5	17.8
Bi-CGSTAB	split ILU	61	38.0	5.8	18.1
Bi-CGSTAB	split SSOR	69	24.6	7.3	17.9
nested iterative solvers					
GCR(∞)	GMRES	17	43.0	21.1	19.2
GCR(∞)	Bi-CGSTAB	11	41.4	15.5	17.7

Table 8: *Timing of various variants of methods, for one particular problem on an 8×8 AP1000. The total solution time is the sum of the PE time, the host communication time, and the matrix load time. The latter is 44 seconds with ILU preconditioning, and 22 seconds for all the other variants. The PE time is compared to performance on a SparcStation 1+ in the rightmost column.*

is no single "natural" ordering that would achieve optimal convergence throughout a long device simulation. In our own experience, we never encountered an ordering that was consistently better than colored orderings [29].

5.6 Variants of iterative methods

Table 8 details some other variants of iterative solvers in PILS. These measurements were taken on an 8×8 AP1000, solving the system DR15C to a relative residual norm of 10^{-10}. This problem is too large for a single processor of the AP1000, and the variations of memory access patterns defy an accurate extrapolation of performance from smaller problems. Speedup in Table 8 is therefore expressed in relation to the performance on a SparcStation 1+. Since the latter is slightly faster than an AP1000 processor on small problems, the actual speedup of the parallelization may be somewhat higher than the presented numbers.

Our experience on other machines has shown that Bi-CGSTAB [33], split preconditioned with a D-ILU factorization, is the fastest iterative method on most of the linear systems occurring in semiconductor device simulation. It appears to be the fastest solver in our DMPP version as well; this is why we chose split D-ILU preconditioned Bi-CGSTAB for all other measurements in this article, and as a reference for the following comparison of other methods. The present sections just highlights a few aspects of the methods that appear to be relevant for the parallelization; the interested reader is referred to [17] for further detail.

CGS [34] has a very similar structure to Bi-CGSTAB, which explains the similarity in the parallelization results. BiCG [35] requires transposed matrix-vector products, which is often an impediment in a parallel implementation. The good speedup here is due to the partitioning by matchings in the jagged diagonal data structure of PILS, and to remote updates in the code generated by Oxygen. GMRES(∞) [36] requires a minimum number of matrix-vector multiplications, at the expense of memory and other operations. In particular, GMRES requires many dot products of vectors. These global reduction operations are a well-known bottleneck in parallel implementations; in our programming model, this bottleneck shows as a high host-to-PE bottleneck, as the host is broadcasting the result of a dot product. The dot product bottleneck can be reduced by grouping more global reduction operations together, but most of these kinds of variants of GMRES (and other methods) that have been presented appear to lose some numerical stability, and this can be devastating on ill-conditioned problems.

The most popular preconditioners perform an approximate factorization of the original system matrix A into a product of two sparse triangular matrices, $A \approx LU$. If $L+U$ has the same sparsity structure as A, a preconditioned matrix-vector product requires twice as many computations than an unpreconditioned product. Our favorite preconditioner is D-ILU, a variant of incomplete factorizations in which all off-diagonal entries of the triangular factors coincide (modulo a diagonal multiplier) with the off-diagonal nonzeros of the original matrix. When D-ILU is used in split position (that is, every preconditioned matrix-vector product applies the operator $L^{-1}AU^{-1}$ to a vector), Eisenstat's trick [37] saves almost half of the computations for a preconditioned matrix-vector product. The savings are only one quarter in right or left preconditioning (using the operators $AU^{-1}L^{-1}$ or $U^{-1}L^{-1}A$, respectively). Classical ILU preconditioning [38, 39] modifies all nonzero entries on irregularly structured matrices as ours, so that Eisenstat's savings can not be refunded. The SSOR preconditioner has the same general structure as D-ILU, and thus achieves the same parallelization effect.

Note that the most powerful preconditioner in PILS, an approximate factorization with a numerical threshold, is too finely grained for an efficient parallelization on today's DMPPs. In some large device simulations, a few among several hundreds of linear systems are so ill-conditioned that they require this more robust preconditioner. This preconditioner runs mostly in scalar mode on vector computers. In our host-attached programming paradigm, it can run sequentially on the host. A fully distributed parallelization would be severely restricted in this case.

Nested iterative solvers are another alternative for very ill-conditioned linear systems that resist standard preconditioning. A preconditioned iterative method (Bi-CGSTAB or GMRES in our examples) is used to precondition another iterative method (GCR in the examples). The inner method is again preconditioned by split D-ILU. The first of the methods listed is equivalent to GMRESR(10) [40]; it uses ten inner iterations for each outer iteration. The inner method of our second nested solver iterates only to a relative tolerance of 10^{-2}, or terminates if this inner tolerance cannot be achieved within ten iterations.

6 Conclusions

Let us recall the three main questions asked at the beginning of this project:

- How difficult is the migration of existing vector supercomputer application codes to DMPPs ?

- How much can compilers ease the above migration ?
- How should applications be designed to keep portability among vector and parallel architectures ?

With the results of Section 5 we can now answer these questions at least for packages similar to PILS, i.e. applications based on finite elements and irregular sparse matrix computations. In our case, the migration was possible due to several features of both, the application code and the supporting parallelizing compiler.

The clear separation of Fortran code for numerically intensive computing and C++ code for the more complex control-flow decisions was a key feature of PILS allowing us to successfully use the parallelization strategy outlined in Section 4.4. The fact that two different languages were used to implement both parts is not important; what allowed us to reduce host/PE traffic significantly were the relatively few data dependences between the two code parts. PILS has been used in several applications, including some semiconductor device simulators. In these applications it was used as a library, i.e. the surrounding application used PILS library calls to solve a linear system of equations stored in memory. The main conclusion for the user of similar such packages (already optimized for vector supercomputers) is to access main data structures if possible exclusively using library calls provided by the underlying package. The less data traffic exists between the surrounding application and the library, the easier our parallelization strategy could be applied. On the other hand, the package developer should include a set of library routines as complete as possible, to allow for a maximum number of operations on the main data structures.

Several features of Oxygen, which are neither supported by HPF nor in other compilation systems (i.e. the origins of HPF), were crucial for the efficient parallelization of PILS:

- Global name space support at run-time and dynamic *data consistency analysis*.
- Oxygen allows fetches and *remote updates* of distributed data (i.e. no "owner computes").
- Oxygen generated parallel programs do not use collective communication routines, which implies at least one global synchronization at the end of all parallel loops. Instead, processors synchronize implicitly by using local communications (non-blocking sends and blocking receives) as required by data exchanges.
- The sequential part of the program execution is duplicated by Oxygen on all processors.

The general opinion of the DMPP operating system research and development community seems to favor a host-less programming paradigm. We agree that in contrast to host-based machines like the AP1000, this paradigm offers more elegance and potential to overcome the host I/O bottleneck. However, unless full UNIX I/O support is provided on all PEs of the system (including a large virtual memory), a strategy as outlined in Section 4.4 for the parallelization of PILS could not be applied to such host-less system. We therefore would recommend to DMPP designers and vendors to continue to also support the "old fashioned" host-based programming paradigm, because it eases parallelization of existing complicated vector machine codes.

In addition, as we have already done in [11, 12], we must again advocate *balanced* DMPP architectures. Applications like PILS can only be parallelized efficiently on DMPPs if the underlying architecture supports equally well performing computation and communication engines. The host can and should be incorporated into the overall parallel application, and thus host communication speed is equally essential. The communication time for message sizes required by a given application should be of the same order of magnitude as the time to consume the message contents in computations. As shown in Figure 8, the compiler generated parallel code with many small message exchanges between processors (due to application requirements, not due to compiler restrictions). Therefore, the efficient execution of PILS depends strongly on the communication startup latency featured by the underlying DMPP architecture. A DMPP's communication latency, its communication bandwidth, and its local computation performance should be balanced to support efficient parallel execution of applications similar to PILS.

Acknowledgments

Access to AP1000 machines of various sizes was provided by the Fujitsu Parallel Computing Research Facilities in Kawasaki, Japan. We would like to thank T. Horie from Fujitsu Laboratories Ltd. for his support. We are grateful to G. Rancourt from Thinking Machines Corporation for providing us access to various CM5 systems in Cambridge, MA. We are indebted to W. Fichtner from the Integrated Systems Laboratory of ETH Zürich, who initiated this project shortly before we both left our shared office. We thank K. Decker and W.M. Coughran for their careful proofreading.

References

[1] G.H. Barnes, R.M.Brown, M. Kato, D.J. Kuck, D.J. Slotnick, and R.A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, pages 746–757, August 1968.

[2] David B. Loveman. High performance Fortran. *IEEE Parallel & Distributed Technology*, February 1993.

[3] P.J. Hatcher, M.J. Quinn, R.J. Anderson, A.J Lapadula, B.K. Seevers, and A.F. Bennett. Architecture-independent scientific programming in Dataparallel C: Three case studies. In *Proc. Supercomputing 91*, Albuquerque, November 1991. ACM-IEEE.

[4] H. Berryman, J. Saltz, and J. Schroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: practice and experience*, 3(3), June 1991.

[5] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. on Parallel and Distr. Systems*, 2(4):441–451, October 1991.

[6] C. Pommerell and W. Fichtner. PILS: An iterative linear solver package for ill-conditioned systems. In *Proc. Supercomputing '91*, pages 588–599, Albuquerque, November 1991. ACM-IEEE.

[7] R. Rühl. *A Parallelizing Compiler for Distributed-Memory Parallel Processors.* PhD thesis, ETH-Zürich, 1992. publ. by Hartung-Gorre Verlag, Konstanz, Germany.

[8] 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.

[9] CM-5 technical summary. Technical report, Thinking Machines Corporation, October 1991.

[10] 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.

[11] M. Annaratone, C. Pommerell, and R. Rühl. Interprocessor communication speed and performance in distributed-memory parallel processors. In *Proc. 16th Symposium on Computer Architecture*, pages 315–324, Jerusalem, June 1989. IEEE-ACM.

[12] 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.

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

[14] T. von Eiken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Symposium on Computer Architecture*, Gold Coast, Australia, May 1992. ACM-IEEE.

[15] G. Heiser, C. Pommerell, J. Weis, and W. Fichtner. Three dimensional numerical semiconductor device simulation: Algorithms, architectures, results. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(10):1218–1230, October 1991.

[16] C. Pommerell and W. Fichtner. Memory aspects and performance of iterative solvers. In *Copper Mountain Conference on Iterative Methods*. SIAM, April 1992. To appear in SIAM J. Sci. Comput.

[17] C. Pommerell. *Solution of Large Unsymmetric Systems of Linear Equations*. PhD thesis, ETH-Zürich, 1992. publ. by Hartung-Gorre Verlag, Konstanz, Germany.

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

[19] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[20] P. Beadle, C. Pommerell, and M. Annaratone. K9: A simulator of distributed-memory parallel processors. In *Proc. Supercomputing '89*, pages 765–774, Reno, November 1989. ACM-IEEE.

[21] R. Rühl. Evaluation of compiler generated parallel programs on three multicomputers. In *Proc. ACM International Conference on Supercomputing*, Washington, July 1992.

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

[23] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Trans. on Parallel and Distr. Systems*, 2(3):361–375, July 1991.

[24] S. Hirandani, K. Kennedy, and C.W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. Supercomputing 91*, Albuquerque, November 1991. ACM-IEEE.

[25] 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*, page 177, Seattle, Washington, March 1990. ACM SIGPLAN.

[26] M. Gupta and P. Banerjee. PARADIGM: a compiler for automatic data distribution on multicomputers. In *Proc. ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[27] H. P. Zima, H. J. Bast, and M. Gerndt. SUPERB: a tool for semi-automatic SIMD/MIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[28] M. Neeracher and R. Rühl. Automatic parallelization of LINPACK routines on distributed memory parallel processors. In *Proc. IEEE International Parallel Processing Symposium*, California, April 1993.

[29] C. Pommerell, M. Annaratone, and W. Fichtner. A set of new mapping and coloring heuristics for distributed-memory parallel processors. *SIAM J. Sci. Stat. Comput.*, 13(1):194–226, January 1992.

[30] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[31] T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata. Improving AP1000 parallel computer performance with message communication. In *Proc. 20th Symposium on Computer Architecture*, pages 314–325, San Diego, California, May 1993. IEEE-ACM.

[32] J. M. Ortega. Orderings for conjugate gradient preconditioning. *SIAM Journal on Optimization*, 1(4), November 1991.

[33] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, March 1992.

[34] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10(1):36–52, January 1989.

[35] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. A. Watson, editor, *Proc. of the 6-th Biennal Dundee Conference on Numerical Analysis*. Springer-Verlag, 1976.

[36] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986.

[37] S. C. Eisenstat. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Stat. Comput.*, 2(1):1–4, March 1981.

[38] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. of Comput.*, 31(137):148–162, January 1977.

[39] H. C. Elman. *Iterative Methods for Large, Sparse Nonsymmetric Systems of Linear Equations.* PhD thesis, Yale University Department of Computer Science, April 1982.

[40] H. A. van der Vorst and C. Vuik. GMRESR: a family of nested GMRES methods. Technical Report 91-80, Dept. of Techn. Math. and Inf., TU-Delft, 1991.