Strategies for Parallelizing PDE Software

Calvin J. Ribbens and George G. Pitts

TR 92-34

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

June 10, 1992

Strategies for Parallelizing PDE Software*

Calvin J. Ribbens
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

George G. Pitts
Department of Mathematics
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

Abstract

Three strategies for parallelizing 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. Each approach to parallelization is described in detail, along with a discussion of the effort involved. In connection with the third strategy, a set of computational kernels useful for PDE solving is proposed. We describe our experience in parallelizing six problem solving components of ELLPACK using each of the three strategies and give performance results for a shared memory multiprocessor. Our results suggest that the two-level strategy allows the best balance among programmer effort, portability, and parallel performance.

1 Introduction

Parallel computation and mathematical software packages are two key components of today's high performance scientific computing environment. It is widely recognized that well-written mathematical software packages can be extremely useful in building systems to solve large scale numerical problems. The community has come to depend on the availability of good algorithms, implemented in quality software, for a great number of problems. 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. As parallel and vector computers become more and more common, and especially as they begin to be used as general purpose scientific computing engines, there is a great need for a wide variety of quality mathematical software packages on a wide variety of machines.

*To appear in the proceedings of Seventh IMACS International Conference on Computer Methods for Partial Differential Equations. This work was supported in part by Department of Energy grant DE-FG05-88ER25068 and by computational resources of the Advanced Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory.

At present there are few examples of large numerical packages available for parallel machines. Strout, et al. [23] report on converting an ODE package for the Cray X-MP. Skjellum and Baldwin [22] describe a set of portable numerical algorithms for message-passing machines. There is work underway at Purdue toward parallelizing parts of ELLPACK for distributed memory machines [9]. Vectorized versions of ITPACK have been developed [10], and recently a parallel version of ITPACK for the Cray Y-MP has been considered [18]. 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. The third strategy for parallelization considered in this paper 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 lack of mathematical software packages for parallel machines is not surprising given the rapid evolution in hardware, the unstable and immature software environments on most new parallel machines, and the many difficulties inherent in building good parallel mathematical software. Developing a completely new package for parallel machines is a very expensive and time consuming process. Given the importance of having good mathematical software on parallel machines and the huge investment in existing sequential packages, it is natural to consider ways in which existing packages can be evolved toward efficient implementations on a range of parallel machines. Unfortunately, parallelizing existing mathematical software packages is also a very difficult process: they are large, complex, and often have many customers who do not want to see significant changes made to their favorite package (other than improved performance on the new parallel machines, of course!).

The purpose of this paper is to compare three approaches to parallelizing a large mathematical software package for a shared memory multiprocessor. We focus on ELLPACK [20], a well-known package for solving elliptic partial differential equations (PDEs). The three strategies considered are 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 of the code, as needed, to use these primitives. In connection with the two-level approach, we propose a set of kernels appropriate for PDE solving software. Section 2 describes the three approaches in more detail.

In order to illustrate and evaluate the three strategies considered, we have used each strategy to parallelize six different ELLPACK problem solving modules. These include three discretization modules (*five point star*, *hodie*, and *hermite collocation*), two linear system solution modules (*linpack spd band* and *jacobi cg*), and a triple module (*hodie fft*) which includes both discretization and solution. In Section 4 we describe our experience in parallelizing these codes and report parallel performance for a test problem on a Sequent Symmetry S81 with up to 16 processors. Section 3 describes the setup for our numerical experiments. We conclude with a summary and discussion in Section 5.

2 Three strategies

2.1 Explicit parallelization

Our first approach to parallelizing the ELLPACK modules is an explicit, "by hand" strategy. Portability is obviously sacrificed with this approach, and hence it is not a realistic alternative for widely used packages. However, 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: code rewriting and reorganizing was allowed, but we did not devote the significant additional effort needed to squeeze every last ounce of performance out of the code (e.g., unrolling loops, 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* [17] which creates processes (or re-uses existing processes) and assigns them to execute copies of a specified 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. Under this scheme, the logic to assign work to processes must be supplied by the programmer. 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 DOACROSS directive is used to parallelize a loop. The compiler automatically translates the DOACROSS directive into the appropriate declarations and *m_fork* call needed to execute the body of the loop in parallel.

2.2 Automatic parallelization

A second general approach to parallelizing large mathematical software packages relies heavily on automatic detection of parallelism by compilers. The obvious potential advantages here are 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 to provide good feedback to the user about what parts of the code could and could not be parallelized. In this way the user may be able to collaborate with the compiler, achieving reasonably efficient code with less effort than by doing the entire process alone.

We used the KAP/Sequent FORTRAN source-to-source preprocessor developed by Kuck & Associates [12]. 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 C and FORTRAN, and for a variety of shared memory (vector) multiprocessors. With only a couple of exceptions (noted in Section 4) we used the default KAP options for all our experiments.

2.3 A two-level strategy

The third major approach we consider for parallelizing a large mathematical software package is a two-level strategy built on parallel implementations of a set of low level primitives. This approach is motivated in essentially the same way as the three levels of Basic Linear Algebra Subprograms (BLAS) [13, 6, 5]. In fact, for PDE applications, these kernels themselves are of considerable

Table 1: Parallel kernels for pde solving.

Name	(Source)	Function
daxpy	(BLAS1)	vector update: $y \leftarrow \alpha x + y$
dcopy	(BLAS1)	copy one vector to another: $y \leftarrow x$
ddot	(BLAS1)	dot product of two vectors: $x^T y$
dgbmv	(BLAS2)	band matrix vector multiply: $y \leftarrow \alpha Ax + \beta y$
dgemm	(BLAS3)	matrix matrix multiply: $C \leftarrow \alpha AB + \beta C$
dgemv	(BLAS2)	matrix vector multiply: $y \leftarrow \alpha Ax + \beta y$
dgthr	(SPBLAS)	vector gather: $x_i \leftarrow y_{k_i}$
dscal	(BLAS1)	scale a vector by a constant: $x \leftarrow \alpha x$
dsctr	(SPBLAS)	vector scatter: $y_{k_i} \leftarrow x_i$
dsyrk	(BLAS3)	perform a symmetric rank k update: $C \leftarrow \alpha AA^T + \beta C$
dtbsv	(BLAS2)	solve a single banded triangular system: $x \leftarrow A^{-1}x$
dtrsm	(BLAS3)	solve triangular systems of equations: $B \leftarrow \alpha A^{-1}B$
dyasx2	(ITBLAS)	sparse matrix-vector multiply: $y_i \leftarrow y_i + \alpha \sum_{j=1}^{k} a_{i,j} x_{m_{i,j}}$
dvadd		componentwise vector addition: $y_i \leftarrow x_i + y_i$
dvfill		vector fill: $x_i \leftarrow \alpha$
dvmult		componentwise vector multiplication: $x_i \leftarrow x_i y_i$
dvrecp		componentwise vector reciprocal: $y_i \leftarrow 1/x_i$
dvsqrt		componentwise vector square root: $y_i \leftarrow \sqrt{x_i}$
eval_grid		evaluate a real function on a grid, returning a matrix
eval_grid_int		evaluate an integer function on a grid, returning a matrix
foreach_point		execute a subroutine once for each point in grid
foreach_hline		execute a subroutine once for each horizontal grid line
foreach_vline		execute a subroutine once for each vertical grid line
foreach_proc		execute a subroutine once for each process

use. Several of the modules in ELLPACK already make use of Level 1 BLAS, so this approach is quite natural for this package. If the number of kernels can be kept relatively small, if an efficient implementation of the kernels is available on a given machine, and if the most time consuming code in the package can be written in terms of these kernels, then a good balance between performance and portability can be achieved. However, there can be considerable work involved if the existing code makes no use of the kernels.

In Table 1 we list the kernels used for the experiments reported in this paper. The first set are taken from the dense BLAS, sparse BLAS [3] or iterative BLAS [16]. The leading "d" indicates double precision data. The second set of five kernels are simple vector operations. The final set of six kernels are more unique to PDE solving. For example, it is useful to have an operation such as foreach_point which allows a given computation to be performed in parallel at each grid point of a rectangular grid. Similar operations over finite element discretizations are obviously possible. We also need operations that apply a function to each horizontal or vertical grid line. It may also be efficient to organize a foreach_point computation by lines in order to improve granularity. The foreach_proc function is used for initializations that need to be done only once for each process. Note that the members of this last block of operations are not kernels in the traditional sense, in

that they do not represent a single simple operation. Instead, they might be termed "operators" or "parallel control structures" since they take an arbitrary function and apply it at each grid point, line, process, etc. For our purposes they do meet the most important criterion for kernels: they hide nonportable details in a small section of code, allowing the large majority of code to remain as is. For purposes of this paper we did no special optimizations of the kernels themselves. We simply implemented them in FORTRAN, using the language extensions available under Sequent's ATS FORTRAN compiler. FORTRAN listings of our current Sequent implementations of the last set of kernels are included in the appendix. Parameters are defined there as well. It is worth mentioning that each of the last set of kernels takes only a small fixed number of parameters. This means that to do complex operations at each point, line, etc., one may have to pass considerable data in COMMON blocks. This is not attractive in many cases. A better alternative might be to allow a variable number of parameters to be passed along with the function or subroutine which is to be applied in parallel. A kernel allowing this would not be implementable in FORTRAN on many systems, however.

3 An experiment

ELLPACK is a system for numerically solving elliptic 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 modules. Each of the modules performs one of the basic steps in solving an elliptic PDE: discretization, reordering of equations and unknowns, linear system solution, etc. 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. The problem solving modules comprise over 100,000 lines of FORTRAN.

For the purpose of this paper we focus on six modules in ELLPACK: *five point star* (discretization by five point centered finite differences), *hermite collocation* (discretization by collocation with hermite bicubic basis functions), *hodie* (discretization by nine point HODIE method), *linpack spd band* (band Cholesky factorization and triangular system solution), *jacobi cg* (Jacobi iteration with conjugate gradient acceleration), and *hodie fft* (HODIE discretization and solution by fast solver). These modules amount to only a fraction of the code in ELLPACK, but they are representative of the important classes of codes in the package, namely discretization modules, direct and iterative solution modules, and so-called "triple" modules which include both discretization and solution.

In the next section we report performance for each of the six modules, using each of the three parallelization strategies on a test problem. For all but *hodie fft* we approximately solve:

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

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. For *hodie fft* the test problem is $\nabla^2 u - u = f$, where the right side function is again chosen so that the true solution is known. A uniform mesh with spacing $h = 1/128$ (1/64 for *hermite collocation*) is placed on the domain, resulting in a discrete system with 16129 equations and unknowns (16384 for *hermite collocation*). For the solution modules, *linpack spd band* and *jacobi cg*, we used the linear system generated by *five point star*. All computations were done in double precision on a Sequent Symmetry S81.

Table 2: Time, speedup, and efficiency for *five point star*. Sequential time is 15.35.

	Explicit			Kernel		
p	Time	Sp	Ef	Time	Sp	Ef
1	15.43	0.99	99	16.04	0.96	96
2	7.90	1.94	97	8.10	1.90	95
4	4.04	3.80	95	4.20	3.65	91
8	2.13	7.21	90	2.17	7.07	88
16	1.16	13.23	83	1.18	13.01	81

4 Performance results

In this section we describe our experience in developing parallel versions of six ELLPACK modules using the three strategies described in Section 2. For each module we briefly mention the most important aspects of the parallelization process and comment on the results. In order to be fair, we tried to devote about the same amount of effort to each approach. Thus, as mentioned above, we only did relatively straightforward code modifications in the explicit and kernel approaches, and we implemented the kernels themselves in a very straightforward way. We used assertions and directives in a few cases to assist KAP in its parallelizing, but we felt it was not fair to do all the modifications required by the explicit approach, and then give that code to KAP. If that were allowed, then KAP's performance would be the same as that of the explicitly parallelized code, and KAP would only be used to put in the DOACROSS compiler directives, a very small help in the context of the entire effort. In Tables 2–7 all times are given in seconds and are the average of at least three runs. The variation in time from run to run was less than 3%. Parallel speedup and efficiency are measured relative to the sequential time taken by the standard ELLPACK implementation.

FIVE POINT STAR. The *five point star* module was written by Ron Boisvert and John Nestor [20]. Its performance is dominated by function evaluations and arithmetic needed to compute the coefficients of the discretization. In order to parallelize the main loop of this module, the code must be modified somewhat. Each iteration of the main loop generates a single equation, and can be done in parallel if a relatively minor dependency involving equation and unknown numbering is removed. A second loop which decides which grid points correspond to unknowns can also be parallelized after some minor modifications. KAP is unable to make these necessary modifications however, since they require interprocedural analysis and knowledge of certain ELLPACK data structures. A version based on PDE solving kernels is relatively easy to derive using dvfill, eval_grid_int, and foreach_point. Table 2 shows quite good performance for both the explicit and the kernel parallel version of *five point star*.

HERMITE COLLOCATION. The *hermite collocation* module was written by Elias Houstis [8]. Like *five point star*, the majority of the work is in function evaluations and arithmetic needed to compute the coefficients of the discrete problem. In order to parallelize the main loop which generates the equations a few more code modifications are required than in the case of *five point star*. While KAP does parallelize a couple of minor loops, it again has no success with the most important loops. The kernel version uses foreach_point, foreach_hline, foreach_vline, and dvfill, and as can be seen in Table 3, achieves results comparable with the explicitly parallelized version.

Table 3: Time, speedup, and efficiency for *hermite collocation*. Sequential time is 12.14.

	Explicit			Kernel		
p	Time	Sp	Ef	Time	Sp	Ef
1	11.93	1.02	102	11.97	1.01	101
2	6.04	2.01	100	6.08	2.00	100
4	3.13	3.88	97	3.15	3.85	96
8	1.64	7.40	93	1.64	7.40	93
16	0.96	12.65	79	0.96	12.65	79

Table 4: Time, speedup, and efficiency for *hodie*. Sequential time is 37.65.

	Explicit			Kernel		
p	Time	Sp	Ef	Time	Sp	Ef
1	39.63	0.95	95	37.99	0.99	99
2	20.13	1.87	94	19.65	1.92	96
4	10.34	3.64	91	9.93	3.79	95
8	5.13	7.34	92	5.00	7.53	94
16	2.79	13.49	84	2.65	14.21	89

HODIE. The *hodie* module was written by Robert Lynch [14]. Fourth order accuracy is achieved for the problem given by (1). In addition to the usual function evaluations and arithmetic, HODIE methods require extra evaluations of the right side function f and the solution of a small linear system to determine the coefficients of each linear equation.

Good parallel performance can be achieved for *hodie* if the main loop over the horizontal grid lines can be parallelized. Each iteration of this loop generates the equations corresponding to a single grid line. The code is particularly hard to parallelize, however, because it makes extensive use of COMMON blocks, with some data in COMMON needing to be shared among the parallel processes, and some needing to be private. Furthermore, exactly which data should be shared depends on the problem (e.g., in a constant coefficient case each process can share a copy of certain data, while in a variable coefficient case they each need their own copy). These problems can be worked around in an explicitly parallelized version by making local some of the variables that were in COMMON, and at the expense of some extra overhead for copying. Table 4 shows that good parallel performance is achieved. It is not surprising that KAP is unable to make the rather significant changes required; it parallelized eight minor loops and achieved virtually no speedup. Finally, a kernel version of *hodie*, using dvfill and foreach_hline, performs similarly to the explicit version. Note also that *hodie* can be used to generate a sixth order accurate discretization for a simpler problem (e.g., a generalized Helmholtz problem). Similar parallel performance is achieved for this case as well.

HODIE FFT. The *hodie fft* module was written by Ron Boisvert [2]. It solves a Helmholtz problem using fourth order accurate 9-point compact finite differences and linear system solution by the fast fourier transform (fft). The method is extremely fast, especially when $n = 1/h = 2^k$ for some k, as is true in our example. The dominant work is in function evaluations, n fft's of length

Table 5: Time, speedup, and efficiency for *hodie fft*. Sequential time is 11.71.

	Explicit			Kernel		
p	Time	Sp	Ef	Time	Sp	Ef
1	10.55	1.11	111	11.79	0.99	99
2	5.49	2.13	107	6.17	1.90	95
4	2.82	4.15	104	3.15	3.72	93
8	1.50	7.81	98	1.79	6.54	82
16	0.85	13.78	86	1.18	9.92	62

n, the solution of n $n \times n$ tridiagonal linear systems, and n inverse fft's of length n.

Our explicitly parallelized version of *hodie fft* contains several loops that are relatively easy to parallelize and do initializing, copying, function evaluations, and vector operations. Approximately 25% of the sequential time is spend in fft's and tridiagonal solves. This work can be parallelized very efficiently if new private workspace can be provided for each parallel task. The size of the workspace depends on the mesh size h, so, assuming no dynamic memory allocation, a local declaration in the library routine will not suffice. The ELLPACK system includes a preprocessor which takes a user's high level description of the problem (the ELLPACK "program") and generates a FORTRAN program with arrays dimensioned to the appropriate sizes, as a function of the mesh size requested by the user. In the present case we can modify the preprocessor to declare private workspace of the appropriate dimension. This is done with a TASK COMMON declaration in ATS FORTRAN. With this solution we see from Table 5 that the results are excellent. The explicit version is more efficient than the sequential version because the extra workspace allows a more efficient version of a few loops in the discretization phase.

The KAP and kernel parallel versions of *hodie fft* do not fare as well as the explicitly parallelized version. KAP parallelizes 100 loops in all (although many only run in parallel if some granularity condition introduced by the system is satisfied). The resulting parallel code achieves only negligible speedup, however. The kernel version meanwhile, is based on eval_grid, foreach_hline, foreach_vline, foreach_proc, and a few Level 1 BLAS routines. From Table 5 it can be seen that the parallel efficiency of the kernel version degrades much more quickly with increasing numbers of processors than the explicitly parallelized version. The primary reason for this degradation has to do with an an $O(n^2)$ vector operation of the form

$$x := \beta_0 x + \beta_1(a+b+c+d) + \beta_2(e+f+g+h),$$

where β_0, β_1, β_2, are scalars, and x, a, b, c, d, e, f, g, and h are stored as two dimensional arrays. This operation can be explicitly parallelized as a single parallel loop. The kernel version, on the other hand, requires three $O(n^2)$ dscal's, four $O(n^2)$ dvadd's, and $4n$ $O(n)$ dvadd's. Thus we have $4n + 7$ parallel steps instead of one. The reason we need $4n$ dvadd's of length n is that the arrays containing e, f, g, and h are not conformable with the other data. Very substantial code modifications would be required to change this. An alternative approach would be to use eval_grid, since this calculation does more or less correspond to evaluating the same function at each of the grid points. But that would require considerable changes to the code as well.

JACOBI CG. The *jacobi cg* module is from ITPACK [11]. The sequential time for this module is dominated by a matrix-vector multiply in the ITPACK routine *pjac*, a subprogram that performs

Table 6: Time, speedup, and efficiency for *jacobi cg*. Sequential time is 546.4.

	Explicit			KAP			Kernel		
p	Time	Sp	Ef	Time	Sp	Ef	Time	Sp	Ef
1	542.68	1.01	101	577.97	0.95	95	584.03	0.94	94
2	273.62	2.00	100	289.64	1.89	94	294.41	1.86	93
4	139.59	3.91	98	145.81	3.75	94	147.55	3.70	93
8	71.53	7.64	95	73.93	7.39	92	75.85	7.20	90
16	37.77	14.47	90	39.51	13.83	86	40.94	13.35	83

Table 7: Time, speedup, and efficiency for *linpack spd band*. Sequential time is 653.92.

	Explicit			KAP			Kernel1			Kernel2		
p	Time	Sp	Ef	Time	Sp	Ef	Time	Sp	Ef	Time	Sp	Ef
1	690.78	0.95	95	703.00	0.93	93	704.13	0.93	93	938.29	0.70	70
2	357.83	1.83	91	375.58	1.74	87	378.19	1.73	86	488.89	1.34	67
4	189.42	3.45	86	204.96	3.19	80	209.78	3.12	78	263.86	2.48	62
8	109.80	5.96	74	122.37	5.34	67	127.89	5.11	64	151.03	4.33	54
16	79.16	8.26	52	84.92	7.70	48	93.56	6.99	44	104.17	6.28	39

one Jacobi iteration. There is also substantial time spent in other (mostly vector) operations. A total of 17 loops were explicitly parallelized. The best performance was achieved by interchanging the nested loops in *pjac*. In fact, for the sequential time reported we interchanged the loops also so that this effect would not obscure other changes due to parallelism. Table 6 shows the excellent results.

KAP and kernel versions of *jacobi cg* also perform well. There were only two loops that KAP would not parallelize that had been in our explicit version—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. Modifying *jacobi cg* to use the two-level strategy is 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 already organized into simple vector operations (e.g., dvfill, dscal). A small loss of efficiency occurs in the kernel version because a vector operation that computes a linear combination of three vectors must be implemented as a parallel dscal, followed by two parallel daxpys.

LINPACK SPD BAND. The *linpack spd band* module consists of routines *dpbfa* and *dpbsl* from LINPACK [4]. Its performance is dominated by vector dot products in the inner loop of *dpbfa*. An explicitly parallelized version of *dpbfa* is described in [19]. The data in Table 7 are from new runs. The most important features of this implementation are a reordering of the computation to compute the upper triangular Cholesky factor by rows instead of columns, and a (static) round-robin scheduling strategy that assigns the work needed to update a pivot row equally to all the processors. The triangular system solves in *dpbsl* parallelize very poorly and become a bottleneck

as the number of processors grows: 22.6 of the 653.9 seconds (3.5%) taken by the sequential version of *linpack spd band* are spent in *dpbsl*; with $p = 16$ processors the explicit parallel version spends 19.4 of 79.2 seconds (24.5%) in *dpbsl*.

A KAP version of *linpack spd band* is also described in [19]. Its performance is comparable to the explicit version, but it does require the modification to compute the triangular factor by rows instead of columns. It does not use the special scheduling strategy, but comes close by using dynamic scheduling (indicated with a directive setting the "chunksize" to one). KAP has the same lack of success with *dpbsl* as the explicit version does.

Finally, we give data for two kernel versions of *linpack spd band* in Table 7. The first version is discussed in [19] and is based on Level 2 BLAS calls. The dominant work of each step of the factorization is formulated in terms of the matrix-vector operation $y = y - A^T x$, where A is a general band matrix (BLAS routine dgbmv). 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 some degradation with respect to the explicitly parallelized code.

A block oriented version of Cholesky factorization, using Level 3 BLAS routines, is also possible. Our second kernel version uses *dpbtrf* and *dpbtrs* from LAPACK [15]. It uses parallel kernels dtrsm, dsyrk, dgemm, dtbsv, and dgbmv. The data reported here is based on a block size of 32. A block size of 16 gives slightly better performance. The primary reason for the significant extra overhead in this version is the extra computation done with zeros (see [15]). Clearly the second kernel version is not as efficient as the others on this example, although its speedup with respect to itself is slightly better than the others.

5 Discussion and conclusions

5.1 Comparing the strategies

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 kernel-based strategy. In terms of programming effort, the explicit and two-level approaches take the most effort and KAP the least. More specifically, the work required with the kernel-based approach depends very much on how the sequential code is constructed, and on whether parallel versions of the kernels themselves are available. The current definition of the "foreach" kernels, which does not allow a variable number of parameters to be passed, can cause problems and should be addressed. With all three strategies, very good knowledge of the algorithm and code is required. One might suppose that using KAP would remove this requirement, but we find that getting good performance with an automatic parallelizer still requires knowing enough of the code to be able to help the parallelizer with the important loops while ignoring others.

Regarding portability, using a tool such as KAP is the most attractive. The kernel-based approach also is quite portable, requiring only efficient implementations of the kernels on a new machine.

Regarding performance, we have seen that results for the kernel-based strategy can approach those of the explicitly parallelized code. Of the six modules tested, only two (*linpack spd band* and *hodie fft*) showed significantly better performance with the explicit versions. There will always be some extra overhead with the two-level strategy, but our experience suggests that it can be made quite small. In both of these cases, we believe that further improvements to the kernel-based algorithms will make a significant difference. The *linpack spd band* module should probably

be a kernel itself, since the most efficient approach depends strongly on the architecture [7, 21]. The automatic parallelization strategy was only successful with the two linear system solvers. The significant code modifications required to achieve good parallel performance with the more complicated discretization modules were beyond the scope of KAP, even with assistance from the user. 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.

5.2 Conclusions and future work

Our experience suggests that the two-level strategy is a good one, with a set of efficient kernels supporting the rest of the software. It seems to represent the best way to balance the conflicting goals of parallel performance, programmer effort, and portability. Some work is certainly required to modify existing software to be based cleanly on the kernels, but we find that this is comparable to the explicit parallelization, with the added benefit that you only have to do it once. From then on, as you move from machine to machine you only need to implement the kernels efficiently. The automatic strategy saves work when it works, but we find it incapable of dealing with typical complex codes.

One slight complication with the kernel-based strategy is the problem of variable granularity (see [19]). Since there are instances when a kernel that normally runs in parallel should be run sequentially (e.g., if it is in the inner loop of a much larger computation, with parallelism occurring at higher levels), the user should have the option of switching between a sequential and a parallel version of each kernel. Exactly how this should be worked out needs further attention.

In order to more completely evaluate the two-level strategy there are several obvious directions for future work. First, there are many more modules in ELLPACK which could be parallelized, not to mention investigating this approach for other packages. Secondly, there are several difficult issues such I/O, error messages, and error recovery, which have not been addressed at all in this work. A production quality parallel math software library obviously must solve these issues. Finally, portability and flexibility of the approach should be tested by implementing the kernels (and thus the modules) on other machines: first on other shared memory architectures, and more interestingly on distributed memory machines. It is important to consider the impact on these ideas of some of the issues that are unique to distributed memory environments (e.g., network topology, message passing, scalability, etc.).

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide.* SIAM, Philadelphia, PA, 1992.

[2] R. F. Boisvert. A fourth order fast direct method for the Helmholtz equation. In G. Birkhoff and A. Schoenstadt, editors, *Elliptic Problem Solvers II.* Academic Press, New York, 1983.

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

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

[5] 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.

[6] 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.

[7] M. T. Heath, E. Ng, and B. Peyton. Parallel algorithms for sparse linear systems. In *Parallel Algorithms for Matrix Computations*, pages 83–124. SIAM, Philadelphia, PA, 1990.

[8] E. N. Houstis, W. F. Mitchell, and J. R. Rice. Collocation software for second-order elliptic partial differential equations. *ACM Trans. Math. Softw.*, 11:379–412, 1985.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] R. E. Lynch and J. R. Rice. High accuracy approximations to solutions of elliptic partial differential equations. *Proc. Nat. Acad. of Sci.*, 75:2541–2544, 1978.

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

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

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

[18] 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.

[19] C. J. Ribbens, L. T. Watson, and C. deSa. Toward parallel mathematical software for elliptic partial differential equations. *ACM Trans. Math. Softw.*, 1992. to appear.

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

[21] E. Rothberg, A. Gupta, E. Ng, and B. Peyton. Parallel sparse cholesky factorization algorithms for shared-memory multiprocessor systems. In G. Richter, editor, *Advances in Computer Methods for Partial Differential Equations VII.* IMACS, 1992.

[22] A. Skjellum and C. Baldwin. The multicomputer toolbox: scalable parallel libraries for large-scale concurrent applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, 1991.

[23] R. E. Strout II, J. R. McGraw, and A. C Hindmarsh. An examination of the conversion of software to multiprocessors. *J. Parallel Distrib. Comput.*, 13:1–16, 1991.

Appendix

```
      subroutine eval_grid (f, a, lda, imin, imax, jmin, jmax)
c
c-----------------------------------------------------------------------
c  Purpose: evaluates a function at each point of a grid, returning
c           the values in a two dimensional array.
c
c  Parameters:
c     f          function to evaluate
c     a          array to return values in
c     lda        leading dimension of array a
c     imin,imax  bounds on first index
c     jmin,jmax  bounds on second index
c-----------------------------------------------------------------------
c
      double precision a(lda,*), f
      external f
c$doacross share(a,imin,imax), local(i)
      do 20 j = jmin, jmax
        do 10 i = imin, imax
          a(i,j) = f(i,j)
   10   continue
   20 continue
      return
      end
```

```
      subroutine eval_grid_int (f, a, nx, ny)
c
c-----------------------------------------------------------------------
c  Purpose: evaluates an integer function at each point of a grid,
c           returning the values in a two dimensional array.
c
c  Parameters:
c     f         function to evaluate
c     a         array to return values in
c     nx,ny     dimensions of array a
c-----------------------------------------------------------------------
c
      integer a(nx,ny), f
      external f
c$doacross share(a,nx), local(i)
      do 20 j = 1, ny
        do 10 i = 1, nx
          a(i,j) = f(i,j)
   10   continue
   20 continue
      return
      end


      subroutine foreach_hline (proc, jstart, jstop)
c
c-----------------------------------------------------------------------
c  Purpose: calls a procedure once for each horizontal grid line
c
c  Parameters:
c     proc      procedure to call
c     jstart    index of first grid line
c     jstop     index of last grid line
c-----------------------------------------------------------------------
c
      external proc
c$doacross
      do 10 j = jstart, jstop
        call proc(j)
   10 continue
      return
      end
```

```
      subroutine foreach_point (proc, nx, ny)
c
c-----------------------------------------------------------------------
c  Purpose: calls a procedure once for each grid point
c
c  Parameters:
c     proc      procedure to call
c     nx,ny     number of grid points in each direction
c-----------------------------------------------------------------------
c
      external proc
c$doacross share(nx), local(i)
      do 20 j = 1, ny
        do 10 i = 1, nx
          call proc(i,j)
   10   continue
   20 continue
      return
      end


      subroutine foreach_proc (proc, x, n)
c
c-----------------------------------------------------------------------
c  Purpose: calls a procedure once for each parallel process
c
c  Parameters:
c     proc      procedure to call
c     x         an array of workspace
c     n         dimension of x
c-----------------------------------------------------------------------
c
      double precision x(n)
      external proc
      if (m_fork(proc, x, n) .ne. 0) stop 'foreach_proc'
      return
      end
```