PARALLEL PROGRAMMING WITH MESSAGE PASSING AND DIRECTIVES

The authors discuss methods for expressing and tuning the performance of parallel programs, using two programming models in the same program: distributed and shared memory. Such methods are important for anyone who uses these large machines for parallel programs as well as for those who study combinations of the two programming models.

Parallel application developers today face the problem of how to integrate the dominant parallel processing models into one source code. Most high-performance systems use the Distributed Memory Parallel (DMP) and Shared Memory Parallel (SMP; also known as Symmetric MultiProcessor) models, and many applications can benefit from support for multiple parallelism modes. Here we show how to integrate both modes into high-performance parallel applications.

These applications have three primary goals:

1521-9615/01/$10.00 © 2001 IEEE

STEVE W. BOVA
Sandia National Laboratories
CLAY P. BRESHEARS, HENRY GABB, BOB KUHN, AND BILL MAGRO
KAI Software
RUDOLF EIGENMANN
Purdue University
GREG GAERTNER
Compaq Computer
STEFANO SALVINI
NAG
HOWARD SCOTT
Lawrence Livermore National Laboratory

- high speedup, scalable performance, and efficient system use;
- similar behavior on a wide range of platforms and easy portability between platforms; and
- low development time and uncomplicated maintenance.

Most programmers use the dominant parallel programming languages for DMP and SMP: message-passing interface[1] (MPI; www.mpi-forum.org) and OpenMP[2,3] (www.openmp.org), respectively. Some applications we study here use PVM instead of MPI (see Table 1). This article illustrates good parallel software engineering techniques for managing the complexity of using both DMP and SMP parallelism.

Applications

The applications listed in Table 1 solve problems in hydrology, computational chemistry, general science, seismic processing, aeronautics, and computational physics. Emphasizing both I/O and computation, they apply several numerical methods including finite-element analysis, wave equation integration, linear algebra subroutines, fast Fourier transforms (FFTs), filters, and a variety of PDEs (Partial Differential Equations) and ODEs (Ordinary Differential Equations). The

Table 1. OpenMP MPI applications chosen for analysis.

Application	Developers	Methods	Observations
CGWAVE	ERDC (Engineer Research and Development Center) Major Shared Resource Center	FEM (Finite Element Method) code that does MPI parameter space evaluation at the upper level with OpenMP sparse linear equation solver at lower level.	Use both MPI and OpenMP to solve the biggest problems.
GAMESS	Univ. of Iowa and Compaq	A computational chemistry application that uses MPI across the cluster and OpenMP within each SMP node.	OpenMP is generally easier to use than MPI.
Linear Algebra Study	NAG Ltd. and Albuquerque High-Performance Computing Center (AHPCC), Univ. of New Mexico	Study of hybrid parallelism using MPI and OpenMP for matrix-matrix multiplication and QR factorization	MPI provides scalability while OpenMP provides load balancing.
SPECseis95	ARCO and Purdue Univ.	Scalable seismic benchmark with MPI or OpenMP used at the same level. It can compare an SMP to a DMP.	A scalable program will scale in either MPI or OpenMP.
TLNS3D	NASA Langley	CFD (Computational Fluid Dynamics) application that uses MPI to parallelize across grids and OpenMP to parallelize each grid.	Some applications need to configure the size of an SMP node flexibly.
CRETIN	Lawrence Livermore National Laboratory (LLNL)	Non-LTE (non-Local Thermodynamic Equilibrium) physics application that has multiple levels of DMP and SMP parallelism.	Restructuring for DMP helps SMP. Tools help find deep bugs.

sidebar, "A Taxonomy of Parallelism," characterizes these applications in terms of how they use parallelism and the complexity of interactions among the different techniques used.

DMP and SMP modes can combine in two ways. The application could have two parallel loop levels where, for the outer loop, each DMP process executes an inner loop as an SMP multithreaded program. This applies also where a single original loop has enough iterations to be split into two nested loops, the outer DMP and the inner SMP. The second, less frequent method is for an application to have one level of parallelism mapped to either DMP or SMP. Using both types interchangeably increases portability and lets each type crosscheck the other to isolate parallelism problems.

Developers frequently ask whether OpenMP or MPI is faster. Most of these applications can run with one MPI process and several OpenMP threads, or several MPI processes and one OpenMP thread. However, except for SPECseis, the results are not directly comparable because MPI applies best to coarser-grained parallelism, which has less overhead, whereas OpenMP applies best to fine-grained parallelism. Because OpenMP parallelism uses SMP hardware, it can run finer-granularity parallelism and still appear to perform as efficiently.

The results for SPECseis, where OpenMP has been substituted for MPI at the same level, show comparable performance between the two models. Some might argue that SPECseis doesn't use "true OpenMP style" because it replaces message-passing primitives moving data between private memory areas with copies to and from a shared buffer. This might seem less efficient than placing all data accessed by multiple processors into shared memory, but using shared memory is not clearly better—using private memory can enhance locality of reference and hence scalability.

We present performance results for four of the applications running on several platforms, including Compaq Alpha, HP, IBM, SGI, and Sun. Compaq, HP, and IBM systems have a cluster SMP architecture, which is suitable for program structures with DMP parallelism on the outside and SMP parallelism inside. Although SGI and Sun machines are SMPs, SGI is a NUMA (Non-Uniform Memory Access) architecture whereas Sun is UMA, each with specific performance implications.

CGWAVE

Work supported by the US Department of Defense High-Performance Computing Modernization Program used MPI and OpenMP simultaneously to dramatically improve the performance of a harbor analysis code. This code models wave motions from Ponce Inlet,[4] an area on Florida's Atlantic coast notorious for patches of rough water that capsize boats as they enter or leave the inlet. The advanced programming techniques demonstrated here reduced calculation times from more than six months to less than 72 hours. Reducing calculation times makes modeling larger bodies of water, such as an entire coastline, feasible.

The DoD now uses CGWAVE, developed at

A Taxonomy of Parallelism

Characterizing the applications studied here helps us understand their different uses of parallelism and the complexity of interactions among them. We classify the applications across two dimensions, one describing the message-passing parallelism and the other describing the directive parallelism.

We break message passing into three classes:

- Parametric—This type has a very coarse-grained outer loop: Each parallel task differs only in a few parameters, communicated at task start. We expect high efficiency and performance scalable to the number of parametric cases. This is easy to implement on DMP systems.
- Structured domains—When algorithms perform neighborhood communications, such as in finite difference computations, efficient DMP execution relies on domain decomposition, which can be broken into structured and unstructured grids. Structured domain decomposition is simpler because data can be decomposed by data structures. None of the applications studied here have unstructured domains, which are more difficult to implement. Domain decomposition leads to higher communication than parametric parallelism.
- DMP direct solvers—Linear algebra solvers can be direct or iterative. Implementing direct solvers on DMP systems is complex because they have very high communication needs and require load balancing. For efficiency's sake, linear systems should be large, usually greater than 2,000 unknowns, to warrant using a DMP model.

We can also roughly characterize the parallel SMP structure into three types, analogous in complexity to the DMP types listed earlier:

- Statically scheduled parallel loops—Directive methods permit easy parallelization of applications that contain either a large single loop calling many subroutines or (less efficient) a series of simple loops.
- Parallel regions—Coordinating scheduling and data structure access among a series of parallel loops increases the efficiency of directive-based applications. On SMP systems, the benefits are reduced scheduling and barrier overhead and better use of locality between loops. Merging all loops into one parallel region is conceptually similar to domain decomposition. Iterative solvers, for example, consist of a parallel region with several loops.
- Dynamic load balanced—In some applications, static scheduling results in gross inefficiency. Load balancing schemes are more complex to implement than parallel regions, which have a fixed assignment of tasks to processors. The direct solvers studied here require this technique because the complex data flow leads to irregular task sizes.

Table A shows the structural characteristics of our applications following this classification. Two applications, GAMESS and CRETIN, consist of multiple significantly different phases and appear in two positions.

Table A. Message-passing versus directive complexity in applications studied.

Message-passing directives	Parametric	Structured domains	DMP direct solvers
Statically-scheduled parallel loops	GAMESS (Integration)	CRETIN (Transport)	
Parallel regions	CGWAVE	TLNS3D, SPECseis	
Dynamic load balanced		CRETIN (Kinetics)	NAG GAMESS (Factor)

Parallel Vector Adds: OpenMP versus MPI–OpenMPI

Figure A shows a simple vector add computation in OpenMP form and in MPI–OpenMP form. The OpenMP form is just the serial form with a parallel directive indicating that the loop can be executed in parallel. The combined MPI–OpenMP form simply scatters sections of the input vectors `b` and `c` from MPI task 0 to all the others. Each MPI task enters the `parallel do`, which divides each section among a set of threads for each MPI task. Then MPI gathers the result sections from each MPI task back together on MPI task 0.

```
      SUBROUTINE vadd(a,b,c,n)
      DIMENSION a(n),b(n),c(n)
!$OMP PARALLEL DO
      DO i = 1,n
       a(i) = b(i) + c(i)
      ENDDO
!$OMP END PARALLEL DO
      RETURN
      END
```
(a)

```
      SUBROUTINE vadd(a, b, c, n)
      DIMENSION a(n),b(n),c(n)
      DIMENSION aloc(n),bloc(n),cloc(n)
      CALL MPI_Init(ierr)
!!             Get an identifier for each MPI task and the number of tasks
      CALL MPI_Comm_Rank(MPI_COMM_WORLD,mytask,ierr)
      CALL MPI_Comm_Size(MPI_COMM_WORLD,ntasks,ierr)
!!             Divide up and send sections of b and c to each task from task 0
      isize = n/ntasks
      CALL MPI_Scatter(b,isize,MPI_REAL,bloc,isize,MPI_REAL,0,MPI_COMM_WORLD,ierr)
      CALL MPI_Scatter(c,isize,MPI_REAL,cloc,isize,MPI_REAL,0,MPI_COMM_WORLD,ierr)
!!             Now divide up each section among available threads
!$OMP PARALLEL DO
      DO i = 1,isize
       aloc(i) = bloc(i) + cloc(i)
      ENDDO
```
(b)

Figure A. Parallel vector adds. (a) Open MP version. If directives are ignored, you must have the serial version. (b) MPI–Open MP version.

the ERDC Coastal and Hydraulics Laboratory, daily to forecast and analyze harbor conditions.[5] CGWAVE also assists with determining safe navigation channels, identifying safe docking and mooring areas, and insurance underwriting.

CGWAVE looks for harbor response solutions within a parameter space. It exploits MPI at a coarse-grained level to perform several simulation runs, with different parameters, in parallel. At each point of the parameter space under consideration, CGWAVE uses OpenMP to parallelize compute-intense portions of the simulation. Combining these two programming paradigms increases how much parameter space we can explore within a reasonable amount of time. This dual-level parallel code can help coastal analysts solve problems previously considered unsolvable.

Parallel execution scheme

CGWAVE characterizes the sea state as several incident wave components defined by period, amplitude, and direction. We see this set of wave components as a parameter space: Each triplet leads to a separate partial differential equation to be solved on the finite element grid, and the parallelism applied uses MPI to distribute the work. Because the execution time of separate wave components might differ by a factor of four or more, a simple boss–worker strategy dynamically balances the workload. Each component calculation approximates the harbor's known depth and shape within a finite-element model that leads to a large, sparse linear system of more than 100,000 simultaneous equations. The solver is parallelized with OpenMP to return the answer to any one component more quickly.

We performed our CGWAVE simulations simultaneously using computers in two different locations: at the ERDC Major Shared Resource Center (MSRC) and the Aeronautical Systems Center MSRC in Dayton, Ohio. We used SGI Origin 2000 platforms, which have shared memory required for the dual-level parallelism. To

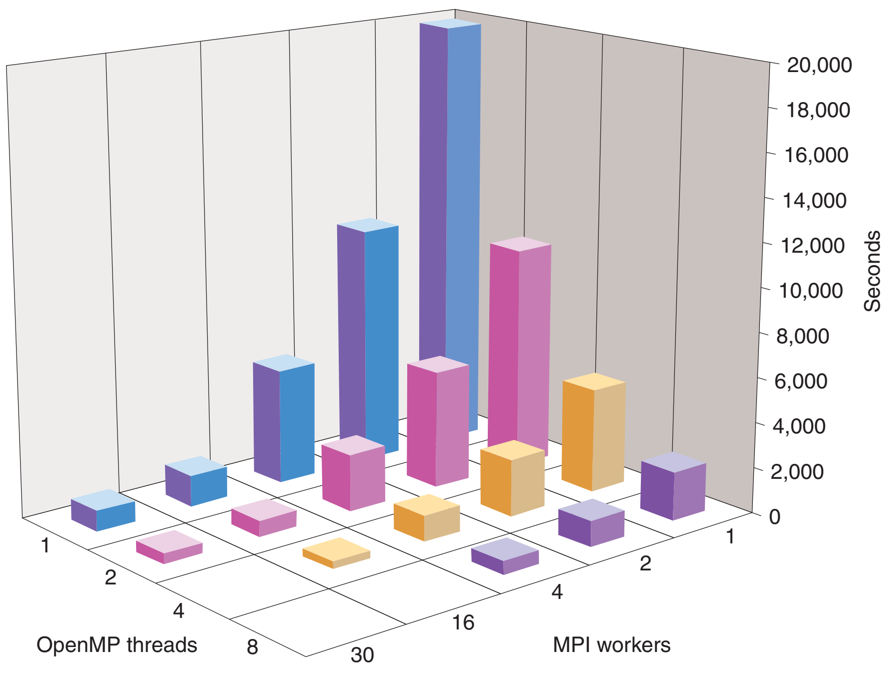


Figure 1. Wall-clock time (in seconds) for CGWAVE analysis of a sediment mound (50,000 elements, 75 components) on an SGI Origin 2000 platform.

couple the geographically separate SGI systems, we used MPI_Connect.[6]

Programming issues

Calling MPI_Init or some other MPI communication routines from within an OpenMP parallel region causes severe problems in an MPI–OpenMP program, and we recommend avoiding such calls. Parallelizing the CGWAVE code avoids these problems by allowing only one thread to execute message-passing operations.

Distributing wave components among MPI processes is highly parallel. Except for a combination step at the end, no communication occurs between the MPI worker processes but only between boss and worker processes. The OpenMP threads of one process therefore do not interfere with those of another because each wave component is independent and each MPI process has local memory.

To achieve scalability in the solver on the SGI Origin 2000 NUMA architecture, the program must assign data needed by the conjugate gradient solver to the processor that will use it most. CGWAVE takes advantage of the first-touch rule to transparently distribute data. Specifically, data resides with the processor that first touches it. Important arrays are initialized in parallel such that the processor initializing the data will also execute the compute-intensive work on that data.

Parallel software engineering

Because CGWAVE is a production code, we don't want to modify the code extensively. Portability is also important.

Our work here represents the first parallelization of CGWAVE for SMP. To prepare the code for SMP parallelism, we declared work arrays in common blocks locally in the conjugate gradient subroutine, eliminating array access synchronization and coherency overhead. This was our only modification to the original source. We used the KAP–Pro Toolset (Kuck and Associates, www.kai.com) for parallel assurance testing (Assure) and performance optimization (GuideView).

Domain decomposition using MPI at the finite-element-solver level would require extensive code modification, whereas distributing the wave components with MPI requires minimal changes: the original serial code simply loops over the wave components. Instead of using the loop iteration value to select the next wave component, we used MPI send–receive calls to request, receive, and distribute wave components. A Fortran 90 module houses the boss–worker subroutines that coordinate component dispatch and acquisition.

Preprocessor directives control conditional compilation, which makes it easy to compile the original serial, OpenMP, MPI, or MPI–OpenMP version of CGWAVE from the same source code.

Performance

We ran several simulations using a range of MPI processes and OpenMP threads. To test portability, we ran the code on the ERDC MSRC SGI Origin 2000, Cray T3E, and IBM SP (the latter two under MPI only). We used an academic test problem (an underwater sediment mound caused by dredging) to measure performance and verify the parallel code's numerical accuracy.

Figure 1 shows the relative performance for different numbers of MPI and OpenMP workers computing the test problem. For Ponce Inlet simulations, we used a grid size of 235,000 elements and almost 300 wave components. Without parallelism, this simulation would require six months of CPU time, conservatively. Sixty processors reduced this to about three days. We conducted timings on a 112-CPU SGI Origin 2000.

Although we're tempted to draw conclusions about the relative performance of MPI and OpenMP, it's not appropriate to compare these models because we applied each to a different level of parallelism in CGWAVE. Running multiple MPI processes, each executing multiple OpenMP threads, yielded the best overall performance. We must note that the boss–worker dynamic load balancing breaks down as the number of MPI worker processes approaches the number of wave components in the test system. On a 100-CPU machine, for example, using 100 MPI workers to perform a 100-component harbor simulation is inefficient due to inappropriate load balance. It would be more efficient to have 25 MPI workers create four OpenMP threads for each assigned wave component.

GAMESS

The Hartree-Fock self-consistent field (SCF) approach to computing wave functions using a finite set of Gaussian-type functions has been central to ab initio quantum chemistry since computers became powerful enough to tackle polyatomic molecules. GAMESS-US,[7] part of the SPEChpc[8] suite, offers a well-studied example of such computation.

Considerable research[9–11] has explored how parallel processing could benefit ab initio computation. GAMESS (General Atomic and Molecular Electronics Structure System) is a large and complex software system, its 70,000 lines of code consisting of several relatively autonomous modules written and extended by many people over many years. Several modules in the GAMESS release we explore here have been restructured to exploit both DMP and SMP parallelism.

Parallel execution scheme

Figure 2 shows the software layers used in most ab initio computation, illustrated here using Global Arrays (www.emsl.pnl.gov), a library for distributed array operations in chemistry applications. The application software layer implements ab initio computations in GAMESS. The next layer consists of application-specific parallel operations and is the first to support parallel processing concepts. The third layer, portable parallel processing, shows two alternatives: PVM, a message-passing library, and OpenMP, directive-based parallel processing. (As with many message-passing applications, the GAMESS message-passing layer is referred to as PVM, but a common subset of message-passing primitives permits porting to MPI.) A program can use both models at different levels of parallelism. The lowest layer visible to a programmer is that bound to a particular operating system. In this case, PVM calls the Memory Channel API while OpenMP calls Digital Unix Threads operations.

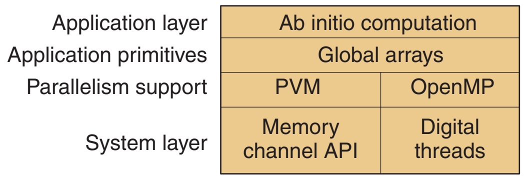


Figure 2. The parallel programming model for ab initio computation.

GAMESS uses different parallelism models for two application primitives—building the Fock matrix with electron orbital integral evaluation and, our focus here, solving the Fock matrix.

We can apply both OpenMP and PVM at the application primitives layer, which consists of selected linear algebra solvers for ab initio chemistry. Because these solvers are communication-intense algorithms that need dynamic scheduling, OpenMP works better than message passing except for very large computational chemistry problems. We therefore implemented these solvers in both PVM and OpenMP to let users select the parallelism mode according to computation size.

Parallel software engineering

Solving the Fock matrix requires several linear algebra operations. In a message-passing system, each processor typically computes some portion of the operation with a local array, then performs global reduction message-passing. The code requires fairly complex modifications to a simple solver, as Figure 3a illustrates for the matrix operation from the GAMESS message-passing version. The color coding highlights the sources of this version's additional complexity:

- Some of the code switches parallel mode on or off. Figure 3b, which uses the same color scheme, shows that OpenMP does this with one directive.
- Some of the code implements scheduling options. In OpenMP, a directive calls out

```
    DO 310 J = 1,M,MXCOLS
        JJMAX = MIN(M,J+MXCOLS-1)
 C      ----- GO PARALLEL! -----
        IF(PARR) THEN
          IF (NXT) THEN
          L2CNT = L2CNT + 1
          IF (L2CNT.GT.NEXT) NEXT=NXTVAL(NPROC)
          IF (NEXT.NE.L2CNT) THEN
           DO 010 JJ=J,JJMAX
             CALL VCLR(V(1,JJ),1,N)
 010         CONTINUE
           GO TO 310
          END IF
        ELSE
          IPCOUNT = IPCOUNT + 1
          IF (MOD(IPCOUNT,NPROC).NE.0) THEN
           DO 020 JJ=J,JJMAX
             CALL VCLR(V(1,JJ),1,N)
 020         CONTINUE
           GO TO 310
          END IF
          END IF
        END IF
        DO 300 JJ=J,JJMAX
         DO 100 I = 1,N
          W = DDOT(M,Q(I,1),LDQV,V(1,JJ),1)
          IF(ABS(W).LT.SMALL) W=ZERO
          V(I,JJ)=W
 100      CONTINUE
 300     CONTINUE
 310    CONTINUE
        IF(PARR) THEN
         IF(LDQV.GT.N) THEN
          NP1 = N + 1
           DO 420 I=NP1,LDQV
            DO 410 J=1,M
             V(I,J) = ZERO
 410         CONTINUE
 420        CONTINUE
           END IF
           CALL MY_MPI_REDUCE(V,LDQV*M,MPI_SUM)
           IF(NXT) NEXT = NXTVAL(-NPROC)
         END IF
```

(a)

```
c$omp parallel if ((N .gt. 100) .and. (M .gt.50))
c$omp& shared (V,Q,WRK,M,N,LDQV)
c$omp& private (j,jj,jjmax,i,w,wrkloc)
c$omp do schedule(dynamic)
        DO 310 J = 1,M,MXCOLS
         JJMAX = MIN(M,J+MXCOLS-1)
         DO 300 JJ=J,JJMAX
          DO 100 I = 1,N
           W = DDOT(M,Q(I,1),LDQV,V(1,JJ),1)
           IF(ABS(W).LT.SMALL) W=ZERO
           WRKloc(I)=W
100        CONTINUE
          DO 200 I = 1,N
           V(I,JJ) = WRKloc(I)
200        CONTINUE
300       CONTINUE
310      CONTINUE
c$omp end do nowait
c$omp end parallel
```

(b)

Color coding:

🟪 Parallel / serial switch
🟧 Dynamic scheduling
🟨 Block scheduling
🟩 Matrix kernel
🟦 Merge global matrix

Figure 3. GAMESS solver, (a) message-passing version and (b) directive version.

the scheduling option from a library.

- Some of the code sets array elements used for the local computation (the V array), which must be properly initialized. Elements not touched are set to zero so that the global reduction operation (the call to MY_MPI_REDUCE) correctly combines all processors' parts. The OpenMP version accesses the shared array directly.

MPI also incurs computational overhead because data must be

1. passed from a matrix in GAMESS to a system buffer,
2. passed from the system buffer across the network to a destination system buffer, and
3. copied from this buffer to the GAMESS matrix again.

This process must repeat several times for each global reduction operation, hence the arrays involved must be very large before parallel processing offers any benefit.

Figure 3b shows the same Fock matrix operations computed with OpenMP directives. Not only does efficiency increase, but the code is much simpler to write, maintain, and debug.

Performance

Table 2 shows how parallel GAMESS performs on a cluster of four Compaq Alpha 8400 systems with eight EV5 Alpha processors on each system connected by a Memory Channel. The dataset is SPEC "medium." The 32-processor cluster completes operations five times faster than the four-processor cluster.

Table 2. Cluster of four 8-processor systems on GAMESS.

Number of CPUs	Elapsed time (sec)	Cluster speedup
4	327	1
8	178	1.84
16	101	3.24
24	76	4.30
32	64	5.11

Linear algebra study

As part of their ongoing collaboration, Numerical Algorithms Group (NAG) and the Albuquerque High Performance Computer Center (AHPCC) analyzed the feasibility of mixed-mode parallelism on a Model F50-based IBM SP system.[12] Each SMP node in this system has four Model F50 processors connected by an enhanced bus. The project aimed to

- study the techniques required to achieve high performance from a single processor to a single SMP node up to multiple SMP nodes;
- develop an approach that incorporates both DMP and SMP parallelism and would allow the same code to be used for pure DMP, at one extreme, to pure SMP, at the other, with hybrid processors in between; and
- make use of emerging or established standards, such as OpenMP and MPI, using only standard directives and MPI functions to ensure portability across different platforms.

The IBM SP system employs POSIX threads, whose thread-safe MPI libraries allow the coexistence of DMP and SMP parallelism. In this work, hybrid parallelism means using the SMP mode *within* each node and explicit message passing *across* nodes. The study also explored communication cost hiding and the feasibility of dynamic load balancing within the node with directive parallelism.

Test cases involved matrix–matrix multiplication and QR factorization, both common linear algebra operations. The approach used showed good scalability and performance in the hybrid mode for both. We discuss the results of the QR factorization study here; the full measurements appear elsewhere.[12]

Why message passing and directives?

Message-passing parallelism is difficult because both functional and data parallelism must be considered. It also, however, provides a more flexible framework than directives in that the parallelism need not fit into specific parallel control constructs. Also, at least in principle, DMP offers extremely high scalability.

OpenMP directive parallelism is much simpler to use. The SMP hardware does data placement automatically—explicit data placement sometimes enhances performance but is not required. Also, within an SMP node, directive parallelism uses shared memory more efficiently and facilitates dynamic load balancing. Scaling to many SMP processors requires using directives at the same level as message passing, however, where these benefits disappear. Little data exists on applications using directives at this level.

We see parallelism as a continuum between the limits of pure message passing and pure directives. Code should then execute anywhere along that continuum depending on the number of processors per node. The ideal code could move seamlessly along the continuum with no modifications, with the parallelism mode selected dynamically at runtime.

Message-passing paradigms have particular difficulty with dynamic load balancing: Any approach based on data migration across nodes entails significant communication cost and code complexity. For a fixed problem size, as node quantity increases, computation time decreases while communication costs and load imbalance increase.

Hybrid parallelism can increase message-passing efficiency: if N_t is the total number of processors and N_{smp} the number of processors per node, message passing would only occur between N_t / N_{smp} communicating entities. Communication costs and overhead would approach those of a smaller message-passing system and load imbalance would decrease. Introducing communication within the extent of dynamic load balancing would also permit overlapping communication and computation within each SMP node, reducing communication costs by up to a factor N_{smp}. For example, on the IBM SP used, communication costs could be reduced by up to 75 percent.

Parallel execution scheme

In the matrix multiply case, $C = aA^TB + bC$, we

Table 3. QR-factorization performance for mixed MPI–OpenMP code on various cluster configurations.

Dynamic versions ($N_n \times N_{smp}$)

	1×1		1×4		2×4		4×4	
n	Hide	No hide	Hide	No hide	Hide	No hide	Hide	No hide
500	218	208	611	494	656	618	628	
1,000	225	229	732	678	1,128	912	1,231	1,131
2,000			746	773	1,310	1,185	1,963	1,579
4,000							2,467	2,124

Adaptive Versions ($N_n \times N_{smp}$)

	1×1		1×4		2×4		4×4	
n	Hide	No hide	Hide	No hide	Hide	No hide	Hide	No hide
1,000	229		700		1,225		1,796	
2,000			713		1,409		2,507	
4,000							2,758	

achieved perfect load balance by partitioning *B* and *C* into equal column-blocks across nodes. We distributed the matrix *A* block-cyclically by columns. Each *A* column block was broadcast to all nodes and then used for partial products, accumulating the results in *C*.

We also used communication hiding (overlapping communication with computation), and the performance results, discussed further later, highlight its importance. We placed the communications outside the parallel region and then inside the parallel region, then used OpenMP directives requiring the dynamic scheduling of a DO loop to hide communication cost by using broadcast of the *A* column-block as one special DO loop iteration. We also tried an adaptive load-balancing scheme that subdivided matrices *B* and *C* into column blocks, one for each processor in the SMP node. The processor performing the communication accessed a narrower column block, its width determined by ad hoc cost parameters.

The QR-factorization communication pattern resembles that of matrix–matrix multiplication except that perfect load balance isn't possible without data movement across SMP nodes. However, our approach allowed good local load balancing—that is, over the data stored within each SMP node. We used a block version of the QR factorization algorithm functionally similar to that of ScaLapack. As with matrix–matrix multiplication, the adaptive strategy performed better than dynamic DO loop scheduling.

Programming issues

When we found that the IBM XL Fortran compiler release we used did not completely implement OpenMP directives—specifically, synchronization directives—we explicitly coded a Fortran barrier subroutine, which might have impacted performance.

Also, the need to communicate information about the message-passing parallelism (number of nodes, data distribution, and so on) across the subroutine interfaces increased code complexity. This makes a hybrid application less portable to a serial or SMP system.

Performance

For the matrix multiply, we tested modest 2,000-square matrices. Table 3 shows the QR-factorization performance results for various cluster configurations, communication hiding or not, and dynamic versus adaptive strategies. The performance data appear in megaflops, measured using system *wall clock* time. *Hide* and *No hide* refer to communication hiding. We tested the configurations $N_n \times N_{smp}$ where N_n represents the number of nodes and N_{smp} the number of processors per node. For comparison, Stefano Salvini and his colleagues[12] reported 1,060 Mflops for the NAG library F68AEF and measured 743 megaflops with the Lapack routine DGEQRF in the 1×4 configuration (pure four-way SMP), for $n = 2000$.

Without communication hiding, communication accounted for 15 to 20 percent of the elapsed time. Communication hiding recovered 75 percent of that time. On four 4-way Model F50 nodes, communication hiding and adaptive load balancing achieved nearly a threefold speedup—encouraging, because we spent little time optimizing these codes.

SPECseis

SPECseis, a seismic processing benchmark used by the Standard Performance Evaluation Corporation (SPEC, www.spec.org), typifies modern seismic processing programs used for oil and gas exploration.[8] It consists of 20,000 lines of Fortran and uses C primarily to interface with the OS. FFTs and finite-difference solvers comprise the main algorithms in the 240 Fortran and 119 C subroutines. The benchmark includes five data sets—the smallest runs in half an hour at 100 Mflops and uses 110 Mbytes of temporary disk space; the largest runs in 240 hours at 100 Mflops and uses 93 Gbytes of disk space.

The benchmark originally used message passing, but growing use of high-performance SMP and NUMA systems has increased the need for realistic benchmarks to test these systems. We also wanted to test whether scalable parallelism is inherent to the programming model—message passing versus directives—or the application itself.

Parallel execution scheme

The code has serial and parallel variants. The parallel variant comes in two very similar forms: message-passing PVM or MPI and directive OpenMP, the latter developed from the former. SPECseis uses an SPMD execution scheme with consecutive computation and communication phases, separated by barrier synchronization.

The message-passing variant starts directly in SPMD mode; that is, all processes start executing the same program. During initialization, executed by the master processor only, the other processes are waiting explicitly. By contrast, the OpenMP variant starts in sequential mode before opening an SPMD parallel section that encompasses the rest of the program.

Both versions keep all data local to the processes and partitioned about equally. PVM or MPI programs always keep data local to the processes, whereas OpenMP programs give explicit locality attributes (default is shared). The only data elements declared shared in SPECseis are interprocess data exchange regions used similarly to message-passing library implementations on SMP systems. The sending thread copies data to the shared buffer, and the receiving thread copies it into its local data space. We could improve this scheme by allocating in shared memory all data for communication, but we have not yet done this in SPECseis.

Programming issues

Several issues arose when we converted the SPECseis message-passing variant to OpenMP.

- Data privatization: All data except for communication buffers was declared private to all threads, making the memory required for a data set equal to that for the message-passing version. OpenMP's syntactic forms allowed us to easily privatize individual data elements, arrays, and entire common blocks. Switching the default for all data to private achieves the same effect.
- Broadcasting common blocks: OpenMP permits broadcasting common blocks that have been initialized before a parallel region to all private copies of the common block inside the parallel region. SPECseis used this extensively. OpenMP requires specifying this broadcast statically and explicitly. SPECseis doesn't require copying all common block data into the parallel regions, but our OpenMP variant does so, incurring slight additional execution overhead.
- Mixing C and Fortran: We used a Fortran OpenMP compiler only because OpenMP for C was not yet available. However, because all calls to C routines occur within parallel regions, performance did not degrade. This raised two issues. First, C global data was difficult to privatize; we had to make an explicit copy for each thread. OpenMP C features would have let us simply give a private attribute to these data. Second, although OpenMP requires OS calls to be thread-safe, C programs compiled with a non-OpenMP-aware compiler don't have this property. We had to use OS-specific mechanisms to guarantee a thread-safe program—for example, placing a critical section around memory allocation.
- Thread binding: OpenMP does not require that processes be bound to specific processors, nor does it provide mechanisms to specify this execution attribute. Again, we had to use OS-specific mechanisms, where available, to achieve this.

OpenMP provides all necessary constructs for scalable SPMD-style programming. Although this is only one case study, we found OpenMP easy to learn, and it even obviated several problems that proved time-consuming in our message-passing experiments. For example, we didn't need to set up configuration files or message-passing daemons.

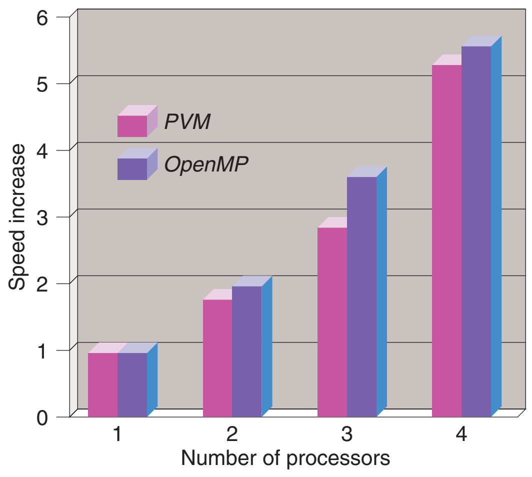


Figure 4. SPECseis speedup comparison for PVM and OpenMP.

Performance

Message passing might seem more scalable than directive parallelism, but our message-passing and OpenMP variants achieve equal scalability using the same parallelization scheme with the same data partitioning and high-level parallelism. Figure 4 shows results obtained on an SGI Power Challenge. Although the OpenMP variant runs slightly faster that the PVM version, this slight difference disappears if we increase the data set size. We attribute this to the higher message-passing costs for exchanging small data sections.

TLNS3D

TLNS3D, developed at NASA Langley, is a thin-layer Navier-Stokes solver used in computational fluid dynamics analyses. The program can handle models composed of multiple blocks connected with boundary conditions. The code, written in Fortran 77, is portable to major Unix platforms and Window NT.

To simplify complex object flow modeling, typical input data sets contain multiple blocks, motivated by the physical model's geometry. You can compute these blocks concurrently and use MPI to divide the blocks into groups and assign each group to a processor. The block assignment is static during the run because the program must create distinct data files for each worker.

This approach works well for models that use significantly more blocks than workers, because TLNS3D can generally group the blocks such that each MPI worker does roughly the same amount of work. Unfortunately, as the number of MPI workers increases, the potential for static load balancing diminishes, eventually reaching a one-to-one mapping of blocks into groups. Load balancing then becomes impossible, which limits the best-case parallel speedup to the number of blocks. Splitting large blocks before the run could increase their number, but modifying blocks becomes harder and the simplified numerical methods at the block level can impact the results.

Parallel execution scheme

At the block level of parallelism, TLNS3D uses a boss–worker model in which the boss performs I/O and the workers do numerical computations. The boss also acts like a worker. A run consists of multiple iterations, and at the end of each, the MPI workers exchange boundary data.

To address the MPI level of parallelism's limitations, we added OpenMP directives to exploit parallelism within each block. Each block appears as a 3D grid, and most computations on that grid take the form of loops that can be performed in parallel. Because TLNS3D has many such parallel loops, we carefully tuned the OpenMP directives to maximize cache affinity between loops and eliminate unnecessary global synchronization among threads.

With single-block data sets, OpenMP parallelism allows scalable parallel performance improvements on up to 10 processors, whereas the MPI version achieved zero speedup. The mixed parallel version, however, provides the ability to load balance cases when the number of CPUs approaches or even exceeds the number of blocks.

TLNS3D achieves load balance by first partitioning the blocks across MPI workers to achieve the best possible static load balance. Next, it partitions a group of threads, matching the number of CPUs to be used, among blocks such that the average number of grid points per thread is approximately equal. For example, a block containing 60,000 grid points would have roughly twice as many threads at its disposal as a block containing 30,000 grid points. This nonuniform thread allocation achieves a second form of load balancing, which is effective for runs on a very large number of processors.

Programming issues

Given many CPUs and two available parallelism levels, the question arises of how many MPI processes to use. Generally, we want to minimize load imbalance while also minimizing communication and synchronization. Finding the correct balance is complicated, however, because load imbalance arises as MPI processes in-

```
subroutine top
!$OMP PARALLEL
allocate ( … )
!$OMP DO
do i = 1,N_lines
call line_transfer
enddo
call diagnostics_0
!$OMP END PARALLEL
```

```
subroutine line_transfer
do while
!$OMP DO
do i = 1,N_angles
…
call mesh_sweep
…
enddo
call diagnostics_1
call accelerate
enddo
```

```
subroutine mesh_sweep
do i = 1,N_zones
do j = 1,N_energies
…
enddo
do j = 1,N_energies
…
enddo
M
enddo
call diagnostics_2
```

Figure 5. Structure of radiation transport in CRETIN. (The code is shown in a simplified version, with the OpenMP IF clauses omitted for clarity.)

crease, and synchronization overhead grows as OpenMP threads proliferate within each block.

One solution is increasing the number of MPI processes until a load imbalance begins to develop, then using OpenMP threads to achieve additional speedup and reduce the remaining load imbalance. This doesn't require modifying the code or data set.

The nonuniform mapping of threads to block groups in TLNS3D makes this application most effective on shared-memory systems, such as the SGI Origin 2000. Other systems, such as the IBM SP, severely limit your ability to use varying numbers of threads per MPI worker. The mixed-parallel approach, however, can be effective on these machines if you limit the number of MPI workers to maintain good static load balance. You can then assign a fixed number of threads to each SMP node for additional speedup.

Parallel software engineering

We found parallel processing tools very effective in analyzing TLNS3D. On the MPI side, VAMPIR and VAMPIRTRACE programs from Pallas (www.pallas.com) analyze the message-passing performance to identify where delays occur. This permits identifying block load imbalance. On the OpenMP side, GuideView from KAI identifies OpenMP performance problems. We also used its Assure tool to find shared variables needing synchronization. This is important when converting programs to directives because shared variables can be touched anywhere.

CRETIN

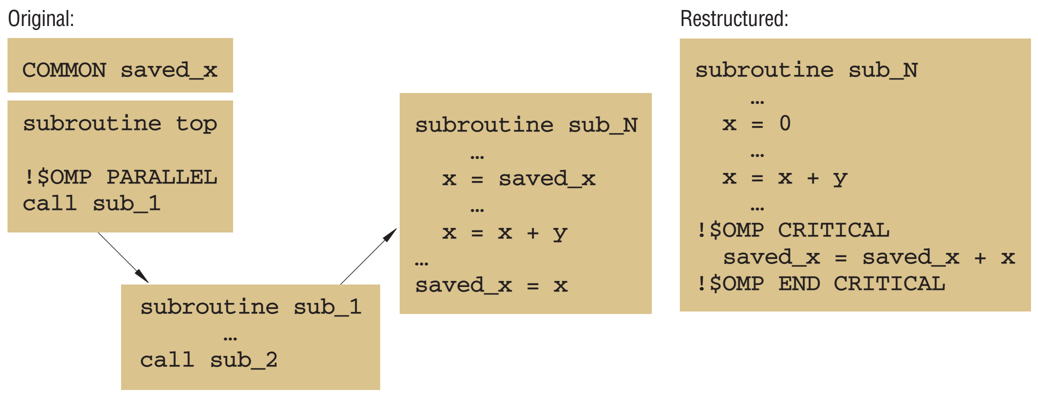
CRETIN is a non-LTE physics application developed at Lawrence Livermore National Laboratory over the past 10 years by one developer. It uses roughly 100,000 lines of Fortran. Compared to many LLNL application packages, it is moderately complex. Smaller and younger codes use more recent software technology, but older, larger, more complex codes developed by physicists must be migrated from vector architectures to newer DMP and SMP parallelism available on ASCI (Accelerated Strategic Computing Initiative) systems.

The Department of Energy ASCI project uses state-of-the-art parallel processing systems to achieve the project's goals: to simulate much larger physical models than previously feasible. The ASCI blue systems, combined DMP–SMP systems, require mixed MPI–OpenMP programming. Using portable MPI and OpenMP, we have ported CRETIN to two ASCI blue systems:

- Blue Pacific—an IBM SP2 system with 1,464 clusters of four-processor PowerPC 604e nodes, and
- Blue Mountain—an SGI Origin 2000 with 48 clusters of 128 processors each.

Parallel execution scheme

Several CRETIN computation packages or modes have a high degree of parallelism. We discuss two: Atomic Kinetics and Line Radiation Transport. Atomic Kinetics is a multiple-zone computation with massive amounts of work in each zone. We can map the loop over zones to either message-passing or directive parallelism. Radiation Transport has potentially several levels of parallelism: lines, directions, and energies. The kernel of the computation is a mesh sweep across the zones. We used several nested parallel regions with OpenMP IF clauses to select the best parallelism level for the problem and the algorithm selected for the run. On the message-passing level the boss performs memory allocation and passes zones to the workers. Figure 5 shows part of Radiation Transport's structure.



```
COMMON saved_x
```

```
subroutine top

!$OMP PARALLEL
call sub_1
```

```
subroutine sub_1
      …
call sub_2
```

```
subroutine sub_N
    …
  x = saved_x
    …
  x = x + y
…
saved_x = x
```

```
subroutine sub_N
    …
  x = 0
    …
  x = x + y
    …
!$OMP CRITICAL
  saved_x = saved_x + x
!$OMP END CRITICAL
```

Figure 6. CRETIN deep bug found with Assure.

Programming issues

In Atomic Kinetics, the amount of work in the zones varies over five orders of magnitude, making load balancing of zones a critical issue. On each time step, we used the computational time for each zone to sort and assign zones to processors for the next step. In the DMP version, we restructured data to use temporary zones to avoid referencing the zone index. This took considerable work but aided data localization for the SMP version as well. Minor restructuring helped localize and minimize communication points for message passing. Here, too, the SMP version benefited with fewer and larger critical sections, making the directive version easy to develop.

We also needed to classify common blocks as shared or thread private and move some private variables from the master initialization code to inside the parallel region before the worksharing `parallel do`.

Line Radiation Transport offers two DMP parallelization options:

- Each processor can transport different lines over the entire mesh, which requires transposing data from the distributed zones. This code is not needed in the serial or SMP version. The SMP version uses the same data structures as the serial version with array accesses interleaved to different caches. This option executes very efficiently but is limited by memory requirements.
- Larger problems can apply domain decomposition. The data transpose is not needed, but load balancing might suffer because the zone distribution across processors might not be efficient for the Atomic Kinetics. The transport algorithm becomes less efficient and requires more iterations for more domains. SMP parallelization helps significantly here by decreasing the number of domains for a given number of processors.

Both options allocated thread private workspaces for temporary space. Minor restructuring to minimize critical sections increased efficiency of the lowest parallelism levels.

Multiple models for dynamic memory are available—Fortran77, Fortran90, and Cray pointers—but make portable memory management tricky. Thread private allocations are performed using the pointers stored in a thread private common block. The scope of thread private data spans several parallel regions. The pointers remain valid as the application moves from parallel region to parallel region. So the application relies on the thread private common retaining these pointers between parallel regions. Although it's a subtle feature of OpenMP that all implementations must implement correctly, it is superior to adding a thread-specific index to all array references.

Parallel software engineering

When designing the application, we spent most of the parallelism work (five months) on the DMP level. We spent just one month on the SMP parallelization, primarily learning how to use an SMP efficiently. However, restructuring the DMP parallelism helped organize the parallel structure for easy conversion to an efficient SMP version.

Getting the parallelism mostly working was relatively quick; finding deep bugs took longer. We uncovered most of the deep bugs with the Assure tool for OpenMP. Figure 6 shows an example scenario for a deep bug. Several levels deep in a parallel region, the programmer made

Table 4. Performance and programming issues in applications that use message passing and directives.

		CGWAVE	GAMESS	Linear algebra study
Why use directives and message passing?		Add performance needed to attack another dimension in problem solving	Flexible use of SMP clusters on problems with lots of parallelism	To get message-passing scalability and good load balancing with directives
Parallelism	**Message passing**	Boss–worker applied to wave parameter space	Outer coarser grain parallel loop	Block solve matrix system with fixed distribution
	Directives	Sparse solver applied to PDE	Inner finer grain more variable size	Dynamic or adaptive scheduling of block solution
Platforms		Multiple SGI O2000	Memory Channel AlphaServer 8400	SP2 with F50 nodes
Problems		Calling message-passing routines in parallel regions	Small granularity in MPI; thread private efficiency of OpenMP	Couldn't use MPI within node; incomplete support for OpenMP
Parallel software engineering		Used Assure to explore OpenMP parallelism	OpenMP versions are sometimes much simpler	Porting and maintaining two levels is difficult

		SPECseis	TLNS3D	CRETIN
Why use directives and message passing?		Provide benchmark portable to DMP and SMP systems	Assignment of grid blocks left poor load balance for MPI	To solve much larger problems with leading-edge computers
Parallelism	**Message passing**	SPMD: Compute, barrier, communicate, then repeat	Group of grid blocks assigned to each worker by boss	Uses explicit data transpose, not needed in SMP
	Directives	Same parallelism but built with different model	More or fewer processors assigned to each worker	Benefited from previous DMP parallelization
Platforms		SGI, Sun	SGI O2000	IBM SP2, SGI O2000
Problems		Setting up message-passing configuration; thread safety of libraries	Need for flexible clustering of processors to SMP nodes	Storage allocation pointers tricky
Parallel software engineering		Emulating message passing in directives	One expert for MPI, another for OpenMP	Used OpenMP tools to find deep bugs

a local copy of a shared common variable, updated it, and stored it back in the common, but forgot a critical section around the update. Also, the local variable is initialized to zero rather than to the `saved_x`, such that updating `saved_x` can be done in one atomic operation. This makes the critical section more efficient.

Finally, we made every effort to keep all parallelism consistent with a single source to simplify maintenance, debugging, and testing. We achieved this, with the exception of data transpose, mentioned earlier, and find it greatly benefits ongoing development.

All six applications described here successfully use both message-passing and directive parallel models to improve program performance. Most use multiple levels of parallelism, with the coarse grain using DMP and the finer grain using SMP. An application can, however, have only one level of coarse-grained, domain decomposition parallelism mapped to both message passing and directive version in the same code without performance degradation. Table 4 summarizes our experiences.

Running applications at the highest performance levels possible remains a challenge that requires experimentation to find the critical parameters and optimal coding for each system. In testing leading-edge parallel processing technology, we found that understanding the semantics and implementation of each programming model presented the primary obstacle. Once we deciphered these, we found it rather easy to successfully apply the models to the many code situations. We hope our experiences will migrate to a daily production environment.

Acknowledgments

This work was supported in part by the US Department of Energy under contract no. W-7405-Eng-48 and by the National Science Foundation under grant no. 9703180. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect sponsors' views.

References

1. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI, Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, Mass., 1994.
2. R. Chandra et al., *Parallel Programming in OpenMP*, Morgan Kaufmann, San Francisco, 2000.
3. L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Eng.*, vol. 4, no. 1, Jan.–Mar. 1998, pp. 46–55.
4. S.W. Bova et al., "Dual-level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP," *Int'l J. High-Performance Computing Applications*, vol. 14, no. 1, Spring 2000, pp. 49–64.
5. B. Bingyi Xu, V. Panchang, and Z. Demirbilek, "Exterior Reflections in Elliptic Harbor Wave Models," *J. Waterway, Port, Coastal, and Ocean Eng.*, vol. 122, no. 3, May/June 1996, pp. 118–126.
6. G.E. Fagg and J.J. Dongarra, *PVMPI: An Integration of the PVM and MPI Systems*, tech. report UT-CS-96-328, Univ. of Tennessee, Knoxville, Tenn., May 1996.
7. M.W. Schmidt et al., "General Atomic and Molecular Electronic Structure System," *J. Comp. Chem.*, vol. 14, no. 11, Nov. 1993, pp. 1347–1363.
8. R. Eigenmann and S. Hassanzadeh, "Benchmarking with Real Industrial Applications: The SPEC High-Performance Group," *IEEE Computational Science & Eng.*, vol. 3, no. 1, Spring 1996, pp. 18–23.
9. R.J. Harrison and R. Shepard, "Ab Initio Molecular Electronic Structure on Parallel Computers," *Ann. Rev. Phys. Chem.*, vol. 45, 1994, pp. 623–658.
10. D. Feller, R.A. Kendall, and M.J. Brightman, *The EMSL Ab Initio Methods Benchmark Report*, tech. report PNNL-10481, Pacific Northwest National Laboratory, Richland, Wash., Mar. 1995.
11. B. Kuhn and E. Stahlberg, "Porting Scientific Software to Intel SMPs Under Windows/NT," *Scientific Computing & Automation*, vol. 14, no. 12, Nov. 1997, pp. 31–38.
12. S. Salvini, B.T. Smith, and J. Greenfield, *Towards Mixed-Mode Parallelism on the New Model F50-Based IBM SP System*, Tech. Report AHPCC98-003, Albuquerque High Performance Computing Center, Univ. of New Mexico, Sept. 1998.

Steve W. Bova works in the Thermal/Fluid Computational Engineering Sciences Group at Sandia National Laboratories. His research interests include parallel computing and adaptive finite-element methods for systems of parabolic and hyperbolic differential equations. He received his PhD from the University of Texas. Contact him at Sandia National Laboratories, P.O. Box 5800, MS 0835, Albuquerque, NM 87185-0835; swbova@sandia.gov.

Clay P. Breshears is a member of the Parallel Applications Center at KAI Software, a division of Intel Americas. His research interests include threading, cryptography, and numerical methods on parallel systems. He received a PhD in computer science from the University of Tennessee, Knoxville. Contact him at KAI Software, 1906 Fox Dr., Champaign, IL 61820; clay.breshears@intel.com.

Henry Gabb is a member of the Parallel Applications Center at KAI Software, where he works on parallel performance tuning for commercial and research applications. He holds a PhD from the University of Alabama at Birmingham. Contact him at KAI Software, 1906 Fox Dr., Champaign, IL 61820; henry.gabb@intel.com.

Bob Kuhn is the manager of new products at KAI Software. His current research and development areas include creating and refining MPI/OpenMP performance analysis tools, pushing the frontiers of OpenMP, and improving peer-to-peer computing. He has a PhD from the University of Illinois at Urbana-Champaign. Contact him at KAI Software, 1906 Fox Dr., Champaign, IL 61820; bob.kuhn@intel.com.

Bill Magro is the manager of the Parallel Applications Center at KAI Software. His current work focuses on parallelizing applications with OpenMP, MPI, and other parallelism models. He also is a product manager for the KAI tools for parallelizing applications. He has a PhD in physics from the University of Illinois at Urbana-

Champaign. Contact him at KAI Software, 1906 Fox Dr., Champaign, IL 61820; bill.magro@intel.com.

Rudolf Eigenmann is an associate professor and chairman of the Computer Area at the Purdue University School of Electrical and Computer Engineering. His research interests include optimizing compilers, programming methodologies and tools, and performance evaluation for high-performance computers. He received his PhD in electrical engineering/computer science from ETH Zurich, Switzerland. He received a 1997 NSF Career award and serves on the editorial boards of the *International Journal of Parallel Programming* and *Computing in Science & Engineering*. He has also served as the chairman and vice-chairman of the High-Performance Group of the Standard Performance Evaluation Corp. (SPEC). He is a member of the IEEE and the ACM. Contact him at Purdue Univ., School of Electrical and Computer Eng., 1285 EE Bldg., West Lafayette, IN 47907; eigenman@purdue.edu.

Greg Gaertner is a member of the Systems Quality and Performance Engineering Group at Compaq. His research interests include high-performance server architectures and applications. He also chairs the SPEC High Performance Group. He received a BS in physics from the University of Minnesota. Contact him at greg.gaertner@compaq.com.

Stefano Salvini is the High Performance Group Lead and Senior Technical Consultant for the Numerical Algorithms Group. He has a PhD in applied mathematics from Queens University in Belfast. His interests include sparse iterative solvers, SMP, and hybrid parallelism. Contact him at The Numerical Algorithms Group, Ltd. Wilkinson House, Jordan Hill Rd., Oxford OX2 8DR, United Kingdom; stef.salvini@nag.co.uk.

Howard Scott is a computational physicist at Lawrence Livermore National Laboratory, where he researches nonlocal thermodynamic equilibrium radiation transport, radiation/hydrodynamics simulations, and parallel processing. He has a PhD in astrophysics from Cornell University. Contact him at Lawrence Livermore National Laboratory, P.O. Box 808, L-18, Livermore, CA 94551; hascott@llnl.gov.