Toward Parallel Mathematical Software for Elliptic Partial Differential Equations

Calvin J. Ribbens
Layne T. Watson
Colin deSa

TR 91-32

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

November 27, 1991

Toward Parallel Mathematical Software for Elliptic Partial Differential Equations

Calvin J. Ribbens* Layne T. Watson* Colin deSa*

Abstract

Three approaches to parallelizing important components of the mathematical software package ELLPACK are considered: an explicit approach using compiler directives available only on the target machine, an automatic approach using an optimizing and parallelizing precompiler, and a two-level approach based on extensive use of a set of low level computational kernels. The focus is on shared memory architectures. Each approach to parallelization is described in detail, along with a discussion of the effort involved. Performance on a test problem, using up to sixteen processors of a Sequent Symmetry S81, is reported and discussed. Implications for the parallelization of a broad class of mathematical software are drawn.

1 Introduction

A dominant feature of the scientific computing scene of recent decades has been the use of mathematical software packages. It is widely recognized that an efficient way to build systems to solve large scale numerical problems is to make use of good algorithms already implemented in quality software. The community has come to depend on the availability of such packages for a great number of problems or subproblems, and in recent years the availability of such software is better than ever, with electronic services such as *netlib* and commercial packages such as IMSL and NAG.

The significance of parallel computation for large scale scientific computing is also widely recognized. The contemporary scientific computing environment is characterized by a wide variety of high performance vector and/or parallel computers—from supercomputers with a few very powerful vector processors, to shared memory machines with tens of processors, to distributed memory machines with hundreds or thousands of processors. These new architectures have the potential for solving problems very much larger and more difficult than those which have been addressed by the sequential machines of a previous generation. Efficient use of these machines often requires considerable work in designing new algorithms, however, and this has stimulated an extremely active research area for some time. Of necessity, the great majority of work has been on individual algorithms or computational kernels. As parallel and vector computers become more and more common, however, and especially as they begin to be used as general purpose scientific computing engines, there is a great need for quality mathematical software packages on these machines.

At present there are few instances of complete numerical packages available for parallel machines. There is work underway at Purdue toward parallelizing parts of ELLPACK for distributed

*Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, Virginia 24061.
The work of the authors was supported in part by Department of Energy grant DE-FG05-88ER25068 and Air Force Office of Scientific Research grant 89-0497, and by computational resources of the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory.

memory machines [13]. Vectorized versions of ITPACK have been developed [14], and recently a parallel version of ITPACK for the Cray Y-MP has been considered [21]. LAPACK [1] is a large package, similar in functionality to LINPACK and EISPACK, and designed to be portable across a range of high-performance machines (mostly shared memory vector multiprocessors). An important component of the LAPACK project is the use of a set of basic linear algebra kernels on which the software is built. In fact, our third approach to parallelization (see Section 5) follows the LAPACK philosophy in this regard. In addition to these research projects, there are several commercially available libraries that have been vectorized, but very few which are available for a broad class of parallel machines.

The purpose of this paper is to compare several approaches to parallelizing a large mathematical software package for a particular class of machine, namely a shared memory multiprocessor. We focus on ELLPACK [23], although in Section 6 we comment on implications for other types of packages as well. Three general strategies are represented in our work: explicit "by hand" parallelization using (nonportable) language extensions, automatic parallelization using a commercially available precompiler, and a two-level approach based on explicit parallelization of a set of low-level primitives and reformulation (as needed) of the code to use these primitives.

Before turning to the experiments, we must emphasize that there are significant qualitative differences between parallelizing a large package such as ELLPACK and parallelizing a single algorithm or computational kernel. There are some similarities, of course. In both cases it is important to identify sections of code that dominate the execution time, and ignore sections that take very little time. In the case of a large package however, one must identify a truly huge amount of code—many entire subprograms, in fact—that may be safely left alone. There simply is not enough time to carefully consider parallelizing every piece of the package, and the payoffs for most of such an effort are negligible. Furthermore, not every combination of methods (e.g., ELLPACK modules) is important. One needs to identify the most important computational paths through the software, and ignore insignificant or illegal paths. This is so not only to avoid unnecessary work, but because efficient parallelism for a particular sequence of modules may require certain choices for algorithms, data structures, or data distribution. If a particular sequence is not likely to be used, it should not be allowed to influence the design of the package. A related question is how to decide when "enough" of the package is parallelized, or when sufficient parallel efficiency is reached. It is impossible to give a final answer to this, of course. The best approach is simply to prioritize the set of modules and routines, and deal with as many as resources allow. Unfortunately, such a prioritization itself is nontrivial.

It is also important to recognize that the actual work of parallelizing those few sections of code that are most important is very difficult in our setting. Loops containing diagnostic output statements, conditional branches out of loops, unnecessary use of global variables (in COMMON blocks), and loops that simply do not parallelize can make the task extremely tedious for typical large scientific codes.

Another issue is that the existing package represents a considerable effort in time and money, and may have a large group of users. Throwing out large quantities of software and starting completely over is very difficult in this setting, if not from a technical or financial point of view, then certainly from a psychological point of view. While it is certain that some components may have to be completely rewritten, one prefers evolution over revolution wherever possible. Furthermore, the existing package may have a large user community with substantial applications built on top of the package. Such a community is very unlikely to suffer in silence if faced with drastic changes to the "look and feel" of the package; they simply want the software to run on their new machine, and to

run fast.

Another set of important differences between parallelizing individual algorithms and entire packages arises from the inherent tensions among the basic properties of good mathematical software. These properties certainly include raw performance, robustness, portability, and maintainability. Whereas raw performance considerations may dominate in the context of a single parallel algorithm—fine-tuned for a single "grand challenge" computation, for example—they cannot be the only consideration in the context of a large heterogeneous package. Even on a purely algorithmic level, to say nothing of robustness or portability, performance on a single kernel cannot always dictate. There are numerous examples (e.g., [4, 5]) where the best algorithm for a given computational kernel is not the best when that step is only one of a series of steps in a much larger computation. It is difficult to strike the right balance, especially when the specific applications on which a general package will be used are not known *a priori*. Keeping the software robust and portable also dictates certain things that work against raw performance. This is already true on sequential machines, but on a parallel machine the relative cost of good error detection and recovery, for example, can be even higher. Portability is also a very difficult problem for parallel mathematical software. Even within a class of similar machines there is currently no clear path toward portability. The new FORTRAN 90 may provide a degree of portability across many machines for new code, of course; but a new language is not a panacea for the problems caused by 100,000 lines of existing FORTRAN.

In view of the above comments, and remembering that ELLPACK is a large, heterogeneous, and general-purpose package, we are left with the following comments: (1) outstanding parallel efficiency simply cannot be guaranteed for every piece of code and every sequence of modules; (2) the user may have to fine-tune something if there is a relatively obscure section of code that heavily influences the performance of his or her application; (3) a parallel version of ELLPACK should remain an open and accessible system so that sophisticated users can do their own modifications and additions to ELLPACK (this seems to be even more important in a parallel environment, and is one of the strengths of the ELLPACK design).

In this paper we seek to further explore the difficulties and tradeoffs surveyed above. The remainder of the paper is organized as follows. In Section 2 we describe the mathematical software package under consideration and give general details of our numerical experiments. Sections 3, 4, and 5 describe the three approaches used and report results on a test problem. We conclude in Section 6 with further comparison of the approaches and a discussion of implications for the rest of ELLPACK and for other packages.

2 The experiment

ELLPACK is a system for numerically solving elliptic partial differential equations (pdes). It consists of a very high level language for defining pde problems and selecting methods of solution, and a library of approximately fifty problem solving methods, or *modules*. Each of the modules performs one of the basic steps in solving an elliptic pde (e.g., discretization, reordering of equations and unknowns, linear system solution). Originally designed as a performance evaluation tool (see [3], [11], [12]), ELLPACK has also proven to be useful in solving research and real world problems; it has been used as a convenient educational tool as well. It is straightforward to use ELLPACK to solve linear elliptic pdes posed on general two dimensional domains or in three dimensional boxes. The system may also be used to solve nonlinear problems, time dependent problems, and systems of elliptic equations, although some extra programming effort is needed in these cases. The

Table 1: Sequential times for the three modules considered.

five point star	15.3
linpack spd band	666.1
setup	5.8
factor (*dpbfa*)	637.7
solve (*dpbsl*)	22.6
jacobi cg	573.2

development of ELLPACK was a major collaborative effort involving more than 25 individuals and institutions. Major contributions came from the University of Texas at Austin, Yale University, the National Bureau of Standards, and Purdue University. The project was coordinated by the group at Purdue University, where the system software was developed as well. ELLPACK is a large package—the problem solving modules alone consist of over 100,000 lines of FORTRAN.

For the purpose of this paper we focus on only three modules in ELLPACK: *five point star* (discretization by second order centered finite differences), *linpack spd band* (band Cholesky factorization and triangular system solution), and *jacobi cg* (Jacobi iteration with conjugate gradient acceleration). These modules represent only a small fraction of the code in ELLPACK, but they are representative of three important classes of codes in the package, namely discretization modules, and direct and iterative solution modules. The *five point star* module was written by Ron Boisvert and John Nestor. Its performance is dominated by function evaluations and arithmetic needed to compute the coefficients of the discretization. The *linpack spd band* module consists essentially of routines *dpbfa* and *dpbsl* from LINPACK [7]. Its performance is dominated by vector products in the inner loop of *dpbfa*. The *jacobi cg* module is from ITPACK [15]; the dominant computational steps of this module are in matrix-vector multiplies and several vector-vector operations.

In the sections below we report performance for each of the three parallelization strategies on a single test problem (Problem 1 from [24]):

$$(e^{xy}u_x)_x + (e^{-xy}u_y)_y - \frac{1}{1+x+y}u = f(x,y),$$

where f is chosen so that the true solution is

$$u(x,y) = 0.75e^{xy}\sin(\pi x)\sin(\pi y),$$

and we impose Dirichlet boundary conditions on the unit square. A uniform mesh with spacing $h = 1/128$ is placed on the domain. All computations were done in double precision on a Sequent Symmetry S81. In the tables below, all times are given in seconds and represent the median of at least five runs. In Table 1 we give results for sequential ELLPACK on one processor. The time required by *linpack spd band* for "setup" is mostly in copying the discrete problem from one data structure to another.

For the remaining tables, parallel speedup and efficiency are computed in the usual way:

$$\mathrm{Sp} = T_1/T_p \qquad \text{and} \qquad \mathrm{Ef} = \mathrm{Sp}/p,$$

where T_p is time using p processors. Note that we compute speedups relative to the same code, *not* necessarily to the best available sequential code. Since we report both time and speedup, it is easy to see which methods perform best in terms of raw performance.

While our example problem is nothing more than a model problem, it does sufficiently represent a broad class of applications addressed by ELLPACK users. Our intention is to study an example of an "everyday" pde computation on an "everyday" machine. We believe that while the grand challenges of computational science are important and exciting, it is also important to have available efficient parallel versions of general purpose packages such as ELLPACK on the current generation of widely available parallel machines. With this exercise we want to illustrate and examine some of the points made in the preceding section, and begin to understand the costs and benefits of developing a parallel version of ELLPACK.

3 Explicit parallelization

Our first approach to parallelizing the ELLPACK modules is an explicit, "by hand" strategy. While this approach clearly is a poor one from the point of view of portability, it does serve as a useful baseline for comparisons, allowing us to see just what is possible when performance on only one machine is of primary importance. For this experiment we did only relatively straightforward parallelization. By this we mean that code rewriting and reorganizing was allowed, but we did not devote the significant additional effort needed to squeeze every last drop of performance out of the code (e.g., unrolling loops to optimal levels, rewriting key sections in assembly language, etc.). We feel it is impractical to expect that kind of effort to be expended across an entire mathematical software package.

The primary language extension used was the DYNIX Parallel Programming Library function *m_fork* [20], which creates processes (or reuses existing processes) and assigns them to execute copies of a specified subprogram. Arguments may be passed to each copy of the subprogram. The *m_fork* call is also a synchronization point in that after finishing with its copy of the specified subprogram, the parent process waits until all child processes have completed their execution of the process before proceeding. A call to *m_set_procs* determines how many processes will be created by subsequent *m_fork* calls. Under this scheme, the logic to assign work to processes must be supplied by the programmer. A library function *m_get_myid* allows each parallel process to determine its unique identifier—an obvious requirement in assigning iterations of a loop, for example.

Sequent's ATS FORTRAN compiler provides a set of special compiler directives which allow loops to be parallelized without directly using the Parallel Programming Library calls. The primary directive used to parallelize a loop is DOACROSS. The compiler automatically translates a DOACROSS into the appropriate declarations and *m_fork* call needed to execute the body of the loop in parallel. With DOACROSS the programmer must classify all variables used in the loop (e.g., as shared, local, reduction, etc.). It is possible to have subprogram calls within a parallel loop as long as space for the local variables of that subprogram is reserved dynamically (i.e., on the stack) rather than in static memory.

We parallelized the three modules of interest using *m_fork* (or DOACROSS where possible). We first describe parallelizing the factorization step of *linpack spd band*. The immediate problem with the original *dpbfa* code from LINPACK is that the upper triangular Cholesky factor is computed by columns instead of rows. It turns out that this original formulation does not parallelize well at all. Psuedocode for two versions of band Cholesky factorization is given in Figure 1. On the left in the figure is the original algorithm, which computes the upper triangular factor by columns; the algorithm on the right computes the factor by rows. Both algorithms overwrite the upper triangular portion of the original matrix with the Cholesky factor. In the first version, each iteration of the i loop must be completed before the next can begin; the only parallelism easily available is within

for $j = 1, n$

  for $i = \max\{1, j-m\}, j-1$

$$a_{ij} = (a_{ij} - \sum_{k=\max\{1,j-m\}}^{i-1} a_{ki}a_{kj})/a_{ii}$$

$$a_{jj} = (a_{jj} - \sum_{k=\max\{1,j-m\}}^{j-1} a_{kj}^2)^{1/2}$$

for $i = 1, n$

$$a_{ii} = (a_{ii} - \sum_{k=\max\{1,i-m\}}^{i-1} a_{ki}^2)^{1/2}$$

  for $j = i+1, \min\{n, i+m\}$

$$a_{ij} = (a_{ij} - \sum_{k=\max\{1,j-m\}}^{i-1} a_{ki}a_{kj})/a_{ii}$$

Figure 1: Two formulations of Cholesky factorization of an $n \times n$ matrix with bandwidth m.

Table 2: Time, speedup, and efficiency for three explicitly parallelized versions of the Cholesky factorization routine *dpbfa*.

	Static			Dynamic			Static-P		
p	Time	Sp	Ef	Time	Sp	Ef	Time	Sp	Ef
1	656.6	–	–	688.4	–	–	679.3	–	–
2	455.4	1.44	0.72	360.1	1.91	0.96	348.4	1.95	0.98
4	250.5	2.62	0.66	188.0	3.66	0.92	180.3	3.77	0.94
8	130.9	5.02	0.63	102.9	6.69	0.84	98.2	6.92	0.86
16	68.7	9.56	0.60	62.3	11.04	0.69	61.2	11.10	0.69

the dot products in the innermost loop. In the second version, the j loop may be done in parallel, resulting in much larger granularity and hence better parallel performance.

In Table 2 we summarize results for three versions of the explicitly parallelized Cholesky factorization. All three are based on computing the upper triangle by rows. The first version uses static scheduling, where the j loop iterations are assigned to processors statically, without regard to the cost of each iteration. Static scheduling is the default used by DOACROSS. The second version is based on a simple dynamic scheduling strategy, indicated by adding "CHUNK=1" to the DOACROSS directive ("chunk" refers to the number of iterations composing a single task). The compiler automatically includes code to define and maintain critical sections which control the assignment of iterations to processes. Each parallel process simply gets the next iteration of the loop remaining to be done.

Finally, the third version of our explicit parallelization is based on a static scheduling which takes advantage of the extremely regular variation in work from iteration to iteration. Notice from Figure 1 (right) that for a typical value of i, the work in the m iterations of the j loop varies roughly from m down to 1 (or at least is proportional to these values). We can exploit this regular pattern by statically assigning work so that each processor has nearly the same amount of total work. To illustrate this "permuted" (Static-P) assignment, suppose we have $m = 16$ tasks and four processors, and suppose the work of task i is proportional to i, for $i = 1, \ldots, 16$. Table 3 gives two typical simple-minded static assignments, and a permuted assignment derived by permuting each group of four tasks before assigning them. Notice that the wraparound assignment yields a modest load imbalance, the blocked assignment a serious imbalance, and the permuted assignment

Table 3: Three static assignment schemes for 16 tasks and four processors. The assigned tasks are listed, followed by the average workload, assuming task i requires work i.

	Wraparound					Blocked					Permuted				
Proc	Assignment				Ave	Assignment				Ave	Assignment				Ave
0	1	5	9	13	7	1	2	3	4	2.5	1	8	11	14	8.5
1	2	6	10	12	8	5	6	7	8	6.5	2	5	12	15	8.5
2	3	7	11	15	9	9	10	11	12	10.5	3	6	9	16	8.5
3	4	8	12	16	10	13	14	15	16	14.5	4	7	10	13	8.5

Table 4: Time, speedup, and efficiency for three explicitly parallelized versions of the entire solution: discretization + factorization + solution.

	Version 1			Version 2			Version 3		
p	Time	Sp	Ef	Time	Sp	Ef	Time	Sp	Ef
1	722.9	–	–	724.9	–	–	722.5	–	–
2	391.7	1.85	0.92	386.1	1.88	0.94	377.9	1.91	0.96
4	223.8	3.23	0.81	216.0	3.36	0.84	204.0	3.54	0.89
8	141.9	5.09	0.64	133.9	5.41	0.68	120.1	6.01	0.75
16	104.7	6.90	0.43	101.8	7.12	0.44	88.0	8.21	0.51

is perfectly balanced.

The data in Table 2 indicate that the default static scheduling is considerably worse than either dynamic or permuted static scheduling. Differences between the latter two schemes are not as dramatic, although there is a slight preference for Static-P. (The advantage of Static-P is greater when the extra overhead of dynamic scheduling is more severe). The speedups for large p are somewhat disappointing. This is not unexpected, however, in view of the problem size. We have at most $m = 127$ parallel tasks, each of which is little more than a dot product. So at best, each parallel process has only eight dot products to do at each iteration of the outer loop. Also, since we have only parallelized the j loop, the small amount of remaining sequential code becomes significant for large p. Finally, from the $p = 1$ processor data in Table 2 (compare with sequential time 637.7 from Table 1), it is clear that there is a noticeable amount of overhead in the parallel code—especially with dynamic scheduling, as expected.

Turning now to the full *linpack spd band* (including triangular solves) and to *five point star*, we report in Table 4 results from an entire solution—discretization plus linear system solution. In order to parallelize the main loop of the *five point star* module, the code must be modified somewhat. Each iteration of the main loop in this code generates a single equation, and can be parallelized if the numbering of equations and unknowns is done first, rather than within the main loop. With this modification, the parallel performance of *five point star* alone is reasonably good (e.g., speedup of 5.85 on eight processors).

The triangular system solves in *linpack spd band* (essentially LINPACK's *dpbsl*) are well-known to be difficult to parallelize. While a carefully constructed block-oriented algorithm can bring *some* improvement here (e.g., see [18], [10], [22]), we did not consider these more complicated strategies

Table 5: Time, speedup, and efficiency for two explicitly parallelized versions of *jacobi cg*.

	Version 1			Version 2		
p	Time	Sp	Ef	Time	Sp	Ef
1	563.2	–	–	541.8	–	–
2	285.1	1.98	0.99	273.6	1.98	0.99
4	143.7	3.92	0.98	139.2	3.89	0.97
8	72.4	7.78	0.97	72.0	7.52	0.94
16	37.7	14.95	0.93	37.8	14.35	0.90

for this paper. The resulting granularity is very fine: a parallel vector update (*daxpy*) or vector product, depending on whether we are doing the forward or back solve. Consequently, we achieve virtually no speedup for more than two processors.

The data in Table 4 should be interpreted as follows. Each version represents the complete solution procedure: *five point star* followed by *linpack spd band*. In Version 1 we only parallelized the factorization step, using the Static-P scheduling described above. For Version 2 we also parallelized the triangular solve routines as described in the previous paragraph. And for Version 3, the discretization module is also parallelized as described above. It is clear that parallelizing the triangular solve makes only a small difference, while a parallelized discretization makes a more noticeable contribution. This illustrates the point that simply plugging in a parallel factorization module may not be enough in the context of the entire computation—it is important to consider parallelizing other components, such as discretization modules. Amdahl's law reminds us that relatively inexpensive steps in a serial computation can become awfully important in a parallel environment.

Even with Version 3, however, the parallel efficiency is noticeably worse for the entire computation than for the Cholesky factorization alone. One could argue that from a "scaled speedup" point of view, where problem size grows with the number of processors, the parallel performance is much better, since the factorization step will dominate. However, it is not always sufficient to argue that scaled speedup makes all inefficiencies disappear. This is especially true for shared memory machines with modest numbers of processors, and for modestly sized problems. Frequently, one simply does not want to solve a problem that is big enough to achieve good parallel efficiencies. One simply wants to solve a given problem quickly. A typical situation involves repeated solutions of a series of pde problems, no one of which is very big, but the sum total of which is very substantial indeed (e.g., when the pde is in the inner loop of an optimization or control problem). Hence, we want to achieve reasonably efficient parallelism even for relatively modest size pde calculations.

Finally, Table 5 gives results for two versions of an explicit parallelization of *jacobi cg*. The sequential time for this module is dominated by a matrix-vector multiply in the ITPACK routine *pjac*—a subprogram that performs one Jacobi iteration. There is also substantial time spent in other (mostly vector) operations, however. Consequently, a total of 17 loops were explicitly parallelized using DOACROSS. The only difference between the two versions reported in Table 5 is our treatment of the main loop of *pjac*. Figure 2 shows the two versions of this loop. In order to understand Figure 2 one needs to understand the sparse matrix storage scheme used in ELLPACK. The nonzeros of an $n \times n$ sparse matrix A are stored by rows in the `n` $\times$ `maxnz` array `coef`, where `maxnz` is the maximum number of nonzeros per row. The array `jcoef` holds column indices, so that

```
      do 20 j = 2, maxnz
        call vgathr (n,u,jcoef(1,j),work)
        do 10 i = 1, n
10        rhs(i) = rhs(i) - coef(i,j)*work(i)
20    continue
```

```
      do 20 i = 1, n
        do 10 j = 2, maxnz
10        rhs(i) = rhs(i) - coef(i,j)*u(jcoef(i,j))
20    continue
```

Figure 2: Two versions of the main loop of ITPACK routine *pjac*.

`coef(i,j)` holds the entry in row `i`, column `jcoef(i,j)` of A. In the original version (Figure 2, top) we parallelize the inner (`do 10`) loop. The vector gather (ITPACK routine *vgathr*) is also parallelized. In the second version (Figure 2, bottom) we parallelize the outer loop, essentially doing the gather on a row-wise basis with indexing in the inner loop. Thus, Version 2 has larger granularity and does not require the temporary workspace vector `work`.

Table 5 shows that there is only a slight advantage to the second version, especially with many processors. Both versions show excellent parallel performance, due to the relatively large number of parallel tasks present throughout the computation. In fact, we see here a clear advantage for *jacobi cg* over *linpack spd band*: not only is the sequential speed of the iterative method better than the direct, but the advantage grows as we add processors. Note also that the time for both parallel versions with one processor is actually better than the original sequential time (see Table 1). The most likely explanation for this with respect to Version 1 is that storing local variables on the stack (as is done only in the parallel versions) gives a slight increase in speed. Version 2 has the added advantage of the loop interchange described above, which is more efficient sequentially.

4 Automatic parallelization

A second general approach to parallelizing large mathematical software packages is a heavy reliance on automatic detection of parallelism by compilers. The obvious potential advantage here is a reduction in effort on the part of the programmer, and a degree of portability (to the extent that such tools are available on other machines). It is widely recognized that automatically recognizing potential parallelism, and performing the necessary code transformations, is an extremely difficult problem. Hence, a major goal of such software tools is typically to provide good feedback to the user about what parts of the code could and could not be parallelized. In this way the user can do some code modifications which might help the compiler, but which generally require much less effort than doing the entire process alone. The compiler may function more as an assistant to the programmer than as the sole agent for "dusty deck" parallelization.

We used the KAP/Sequent FORTRAN source-to-source preprocessor developed by Kuck & Associates [16]. KAP trys to recognize parallelism in sequential code and automatically convert programs to run in parallel using Sequent parallel programming directives. KAP also does optimizations to improve the scalar performance of codes. There are KAP products available for both

Table 6: Time, speedup, and efficiency for KAP-based approach.

	dpbfa			*jacobi cg*		
p	Time	Sp	Ef	Time	Sp	Ef
1	655.7	–	–	569.8	–	–
2	455.4	1.44	0.72	289.1	1.97	0.99
4	250.1	2.62	0.66	145.4	3.92	0.98
8	130.8	5.01	0.63	73.8	7.72	0.97
16	68.8	9.53	0.60	38.6	14.77	0.92

C and FORTRAN, and for a variety of shared memory (vector) multiprocessors. With the one exception noted below, we used the default KAP options for all our experiments.

Since parallelizing the *five point star* module requires nontrivial code modifications, it is not surprising that KAP was unable to achieve any speedups in this code. KAP had no more success with the triangular solves than the explicit approach either. In Table 6 we report more successful results from the Cholesky factorization routine *dpbfa* and *jacobi cg*. For *dpbfa* we used a KAP directive which sets the chunksize to one (implying dynamic scheduling). Without this option the performance is essentially the same as the statically scheduled version in Table 2. It must also be emphasized that we used the row-wise version of *dpbfa* described in Section 3; KAP achieves virtually no speedup on the original column-wise version. As expected, KAP only parallelized the single loop that we also did by hand, and the performance is essentially the same as the dynamically scheduled version in Table 2.

Concerning the *jacobi cg* results in Table 6, we again see good parallel performance—only slightly slower than that achieved by explicit parallelization. There were only two loops which KAP would not parallelize that had been in our explicit approach—neither amounting to a significant amount of time. For two other loops it was necessary to use the *assert* directive to convince KAP to parallelize a loop despite the presence of procedure calls. Interestingly, KAP parallelized two minor loops which we had missed in our explicit parallelization.

5 BLAS-based parallelization

The third major approach we consider for parallelizing a large mathematical software package is a two-level strategy built on good parallel implementations of a set of low level primitives. For our applications, the most obvious set of primitives are the three levels of Basic Linear Algebra Subprograms (BLAS) [17, 9, 8], and two related sets of kernels: a sparse BLAS set proposed in [6] and the iterative BLAS proposed in [19]. If an efficient implementation of these kernels is available on a given machine, and if the most time consuming code in the package can be rewritten to use these kernels as much as possible, then a good balance between performance and portability can be achieved. In fact, several of the modules in ELLPACK already make use of Level 1 BLAS, so this approach is not entirely foreign to the package.

In terms of the three ELLPACK modules of interest, we again get no help from this approach for *five point star*. The *linpack spd band* module can be rewritten in terms of Level 2 BLAS calls, however. In particular, the dominant work of each step of the factorization can be formulated in terms of the matrix-vector operation $y = y - A^T x$, where A is a general band matrix. This

Table 7: Time, speedup, and efficiency for BLAS-based approach.

	dpbfa			*jacobi cg* 1			*jacobi cg* 2		
p	Time	Sp	Ef	Time	Sp	Ef	Time	Sp	Ef
1	684.4	–	–	577.9	–	–	578.5	–	–
2	362.6	1.89	0.94	303.0	1.91	0.95	291.9	1.98	0.99
4	195.2	3.51	0.88	163.9	3.53	0.88	148.3	3.90	0.98
8	112.1	6.10	0.76	95.8	6.03	0.75	76.3	7.58	0.95
16	73.0	9.37	0.59	63.5	9.10	0.57	41.1	14.08	0.88

corresponds to the BLAS Level 2 routine *dgbmv*. A more complicated version is possible, where we use a triangular matrix-vector multiply routine for most of the steps and the general one "in the corners" (i.e., in the first and last few steps of the factorization), but our experiments show the performance improvements are not significant. With the factorization formulated in terms of *dgbmv*, and with a straightforward (dynamically scheduled) parallelization of *dgbmv* itself, the results are as shown in the first column of Table 7. Comparing these results with the explicitly parallelized results in Table 2, it is clear that we have lost a little performance, especially for large numbers of processors. The extra overhead of using a general routine like *dgbmv* and the fact that there is slightly more sequential computation in the BLAS-based version causes this degradation.

A block oriented version of Cholesky factorization, using Level 3 BLAS routines, is also possible [2]. While block oriented methods have much better performance on dense matrices, a recent report of DuCroz, et al. [10] considers such an algorithm for band matrices. This variation would be a possible alternative for the version we implemented.

Rewriting *jacobi cg* to make extensive use of BLAS kernels is fairly straightforward. It requires the use of scatter and gather operations from the sparse BLAS, a matrix-vector multiplication kernel (*yasx2*) from the iterative BLAS, and several Level 1 BLAS. Since the ITPACK code is organized to be easily vectorizable, much of the computation is organized into simple vector operations (e.g., vector fill, scaling). These should be parallelized in our setting. Although such vector operations are technically not part of any of the BLAS standards we are using, they are obviously simple kernel computations which can be easily made available on any architecture. The importance of parallelizing the two most important vector operations is clear from Table 7, where the only difference between the two *jacobi cg* versions is parallelized vector fill and vector scale. With this improvement, the results are quite close to the explicitly parallelized results in Table 7.

6 Discussion and conclusions

6.1 Summary of comparisons

We have considered three general strategies for parallelizing large mathematical software packages: explicit parallelization using nonportable language extensions, automatic parallelization using the KAP preprocessor, and a BLAS-based strategy. In terms of effort, a very brief summary is that the explicit approach takes the most effort and KAP the least. The BLAS-based approach can take nearly as much effort as the explicit, especially if one does not already have parallel versions of the kernels themselves.

Regarding performance, we have seen that results for both the KAP and the BLAS-based approach, when used carefully and intelligently, can approach those of the explicitly parallelized code. It must be pointed out that our parallelization of the BLAS kernels themselves was entirely straightforward; we did none of the optimizing that would normally be done for such routines. Despite some success with KAP and the BLAS-based strategy, there are important cases, such as the *five point star* module, where these approaches fail completely. Considerable code modification can be required. The automatic approach, in particular, has problems with several typical features of high quality mathematical software, namely extensive use of subroutine and function calls, extensive error checking for increased robustness, and extensive use of general workspace arrays (often duplicate copies of workspace are needed in order to parallelize a section of code). Without considerable code modification, these characteristics give automatic parallelizers great difficulty. The BLAS-based strategy does not solve all problems either, of course. Not everything in ELLPACK is expressible as a BLAS kernel; an important first step under such an approach would be to identify a set of additional "basic pde solving" kernels which could then be added to the layer on which a parallel ELLPACK is built.

Purely in terms of portability, using a tool such as KAP is the most attractive, although it is not guaranteed that an important directive such as "chunk" will be available on all machines. The BLAS-based approach also is quite portable, requiring only efficient implementations of the kernels on a new machine.

6.2 Variable granularity

An interesting implication of the heterogeneity of a package such as ELLPACK, and which arose in our work, is that one fixed level of granularity is not sufficient for every algorithm or every application. Even for a single piece of code, there are cases when a single loop should be run in parallel (i.e., spread across available processors) and other cases when it should be run sequentially—because, for example, other copies of the same code are running concurrently on other processors.

Consider the various levels of the BLAS, for example. It is easy to think of typical computations where parallelism at each of the three levels is appropriate. Computing acceleration parameters for an iterative linear equation solver often requires that a single vector operation (e.g., dot product) be done very quickly. At a higher level, a single sweep of an iterative linear equation solver is primarily a matrix-vector operation. At a higher level still, block methods for solving linear equations and tensor product alternating direction methods are efficiently implemented in terms of matrix-matrix operations. One could even consider a higher level of granularity, in which an entire pde solve was viewed as an atomic task (e.g., domain decomposition methods).

The point is that fixing the level of granularity for a given routine at a particular level is a mistake. It would be better to allow the best level of granularity to be selected, depending on the given computation. One possibility is to introduce an ELLPACK option *granularity*. This option could have four possible values corresponding to the four levels illustrated in the above paragraph. Users could set this option statically by setting the option in their ELLPACK program, or (since there would be a corresponding FORTRAN variable) change granularity at runtime. The functionality of this option would be implemented by modifying the BLAS so that they always executed sequentially, unless the granularity option was set to their own level. For example, the *saxpy* routine would do its work in parallel only if $granularity = 1$; for other values it would execute sequentially (the typical situation being that parallelism was occurring at higher levels of granularity). This idea needs further investigation; but we believe that something along these lines is needed to allow efficient parallel execution of the wide variety of computations for which

ELLPACK is designed.

6.3 Conclusions and future directions

A general strategy emerging from the above experiments and comments would combine the two-level, BLAS-based approach with a few extras. As already mentioned, a few additional "basic" subprograms are needed (e.g., evaluate a function at a set of grid points, evaluate a set of integrals over a set of elements). Applying this strategy to other types of mathematical software packages would require that a similar set of kernels be identified. Furthermore, it would be very useful if one could (portably) define a simple parallel loop to handle the odd loops that do not correspond to a reasonably low level kernel and yet should be done in parallel. Given the continuing ferment in parallel language research, it is likely to be some time before such a possibility exists, however.

While a tool such as KAP is quite easy to use, and is a useful tool to have available in certain situations, it simply cannot be expected to recognize the often extensive code modifications which are sometimes needed to achieve good speedups. The three modules we consider in this paper are representative of typical ELLPACK modules, but there are certainly much more complicated codes in the package than these. In our experience, KAP has very little success with these more complicated programs.

Since some code modification is needed in any case, we feel that building on low level BLAS-like primitives, efficiently implemented, is the best strategy. Again, since some nonportability is unavoidable, we feel the best strategy is to isolate as much of it as possible in a small number of kernels (precisely the rationale behind routines that define machine constants for many software packages, for example). Ideally, nonportability is hidden in a precompiler or compiler, or in a library of system calls; but lacking these, we prefer to define our own library of problem-oriented functions, and to build on these. Finally, the BLAS-based strategy is attractive in terms of flexibility and adaptability in the context of future systems. In a sense, this is another portability issue (i.e., portability between current and future systems). It is important that parallel mathematical software packages be flexible enough to evolve into subsequent generations of problem solving environments. Clearly, the current scientific computing environment is not likely to remain unchanged for very many years. Mathematical software packages will continue to be needed as environments change (new hardware, user interfaces, computational demands, research paradigms). We feel that isolating the most critical and system dependent code in a small number of kernel routines will facilitate a smooth evolution path, and that this appears to be the most promising path overall to balancing the conflicting goals of parallel performance, programmer effort, and portability.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. Technical Report CS-90-105, Computer Science Department, University of Tennessee, Knoxville, TN, 1990.

[2] E. Anderson and J. Dongarra. Evaluating block algorithm variants in LAPACK. Technical Report CS-90-103, Computer Science Department, University of Tennessee, Knoxville, TN, 1990.

[3] J. P. Bonomo, W. R. Dyksen, and J. R. Rice. The ELLPACK performance evaluation system. Technical Report CSD-TR 569, Department of Computer Sciences, Purdue University, West Lafayette, IN, 1986.

[4] A. Chakraborty, D. C. S. Allison, C. J. Ribbens, and L. T. Watson. Parallel orthogonal decompositions of rectangular matrices for curve tracking on a hypercube. In J. L. Gustafson, editor, *Proc. Fourth Conf. on Hypercubes, Concurrent Computers, and Applications*, pages 651–654. ACM, 1989.

[5] A. Chakraborty, D. C. S. Allison, C. J. Ribbens, and L. T. Watson. Note on unit tangent vector computation for homotopy curve tracking on a hypercube. *Parallel Comput.*, 1992. to appear.

[6] D. S. Dodson and J. G. Lewis. Proposed sparse extensions to the basic linear algebra subprograms. *ACM Signum Newsletter*, 20(1):22–25, 1985.

[7] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users Guide*. SIAM, Philadelphia, PA, 1979.

[8] J. J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16:1–17, 1990.

[9] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14:1–17, 1988.

[10] J. DuCroz, P. Mayes, and G. Radicati. Factorizations of band matrices using level 3 BLAS. Technical Report CS-90-109, Computer Science Department, University of Tennessee, Knoxville, TN, 1990.

[11] W. R. Dyksen, E. N. Houstis, R. E. Lynch, and J. R. Rice. The performance of the collocation and galerkin methods with hermite bi-cubics. *SIAM J. Numer. Anal.*, 21:695–715, 1984.

[12] W. R. Dyksen, C. J. Ribbens, and J. R. Rice. The performance of numerical methods for elliptic problems with mixed boundary conditions. *Num. Meth. PDEs*, 4:347–361, 1988.

[13] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, and K-Y Wang. Parallel ELLPACK: a numerical simulation programming environment for parallel mimd machines. Technical Report CSD-TR 949, Department of Computer Sciences, Purdue University, West Lafayette, IN, 1990.

[14] D. R. Kincaid, T. C. Oppe, and D. M. Young. ITPACKV 2d user's guide. Technical Report CNA-232, Center for Numerical Analyis, University of Texas at Austin, Austin, TX, 1989.

[15] D. R. Kincaid, J. R. Respess, D. M. Young, and R. G. Grimes. ITPACK 2c: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Trans. Math. Softw.*, 8:302–322, 1982.

[16] Kuck & Associates, Champaign, IL. *KAP/Sequent User's Guide*, 1989.

[17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[18] P. Mayes and G. Radicati. Banded Cholesky factorization using level 3 BLAS. Technical Report ANL/MCS-TM-134, Argonne National Laboratory, Argonne, IL, 1989.

[19] T. C. Oppe and D. R. Kincaid. Are there iterative BLAS? In *Proceedings of the Copper Mountain Conference on Iterative Methods*, 1990.

[20] A. Osterhaug. *Guide to Parallel Programming.* Sequent Computer Systems, Inc., Beaverton, OR, 1987.

[21] M. Ramdas and D. R. Kincaid. Parallelizing ITPACKV 2d for the Cray Y-MP. Technical Report CNA-249, Center for Numerical Analyis, University of Texas at Austin, Austin, TX, 1991.

[22] C. J. Ribbens. On parallel ELLPACK for shared memory machines. Technical Report 90–46, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA, 1990.

[23] J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems Using ELLPACK.* Springer-Verlag, New York, 1985.

[24] J. R. Rice, E. N. Houstis, and W. R. Dyksen. A population of linear, second order, elliptic partial differential equations on rectangular domains. *Math. Comp.*, 36:475–484, 1981.