Parallel Programming with Message Passing and Directives

- by -
Steve W. Bova, swbova@sandia.gov, Sandia National Laboratories,
Clay P. Breshears, Henry Gabb, Bob Kuhn, Bill Magro, magro@kai.com,
KAI Software, A Division of Intel Americas, Inc.
Rudolf Eigenmann, eigenman@ecn.purdue.edu, Purdue University
Greg Gaertner, ggg@zko.dec.com, Compaq Computer
Stefano Salvini, stef@nag.co.uk, NAG Ltd
Howard Scott, hascott@llnl.gov, Lawrence Livermore National Laboratory

.

Abstract

This paper discusses methods for expressing and tuning the performance of parallel programs, by using two programming models in the same program: distributed and shared memory. Case studies show how mixing these two approaches results in efficient machine use because the two models match the two levels of parallelism present in the architecture of current SMP clusters. This issue is important for anyone who uses these large machines for parallel programs as well as for those who study combinations of the two programming models.

Introduction

One problem that developers of parallel applications face today is how to combine into one source code the dominant models for parallel processing. Most high performance systems are both Distributed Memory Parallel (DMP) and Shared Memory Parallel (SMP, Note, sometimes this acronym stands for Symmetric MultiProcessor; they are interchangeable here). For many applications there are several reasons why one should support multiple modes. In this paper we show how to integrate both modes into an application. For the most part, this paper uses the dominant parallel programming languages for DMP and SMP. These are MPI(www.mpi-forum.org, [Grop94]) and OpenMP(www.openmp.org, [Chan00], [Dagu98]) respectively. However, a couple of the applications studied also use PVM instead of MPI.

We will illustrate good parallel software engineering techniques for managing the complexity of using both DMP and SMP parallelism. The goals that these applications are trying to achieve are:

- High parallel performance – That is: high speedup, scalable performance, and efficient use of the system.
- Portability – The application should behave the same way on a wide range of platforms and porting the software between platforms should be easy.
- Low Development Time – Developing and maintaining the software should be easy.

Parallel programming tools and parallel software engineering are widely used to leverage programmer productivity. In this paper we will illustrate using good tools and engineering as key techniques for managing the complexity of DMP and SMP parallel applications.

Several applications were selected for this paper. Table 1 gives an overview.

Table 1. Overview of OpenMP MPI Applications Analyzed in this Paper.

Application	Developer	Methods Used	Major Observation
CGWAVE	ERDC MSRC	FEM code which does MPI parameter space evaluation at the upper level with OpenMP sparse linear equation	Use both MPI and OpenMP to solve the biggest problems.

		solver at lower level	
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 AHPCC, UNM	Study of hybrid parallelism using MPI and OpenMP for matrix-matrix multiplication and QR factorization	MPI provides scalability level while OpenMP provides load balancing.
SPEC-Seis95	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 application which 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	LLNL	Non-LTE physics application which has multiple levels of DMP and SMP parallelism.	Restructuring for DMP helps SMP. Tools help find deep bugs.

There are several applications areas represented: Hydrology, Computational Chemistry, General Science, Seismic Processing, Aeronautics, and Computational Physics. A wide variety of numerical methods are represented also: finite element analysis, wave equation integration, linear algebra subroutines, FFTs, filters, and a variety of PDEs and ODEs. Both I/O and computation are stressed by these applications.

This survey also describes several parallelism modes. DMP and SMP modes can combine in a program in two ways. First, the application could have two parallel loop levels. For the outer loop, each DMP process can execute an inner loop as an SMP multithreaded program. This includes the case where a single original loop had enough iterations to be split into two nested loops, the outer DMP and the inner SMP. Second, an application can have a single level of parallelism that can be mapped to either DMP or SMP. The second way is less frequent than the first in this set of applications. But using both types interchangeably increases portability and allows each type to cross check the other to help isolate parallelism problems.

One question that is frequently asked is, "Which is faster -- OpenMP or MPI?" Most of these applications can be run with one MPI process and some number of OpenMP threads or the same number of MPI processes and one OpenMP thread. However, except for SPECseis the results are not directly comparable because MPI is used for coarser grain parallelism where there is less overhead than for the fine grain parallelism where OpenMP is applied. 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, indicate that the performance is comparable. It could be argued that "true OpenMP style" has not been used here. In SPECseis, message passing primitives moving data between private memory areas has simply been replaced with copies to and from a shared buffer. This may be less efficient than placing all data that is accessed by multiple processors into shared memory. However, using shared memory is not clearly better because using private memory can enhance locality of reference and hence scalability.

The platforms used by these applications include systems from Compaq Alpha, HP, IBM, SGI, and Sun. Performance results are presented for several of these systems on four of the applications. Compaq, HP, and IBM systems have a cluster of SMPs architecture which is suitable for program structures with DMP parallelism on the outside and SMP parallelism inside. While SGI and Sun machines are both SMPs, the former is a NUMA (Non-Uniform Memory Access) architecture while the latter is an UMA. We will discuss performance implications of these two architectures.

What Types of Parallelism Are in These Applications

First we will characterize the applications studied to understand the types of parallelism and the complexity of interactions among them. This characterization is not a general taxonomy but simply a mechanism to

allow the user to understand the range of applications discussed here. The applications are classified in two dimensions, one describing the message passing parallelism and one describing the directive parallelism.

For the purpose of this paper we break the message passing dimension of our applications into three types:

- **Parametric** – Very coarse grained outer loop. Each parallel task differs only in a few parameters that need to be communicated at task start. We expect the efficiency to be very high and the performance to be scalable to the number of parametric cases. They are easy to implement on DMP systems.
- **Structured Domains** – When algorithms perform neighborhood communications, such as in finite difference computations, domain decomposition is critical for efficient DMP execution. Domain decomposition can be broken into two cases, structured grids and unstructured grids. Structured domain decomposition is the simpler case because data can be decomposed by data structures. None of the applications studied here have unstructured domains, which would be more difficult to implement. Domain decomposition leads to higher communication than parametric parallelism.
- **DMP Direct Solvers** – Linear algebra solvers can be divided into direct and iterative. Implementing direct solvers on DMP systems is more complex. They can have very high communication needs and require load balancing. Because of efficiency considerations, linear systems must be large, usually greater than 2000 unknowns, to warrant using a DMP model.

The parallel structure of the directive applications here can also be roughly characterized into three types, analogous in complexity to the DMP types above:

- **Statically-Scheduled Parallel Loops** – Applications that contain either a large single loop calling many subroutines or a series of simple loops are easy to parallelize with directive methods. Efficiency in the former case is usually higher than the latter case.
- **Parallel Regions** – To increase the efficiency of directive-based applications, it is useful to coordinate scheduling and data structure access among a series of parallel loops. On SMP systems the benefits are reduced scheduling and barrier overhead, and better utilization of locality between loops. If one merges all the loops in an application into one parallel region, it is conceptually similar to domain decomposition mentioned above. Iterative solvers, for example, consist of a parallel region with several loops.
- **Dynamic Load Balanced** – In some applications static scheduling represents a serious source of 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 are one example requiring this technique, due to the complex data flow leading to irregular task sizes.

The following table shows the structural characteristics of our applications in terms of the above classification. Note that two of the applications, GAMESS and CRETIN, consist of multiple significantly different phases and are shown in two positions.

Table 2. Message Passing versus Directive complexity in applications studied.

Directives \ Message Passing	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)

CGWAVE

Application Overview

MPI and OpenMP were simultaneously used to dramatically improve the performance of a harbor analysis code. This work was supported by the Department of Defense (DoD) High Performance Computing Modernization Program. The project "Dual-level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP" won the "Most Effective Engineering Methodology" award for the SC98 HPC Challenge competition modeling wave motions from Ponce Inlet on Florida's Atlantic coast. [BoBr99] (Ponce Inlet is notorious for patches of rough water, which have capsized boats as they enter or leave the inlet.) The advanced programming techniques in the demonstration reduced calculation times by orders of magnitude - in this case from over 6 months to less than 72 hours. Reducing calculation times makes modeling larger bodies of water, such as an entire coastline, feasible.

CGWAVE, developed at the WES Coastal and Hydraulics Laboratory, is a code used daily by the DoD for forecasting and analyzing harbor conditions [XuPa96]. Other applications of CGWAVE include determining safe navigation channels, identifying safe docking and mooring areas, and insurance underwriting.

Why Message Passing and Directives?

CGWAVE attempts to find harbor response solutions within a parameter space. MPI Parallelism is exploited at a coarse-grain level. Several simulation runs, with different simulation parameters, are performed in parallel. Within the simulation at each point of the parameter space under consideration, OpenMP is used to parallelize compute-intense portions of the simulation. The combination of these two programming paradigms can increase the size of the parameter space that may be explored within a reasonable amount of time. Using this dual-level parallel code, coastal analysts can now solve problems that they could not consider solving before.

Parallel Execution Scheme

The sea state is characterized by a number of incident wave components, which are defined by period, amplitude, and direction. This set of wave components may be regarded as a parameter space: each triplet leads to a separate partial differential equation to be solved on the finite element grid. The parallelism that can be exploited on that parameter space uses MPI to distribute the work. Because the execution time of separate wave components may differ by as much as a factor of four, a simple boss-worker strategy dynamically balances the workload. For each component calculation, the known depth and shape of the harbor are approximated within a finite element model that leads to a large, sparse linear system of over 100,000 simultaneous equations. The solver is parallelized with OpenMP to return the answer to any one component more quickly.

The CGWAVE simulations were performed using computers in two different locations - at the ERDC Major Shared Resource Center (MSRC) and at the Aeronautical Systems Center MSRC in Dayton, Ohio - simultaneously. SGI Origin 2000 platforms were used which have shared memory required for the dual-level parallelism. To couple the geographically separate SGI systems MPI_Connect [FaDo96] was used.

Programming Issues

Severe problems occur in an MPI/OpenMP program if MPI_Init or some other MPI communication routines are called from within an OpenMP parallel region. The parallelization of the CGWAVE code was

able to avoid these potential programming problems by allowing only one thread to execute message passing operations. It is recommended to avoid MPI calls within OpenMP parallel regions.

Distributing wave components among MPI processes is highly parallel. Except for a combination step at the end, there is no communication between the MPI worker processes. Communication only occurs between the boss and worker processes. Therefore, the OpenMP threads of one process do not interfere with those of another since 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, data required by the conjugate gradient solver must be assigned to the processor which 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 them. So, important arrays are initialized in parallel in such a way that the processor initializing the data will be the processor to execute the compute intensive work on the data.

Parallel Software Engineering

Because CGWAVE is a production code, extensive code modification is undesirable. Portability is also an important design consideration.

The work reported here on the solver was the first time that CGWAVE has been parallelized for SMP. To prepare the code for SMP parallelism, work arrays in COMMON blocks were declared locally in the conjugate gradient subroutine, which eliminated array access synchronization and coherency overhead. This was the only modification to the original source. The KAP/Pro Toolset (Kuck & Associates, Inc. [http://www.kai.com]) was used for parallel assurance testing (Assure) and performance optimization (GuideView).

Domain decomposition using MPI at the finite element solver level would require extensive code modification. Distributing the wave components with MPI requires minimal code changes. The original serial code simply loops over the wave components. Use of the loop iteration value to select the next wave component was replaced by MPI send/receive calls to request, receive, and distribute wave components. The boss-worker subroutines used to coordinate the dispatch and acquisition of components are housed in a Fortran 90 module.

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

Performance

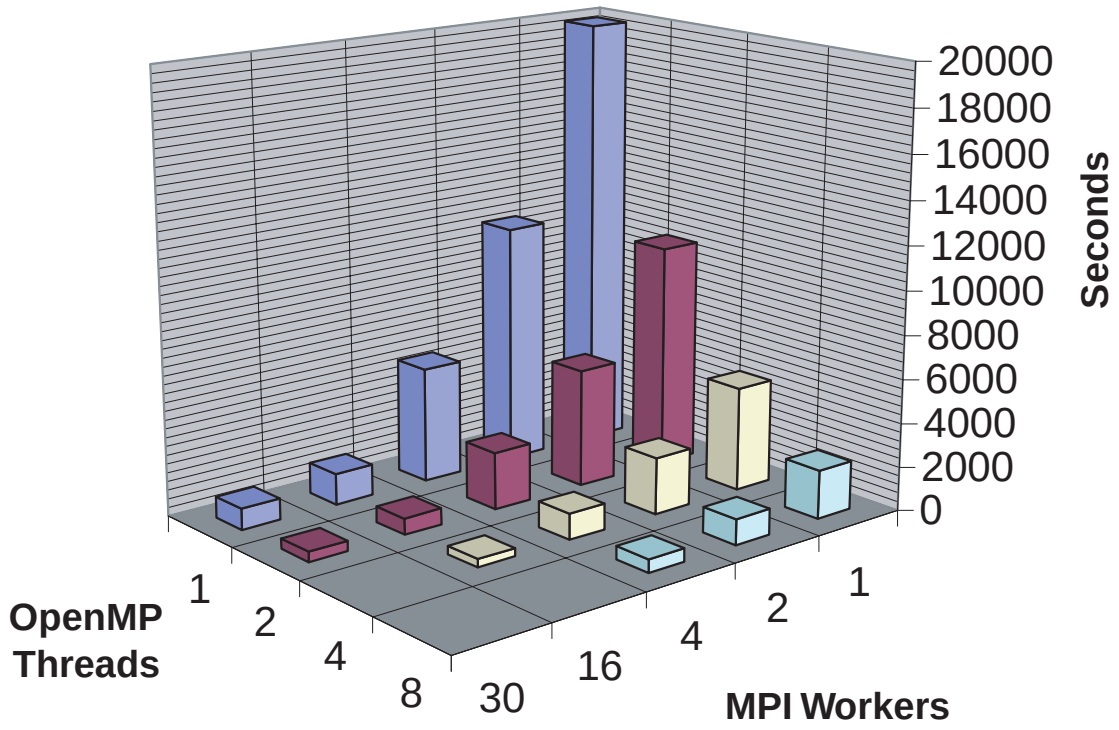
Several simulations were run with a range of MPI processes and OpenMP threads. Since portability is an important design consideration, the code was tested on the ERDC MSRC SGI Origin 2000, CRAY T3E, and IBM SP (the latter two platforms were run under MPI only). An academic test problem (an underwater sediment mound caused by dredging) was used to measure performance and to verify that the parallel code is numerically accurate.

Figure 1 shows relative performance for different numbers of MPI and OpenMP workers computing the test problem. For simulations of Ponce Inlet, a grid size of 235,000 elements and almost 300 wave components was used. A conservative estimate of the required CPU time for this simulation is six months if parallelism is not used. On 60 processors, actual time was reduced to about 3 days. Timings were done on a 112-CPU SGI Origin 2000.

Although it is tempting to draw a conclusion about the relative performance of MPI and OpenMP, because each model was applied to a different level of parallelism in CGWAVE, comparison of these two models is not appropriate. Running multiple MPI processes, each executing multiple OpenMP threads, yielded the best overall performance on the test problem. It is important to note, although not shown here, 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 much more efficient to have 25 MPI workers create four OpenMP threads for each assigned wave component.

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



GAMESS

Application Overview

The traditional Hartree-Fock self-consistent field (SCF) approach to computing wave functions using a finite basis set of Gaussian-type functions has been mainstay of ab initio quantum chemistry since digital computers first became sufficiently powerful to tackle polyatomic molecules. GAMESS-US [Schm92] is a well studied example of this type of computation. GAMESS, as its full name (General Atomic and Molecular Electronics Structure System) suggests, is a large and complex software system. It contains about 70K lines of code, consisting of a number of relatively autonomous modules written and extended by many people over the course of many years. Nonetheless, a number of modules in the release of GAMESS on which this work is based were restructured to exploit both DMP and SMP parallelism. GAMESS is also part of the SPEChpc[EiHa96] suite. The use of parallel processing in ab initio computation has been the subject of considerable research [HaSh94] [FeKe95] [KuSt97].

Why Message Passing And Directives?

A key reason that both DMP and SMP were targeted is illustrated by the following table. As an example of current parallel processing technology, consider the key points of Compaq's Alpha Server architecture. A typical clustered system would consist of AlphaServer 8400 and AlphaServer 4100 servers, connected together with a MEMORY CHANNEL. Typical parallel system software would include the KAP/Pro Toolset and Digital MPI libraries from Compaq to support directives and message passing. [http://www.dec.com/info/hpc/hpc.html.]

Table 3 compares MEMORY CHANNEL to a typical SMP Bus and a typical network. We see that in principle we can get 5 times more performance by using the SMP bus than MEMORY CHANNEL where possible.

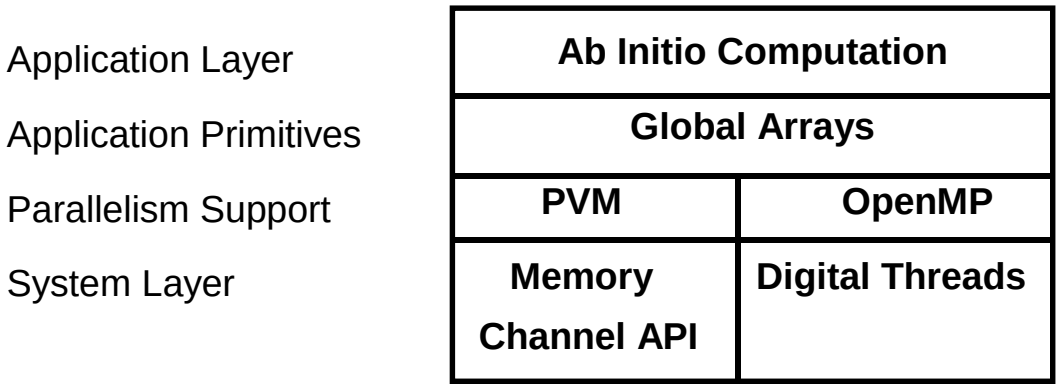
Table 3. SMP bus, MEMORY CHANNEL , and network characteristics.

Measurement	SMP Bus	MEMORY CHANNEL	Network
Latency in transferring one word (Microsec)	~ 0.5	< 5	400-500
CPU Overhead processing protocol if any (Microsec)	< 1	< 1	~ 200
Maximum messages per second (millions)	> 10	> 2	< 0.01
Bandwidth (MB per second)	> 500	> 100	~ 10

Parallel Execution Scheme

Figure 2 shows the layers of software that are used in most ab initio computation. The top layer, the application software layer, implements the various ab initio computations in GAMESS. The next layer consists of application-specific parallel operations. In the figure we cite, as an example, Global Arrays, a library for distributed array operations in Chemistry applications [http://www.emsl.pnl.gov.]. This layer is the first layer to support concepts of parallel processing. The third layer is the portable parallel processing layer. Here two alternatives are contrast: PVM, a message passing library, and OpenMP, directive-based parallel processing. (As with many message passing applications, the message passing layer of GAMESS is referred to as PVM but a common subset of message passing primitives is used which allow it to be ported to MPI as well.) A program can use both of these models in the same program 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. Parallel Programming Model for Ab Initio Computation.



Both models of parallelism have been applied at the application software layer and at the application primitives layer. We will discuss the two application primitives where GAMESS uses parallelism: building the Fock matrix with electron orbital integral evaluation and solving the Fock matrix. They have different parallelism models. Here only solving the Fock matrix is considered.

Figure 3. Message Passing Version of GAMESS Solver.

```
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
```

Color Coding:

- Parallel / Serial Switch
- Dynamic Scheduling
- Block Scheduling
- Matrix Kernel
- Merge Global Matrix

Figure 4. Directive Version of GAMESS Solver.

```
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
```

Both OpenMP and PVM were applied at the application primitives layer which consists of selected linear algebra solvers for ab initio chemistry. Because direct linear algebra solvers are communication intense algorithms with significant need for dynamic scheduling, message passing is less appropriate than OpenMP except for very large computational chemistry problems. So, two versions of these solvers, in PVM and in OpenMP, were implemented. This allows the user to select the mode of parallelism according to the size of the computation. We contrast the difference between these two implementations in the next subsection.

Parallel Software Engineering

Solving the Fock matrix requires several linear algebra operations. The standard way to do linear algebra operations in message passing is to have each processor compute some portion of the linear algebra operation with a local array. Then global reduction message passing operations are performed. The code requires fairly complex modifications to a simple solver as seen in the matrix operation from the message passing version of GAMESS in Figure 3. As the color coding indicates, the additional complexity of the MPI version stems from several factors:

- Some of the code switches parallel mode on or off. OpenMP does that with one directive. (The same color is used in Figure 4.)
- Some of the code implements scheduling options. In OpenMP, the user calls out the scheduling option from a library in a directive.
- Finally, some of the code sets elements of the array used for the local computation (the V array) which need to be properly initialized. And, elements that are not touched are set to zero, so that the global reduction operation (the call to MY_MPI_REDUCE) combines the parts from all processors correctly. In contrast, the OpenMP version accesses the shared array directly.

In addition to code complexity, computational overhead is incurred because data must be passed in three stages: first, from a matrix in GAMESS to a system buffer; then from a system buffer across the interconnection network to a system buffer at a destination; and finally, copied from system buffer to GAMESS matrix again. This process must be repeated several times for each global reduction operation. This means that the arrays involved must be very large before any benefit is gained from parallel processing.

However, SMP systems achieve considerable benefit from parallel processing on Fock Matrix operations much more simply. The same routine with OpenMP directives is shown in Figure 4. Not only is the efficiency increased but the code is much simpler to write, maintain, and debug.

Performance
Table 4 shows the performance of parallel GAMESS on a cluster of 4 Compaq Alpha 8400 systems with 8 EV5 Alpha processors on each system connected by a MEMORY CHANNEL. The dataset is SPEC"medium". Speedup relative to the 4 processor time range of over 5 is shown for the 32 processor time.

Table 4. Cluster of four 8 processor systems on GAMESS.

Number CPUs	Elapsed Time (secs)	Cluster Speedup
4	327	1
8	178	1.84
16	101	3.24
24	76	4.3
32	64	5.11

Linear Algebra Study

Application Overview
As part of their ongoing collaboration, NAG Ltd. and the Albuquerque High Performance Computer Center (AHPCC) analyzed the feasibility of mixed mode parallelism on a Model F50-based IBM SP system [SaSm98]. Each SMP node in this system has 4 Model F50 processors connected by an enhanced bus. The project aimed to:

- Study the techniques required to achieve high performance across the parallel system from a single processor, to a single SMP node, up to multiple SMP nodes.
- Work towards the development of an approach to hybrid parallelism that aims to build on existing technology, incorporating aspects of both DMP and SMP parallelism. Moreover, such an approach should allow the same code to be used for pure DMP, at one extreme, to pure SMP parallelism, at the other extreme, with all the intermediate stages of a hybrid collection of processors.
- Make use of emerging or established standards, such as OpenMP and MPI. In order to ensure portability across different platforms, only standard directives and MPI functions were used.

The IBM SP system employs POSIX Threads, and their MPI libraries are thread-safe which allows the coexistence of both modes of parallelism, DMP and SMP. In the context of this work hybrid parallelism meant using the SMP mode *within* each node, and explicit message passing *across* nodes. The study also

considered the hiding of communication costs and the feasibility of dynamic load balancing within the node via directive parallelism.

As test cases, matrix-matrix multiplication and QR factorization were studied, both common linear algebra operations. For both cases, the approach used showed good scalability and performance in the hybrid mode. In this paper we discuss the results of the QR factorization study. The full measurements are presented in [SaSm98].

Why Message Passing and Directives?

Message passing parallelism is "difficult" in the sense that both functional and data parallelism must be considered. However, currently, message passing provides a more flexible framework for parallelism than directives in the sense that the parallelism need not fit into specific parallel control constructs. And, at least in principle, DMP offers scalability up to very large numbers of processors. A considerable body of software using message passing has been produced. OpenMP directive parallelism is much simpler to use. The user does not need to deal with issues of data placement. Instead, it is done automatically by the SMP hardware. Explicit data placement can sometimes enhance performance, but it is not a prerequisite. In addition, within an SMP node, directive parallelism utilizes the shared memory more efficiently and dynamic load balancing is less difficult to achieve. To achieve scalability to large numbers of SMP processors, it is necessary to use directives at the same level as message passing where these benefits of may be lost and other effects may occur. There is very little data about applications using directives at this level. (See SPECseis below for some data on this case.)

Perhaps, parallelism could be viewed as a continuum between the limits of pure message passing and pure directives. In this context, codes should be written to allow for positioning execution anywhere along the continuum as determined by the number of processors per node. Ideally, no modifications to the code should be required to move seamlessly along the continuum, with the mode of parallelism selected dynamically at runtime.

Dynamic load balancing is particularly difficult for message passing paradigms: any approach based on migration of data across nodes would entail significant communication cost and code complexity. As the number of nodes increases, for a fixed problem size, computation time decreases, whilst communication costs and load imbalance both tend to increase. Message passing efficiency can be increased with hybrid parallelism: 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. In other words, communication costs and overheads would be comparable to those of a *smaller* message passing system. Load imbalance would also be reduced. Furthermore, by introducing communication within the extent of dynamic load balancing, communication and computation could be overlapped 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%.

Parallel Execution Scheme

In the matrix multiply case, $C = aA^TB + bC$, perfect load balance was achieved across nodes by partitioning B and C into equal column-blocks across nodes. The matrix A was distributed block-cyclically by columns. Each column-block of A was broadcast to all nodes and then used for partial products, accumulating the results in C.

Communication hiding (overlapping communication with computation) was also used and the performance results highlight its importance. This was measured quite simply by first placing the communications outside the parallel region and then inside the parallel region. OpenMP directives requiring the dynamic scheduling of a DO loop could be simply used to hide communication cost by using broadcast of the column-block of A as one special iteration of the DO loop. Alternatively, an adaptive load balancing scheme was used, where the matrices B and C were subdivided into column-blocks, one for each processor in the SMP node. The column block accessed by the processor performing the communication was narrower than the others, its width determined by a set of ad hoc cost parameters.

The communication pattern in the QR-factorization is similar to that of the matrix-matrix multiplication, but load balancing characteristics are rather different. In particular, perfect load balance cannot be achieved without data movement across SMP nodes. However, the approach followed here allowed good *local* load balancing, i.e. over the data stored *within* each SMP node. A block version of the QR factorization algorithm was used, functionally similar to that of ScaLAPACK. As in the case of the matrix-matrix multiplication, the adaptive strategy performed better than using dynamic scheduling of DO loops. In any case, communication cost hiding was found to have a significant impact on the performance achievable.

Programming Issues

Two programming issues came up. First, at the directive level, the OpenMP implementation was not complete in the release of the IBM XL Fortran compiler used. In particular, synchronization directives were not implemented. We explicitly coded a Fortran barrier subroutine, which may have had an impact on performance.

Second, a major issue is represented by the needs of to communicating information about the message passing parallelism (number of nodes, data distribution, etc) across the subroutine interfaces. On the one hand, this makes the code more complex; on the other hand, it hinders the portability of a hybrid application to a serial or SMP system. Interface issues were not tackled in this study. They will be the object of further research.

Performance

For the matrix multiply only modest size 2000 square matrices were tested. Here are several performance observations:

- Without communication hiding, communication time accounted for 15-20% of the elapsed time.
- With communication hiding, 75% of that time was recovered.
- On four 4-way Model F50 nodes, the best performance of 3500 megaflops was achieved using communication hiding and adaptive load balancing
- This should be compared to 1200 megaflops on a single F50 node using ESSLSMP DGEMM.

A speedup of almost 3 on 4 nodes was achieved. This is encouraging because little time was spent in optimizing these codes.

Table 5 shows the performance of QR-factorization with various cluster configurations, communication hiding or not, and dynamic versus adaptive strategies. No attempt was carried made to optimize the routines manually. For comparison in the *1 X 4* configuration (i.e. pure 4-way SMP), for $n = 2000$ [SaSm98] reported a performance of 1060 megaflops by the NAG library F68AEF; 743 megaflops was measured with the LAPACK routine DGEQRF.

Table 5. QR-Factorization performance for mixed MPI/OpenMP code on various cluster configurations. The performance data are shown in megaflops, where all execution times were measured using the system "wall clock" time. The "Hide" and "No Hide". Column headers refer to communication hiding and no communication hiding respectively. The configurations tested are N_n X N_{smp} where N_n is the number of nodes and N_{smp} is the number of processors per node.

Dynamic Versions (N_n X N_{smp})								
	1 X 1		**1 X 4**		**2 X 4**		**4 X 4**	
n	**Hide**	**No Hide**	**Hide**	**No Hide**	**Hide**	**No Hide**	**Hide**	**No Hide**
500	218	208	611	494	656	618	628	
1000	225	229	732	678	1128	912	1231	1131
2000			746	773	1310	1185	1963	1579
4000							2467	2124
Adaptive Versions (N_n X N_{smp})								
	1 X 1		**1 X 4**		**2 X 4**		**4 X 4**	

n	Hide	No Hide	Hide	No Hide	Hide	No Hide	Hide	No Hide
1000	229		700		1225		1796	
2000			713		1409		2507	
4000							2758	

SPECseis

Application Overview

SPECseis is a seismic processing benchmark used by the Standard Performance Evaluation Corporation (SPEC). [http://www.spec.org] It is representative of modern seismic processing programs used in the search for oil and gas [EiHa96]. It consists of 20,000 lines of Fortran. C is also used, mostly to interface with the operating system. The main algorithms in the 240 Fortran and 119 C subroutines are Fast Fourier Transforms and finite difference solvers. The benchmark includes five data sets. The smallest runs in 0.5 hours at 100Mflops and uses 110MB of temporary disk space. The largest data set runs in 240 hours at the same speed and uses 93 GB of disk space.

Why Message Passing And Directives?

Originally the benchmark was in message passing form. As high performance SMP and NUMA systems have become more widespread, the need for realistic high performance benchmarks to test SMP systems has increased. In addition, the authors wanted to test whether scalable parallelism is inherent in the programming model, message passing versus directives, or in the application itself.

Parallel Execution Scheme

The code is available in a serial and a parallel variant. The parallel variant comes in either a message-passing, PVM or MPI, and a directive, OpenMP, form. The two forms are very similar, in fact the OpenMP version was developed starting from the message passing variant. SPECseis uses an SPMD execution scheme with consecutive phases of computation and communication, separated by barrier synchronization.

The message passing variant starts directly in SPMD mode, that is, all processes start executing the exact same program. During the initialization phase, which must be 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, which encompasses the rest of the program.

In both versions, all data are kept local to the processes and are partitioned in the same way. In PVM or MPI programs all data are always local to the processes, whereas OpenMP programs give explicit locality attributes (default is *shared*). The only data elements that are declared shared in SPECseis are the regions for exchanging data between the processes. These regions are used in a mode similar to implementations of message passing libraries on SMP systems. The "sending" thread copies data to the shared buffer and the "receiving" thread copies it into its local data space. This scheme can be improved by allocating in shared memory all data that will need communication. However, we have not yet done this optimization in SPECseis.

Programming Issues

A number of issues arose when converting the message passing variant of SPECseis to OpenMP:

1. Data privatization: All data, except for communication buffers were declared *private* to all threads so that the memory required for a dataset became the same as for the message passing version. OpenMP's syntactic forms allowed us to privatize individual data elements, arrays, as well as entire common blocks in a straightforward manner. The default for all data can be switched to *private* to achieve the same effect.
2. Broadcasting common blocks: OpenMP allows broadcasting common blocks that have been initialized before a parallel region to all the private copies of the common block inside the parallel region. This was used extensively in SPECseis. One issue was that OpenMP requires this broadcast to be specified

statically and explicitly. In SPECseis, it is not always necessary to copy all common block data into the parallel regions, however our OpenMP variant does so, incurring slight additional execution overhead.

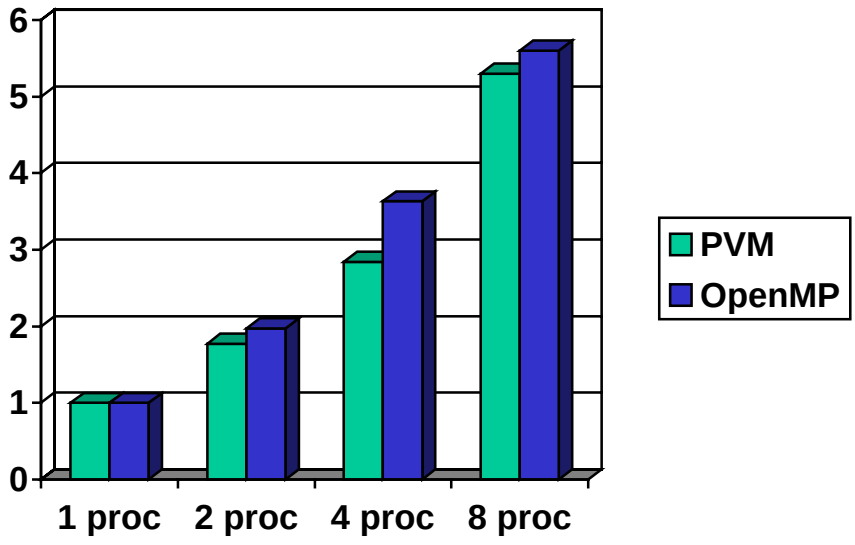
3. Mixing C and Fortran: Our implementation was done using a Fortran OpenMP compiler only. OpenMP for C was not yet available. However, since all calls to C routines in SPECseis are done within parallel regions, this did not result in degraded performance. The issues were twofold. First, C global data was difficult to privatize We did this by making an explicit copy for each thread. Instead, using OpenMP C features directly, we could simply have given a private attribute to these data. Second, although OpenMP requires operating system calls to be thread-safe, C programs that are compiled with a non OpenMP-aware compiler don't have this property. We had to use operating system-specific mechanisms to guarantee a thread-safe program, placing a critical section around memory allocation, for example.
4. Thread binding: A final issue in our OpenMP implementation was that OpenMP does not require processes to be bound to specific processors, nor does it provide mechanisms to specify this execution attribute explicitly. Again, we had to use operating system-specific mechanisms to achieve this, where available.

We have found that OpenMP provides all necessary constructs to do scalable SPMD style programming. Although this is only one case study, we feel that OpenMP is easy to learn and even obviated several problems that turned out to be time-consuming in our message passing experiments. For example, there was no need to setup configuration files and message-passing demons. Basically the program ran "out-of-the-box".

Performance

Many might perceive message passing to be more scalable than directive parallelism. However, our message passing and OpenMP variants use the same parallelization scheme with the same data partitioning and high-level parallelism and the same scalability is achieved. Figure 5 shows results obtained on an SGI Power Challenge. Although the OpenMP variant runs slightly faster that the PVM version, we have seen that this slight difference disappears if we increase the data set size. Hence, we attribute it to the higher message passing costs for exchanging small data sections.

Figure 5. SPECseis speedup comparison for PVM and OpenMP.



TLNS3D

Application Overview
TLNS3D, developed at NASA Langley, is a thin-layer Navier-Stokes solver used in computational fluid dynamics analyses. The program is capable of handling models composed of multiple blocks connected to each other via various boundary conditions. The code is written in Fortran 77 and is portable to major Unix platforms and Window NT.

Why Message Passing And Directives?
To simplify flow modeling of complex objects, typical input data sets contain multiple blocks, motivated by the geometry of the physical model. These blocks can be computed concurrently, and MPI is used to divide the blocks into groups and assign each group to a processor. The block assignment is static during the duration of the run because distinct data files must be created for each worker.

This approach is quite effective for models in which the number of blocks greatly exceeds the number of workers, since TLNS3D can generally group the blocks in such a way 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. In this case, no load balancing is possible. This limits the best-case parallel speedup to the number of blocks. A tool for splitting large blocks into multiple blocks before the run could be used to increase the number of blocks but modifying blocks if desired 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 all input/output and the workers do all the numerical computations. In addition to performing I/O, the boss acts like a worker. A run consists of multiple iterations, and at the end of each iteration, boundary data is exchanged among the MPI workers.

To address the limitations of the MPI level of parallelism, OpenMP directives were added to exploit parallelism within each block. Each block is represented as a three-dimensional grid, and most of the computations on that grid are in the form of loops that can be performed in parallel. Because there are many such parallel loops in TLNS3D, the OpenMP directives were carefully tuned, to maximize cache affinity between loops and to eliminate unnecessary global synchronization among threads.

In the case of single-block data sets, the OpenMP parallelism allows scalable parallel performance improvements on up to ten processors, where the MPI version achieved zero speedup. The key advantage of the mixed parallel version, however, is its 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, a group of threads, equal in number to the number of CPUs to be used, is partitioned 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 non-uniform allocation of threads achieves a second form of load balancing, which is effective for runs on a very large number of processors.

Programming Issues
Given a large number of CPUs, and two levels of available parallelism, the question arises of how many MPI processes to use. Generally, the goals are to minimize load imbalance while also minimizing communication and synchronization. Since the load imbalance arises when the number of MPI processes grows, and synchronization overhead grows with the number of OpenMP threads within each block, the correct balance can be difficult to find. An effective approach is to increase the number of MPI processes until a load imbalance begins to develop. At that point, use OpenMP threads to achieve additional speedup and to reduce the remaining load imbalance. This can be performed without modifying the code or the dataset.

The non-uniform mapping of threads to block groups in TLNS3D makes this application most effective on shared memory systems, such as the SGI Origin 2000. On other systems, such as an IBM SP, one's ability to use varying numbers of threads per MPI worker is severely limited. The mixed-parallel approach, however, can be used effectively on these machines if the number of MPI workers is limited to a number that permits good static load balance. In that case, a fixed number of threads can be assigned to each SMP node to achieve additional speedup.

Parallel Software Engineering
Parallel processing tools were found to be very effective in analyzing TLNS3D. On the MPI side, VAMPIR and VAMPIRTRACE programs from Pallas [http://www.pallas.com] analyzes the message passing performance to identify where delays occur. Block load imbalance can be identified in this way. On the OpenMP side, GuideView from KAI identifies OpenMP performance problems. Their Assure tool was also used to find shared variables that needed to be synchronized. This is an important issue when converting programs to directives because shared variables can be touched anywhere.

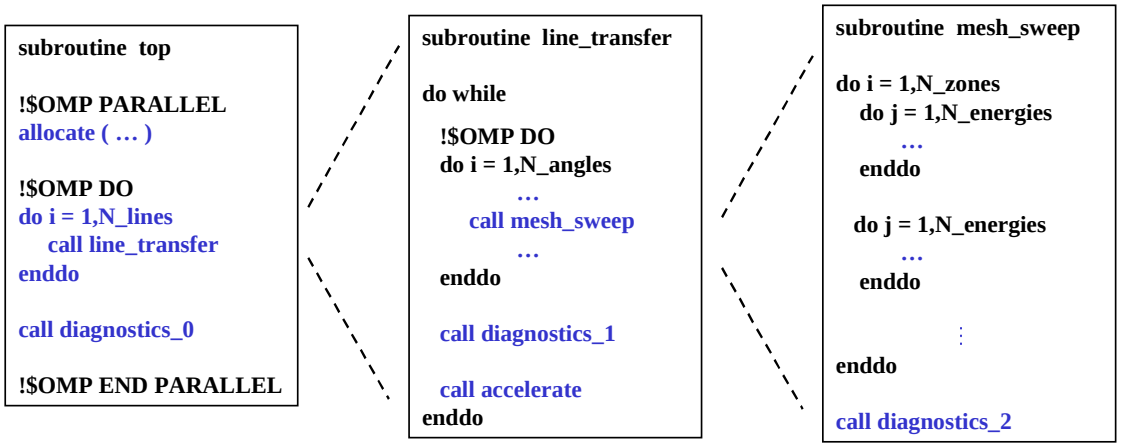
CRETIN

Application Overview
CRETIN is an non-LTE Physics application developed at Lawrence Livermore National Laboratory. It was developed over the last ten years by one developer. It is about 100K lines of Fortran. In terms of the many application packages developed at LLNL, it is moderate in complexity. There are smaller and younger codes which use more recent software technology and there are more complex codes which are considerably larger, older, and developed by a team of physicists. With the moderate and large codes, a major challenge for LLNL is to migrate these legacy applications from vector architectures to newer DMP and SMP parallelism available on ASCI systems.

Why Message Passing And Directives?
The Department of Energy ASCI project has installed parallel processing systems which push the state of the art. The ASCI blue systems are combined DMP / SMP systems which require mixed MPI OpenMP programming to achieve the goals of the ASCI project -- to simulate much larger physical models than previously feasible. By using portable MPI and OpenMP, CRETIN has been ported to two ASCI blue systems:

Blue Pacific : An IBM SP2 system with 1464 clusters of 4 processor PowerPC 604e nodes.
Blue Mountain : An SGI Origin2000 with 48 clusters of 128 processors.

Figure 6. Structure of radiation transport in Cretin. (The code is shown in a simplified version. E.g., the OpenMP IF clauses are omitted for clarity.)



Parallel Execution Scheme

CRETIN has several packages or modes of computation which have a high degree of parallelism. We will discuss two of them: Atomic Kinetics and Line Radiation Transport. Atomic Kinetics is a multiple zone computation with massive amounts of work in each zone. The loop over zones can be mapped 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. Several nested parallel regions were used with OpenMP IF clauses to select the best level of parallelism for the dimension of 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 6 shows part of the structure of Radiation Transport.

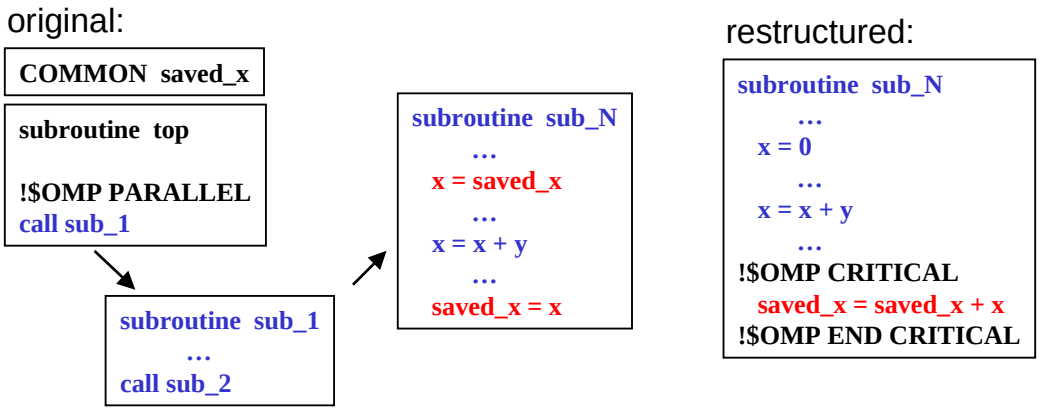
Programming Issues

In Atomic Kinetics, the amount of work in the zones varies over 5 orders of magnitude making load balancing of zones a critical issue. On each time step the computational time for each zone is used to sort and assign zones to processors for the next step. In the DMP version data was restructured to use temporary zones to avoid referencing the zone index. This took considerable work but it aided localization of data for the SMP version as well. Minor restructuring was performed to localize and minimize communication points for message passing. Here too, the SMP version benefited with fewer and larger critical sections. Hence the directive version was easy to develop. However, some additional work was required. Common blocks needed to be classified as shared or thread private and some private variables needed to be moved from the master initialization code to inside the parallel region before the worksharing parallel do.

In Line Radiation Transport, there are two options for DMP parallelization. Each processor can transport different lines over the entire mesh. In this case, data must be transposed from the distributed zones. This is code that is not needed in the serial or SMP version. The SMP version used 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. For larger problems, domain decomposition can be used. In this case, the data transpose is not needed but load balancing may suffer since the zone distribution across processors may not be efficient for the atomic kinetics. The transport algorithm becomes less efficient and requires more iterations for increasing numbers of domains. SMP parallelization helps significantly here by decreasing the number of domains for a given number of processors.

In both options, thread private workspaces were allocated for temporary space. Minor restructuring to minimize critical sections increased the efficiency of the lowest levels of parallelism. Multiple models for dynamic memory are available: Fortran77, Fortran90, and Cray pointers. This makes portable memory management tricky. Thread private allocations are performed using the pointers that are 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. We found this to be this is a subtle feature of OpenMP which all implementations must implement correctly. Using this is superior however to adding a thread specific index to all array references.

Figure 7. Cretin deep bug found with Assure.



Parallel Software Engineering

The majority of the parallelism work on the application was spent on the DMP level, 5 months. Only 1 month was spent on the SMP parallelization and much of that time was spent learning about how to use an SMP efficiently. However, it should be pointed out that the restructuring performed for 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. Most of the deep bugs were uncovered with the Assure tool for OpenMP. Figure 7 shows an example scenario for a deep bug. Several levels deep in a parallel region a local copy of a shared common variable had been made, updated, and stored back in the common. A critical section around the update had been forgotten. Also note that the local variable is initialized to zero rather than to the saved_x, such that the update of saved_x can be done in one atomic operation. This makes the critical section more efficient.

Finally, every effort was made to keep all of the parallelism consistent with a single source to simplify maintenance, debugging, and testing. This was achieved with the exception of data transpose, mentioned above, and is a great benefit to ongoing development.

Summary

All of the six applications described in the paper successfully developed high performance programs that use both message passing and directive parallel models. For the most part they used multiple levels of parallelism with the coarse grain utilizing DMP and the finer grain utilizing SMP. But, it is also shown that an application can have only one level of coarse grain, domain decomposition, parallelism mapped to both message passing and directive version in the same code. This can be achieved without loss of performance. Table below summarizes our experiences.

Table 6. Comparison of 6 applications using message passing and directives.

		CGWAVE	GAMESS	Linear Algebra Study
Why Use Directives And Message Passing		Add performance needed to attack another dimension in problem to solve	Flexible use of SMP clusters on problem with lots of parallelism	To get message passing scalability and good load balancing with directives
Paral lelis	**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 That Came Up		Calling message passing routines in parallel regions	Small granularity in MPI and Thread private efficiency of OpenMP	Couldn't use MPI within node and incomplete support for OpenMP
Parallel Software Engineering		Used Assure to explore OpenMP parallelism	OpenMP versions sometimes much simpler	Porting and maintaining two levels 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 computers at the leading edge.
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 That Came Up		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

Most platforms today support both parallel models. Very high levels of parallelism were achieved on all the systems tested: Compaq Alpha, IBM SP, and SGI. Intel also support both models. However the large variety of interconnection systems used with Intel platforms makes performance comparisons difficult. Although clustered SMPs are the current commodity architecture, using large-scale NUMA systems for variable size clusters has demonstrated its value in some applications.

In each case, the developers found and overcame problems in using leading-edge parallel processing technology. The main issue often was in understanding the semantics and implementation of the programming models. Their application to the many code situations was relatively straightforward and was successful in all cases. Although the situation is improving, sometimes it is a matter of availability of complete support for a parallel processing model. Running applications at the highest levels of performance is still a challenge. Running these parallel applications requires experimentation to operate the system correctly. We hope that our experiences will migrate to a daily production environment.

Figure 8. Parallel vector add.
(a) OpenMP version of vector add. (If directives are ignored, you have the serial version)

```
      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
```

(b) MPI / OpenMP version

```
      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
!$OMP PARALLEL DO
      DO i = 1,isize
       aloc(i) = bloc(i) + cloc(i)
      ENDDO
!$OMP END PARALLEL DO
!!                      Gather up sections of aloc into a on task 0
      CALL MPI_Gather(a,isize,MPI_REAL,aloc,isize,MPI_REAL,0,MPI_COMM_WORLD,ierr)
      CALL MPI_Finalize(ierr)
      RETURN
      END
```

Example

Figure 8 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.

References

[Grop94] William Gropp, Ewing Lusk, Anthony Skjellum, Using MPI, Portable Parallel Programming with the Message-Passing Interface, The MIT Press, 1994.

[Chan00] Rohit Chandra et al, Parallel programming in OpenMP, Morgan Kaufmann, London, 2000, ISBN: 1558606718.

[Dagu98] Leonardo Dagum and Ramesh Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming", Computational Science & Engineering, pp. 46-55, Jan/Mar 1998.

[FaDo96] Graham E. Fagg and Jack J. Dongarra, "PVMPI: An Integration of the PVM and MPI Systems". Technical Report UT-CS-96-328, University of Tennessee, Knoxville, TN, May 1996.

[XuPa96] By Bingyi Xu, Vijay Panchang, and Zeki Demirbilek, "Exterior Reflections in Elliptic Harbor Wave Models" Journal of Waterway, Port, Coastal, and Ocean Engineering, pp. 118-126, May/June 1996.

[BoBr99] Steve W. Bova, Clay P. Breshears, Christine Cuicchi, Zeki Demirbilek, Henry A. Gabb, "Dual-level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP" The International Journal of High Performance Computing Applications, Volume 14, Number 1, Spring, 2000, pp. 49-64.

[Schm92] M. W. Schmidt, et al, "General Atomic and Molecular Electronic Structure System", *J. Comp. Chem.* **14**, 1347-1363 (1993).

[HaSh94] R. J. Harrison and R. Shepard, "Ab Initio Molecular Electronic Structure on Parallel Computers", Ann. Rev. Phys. Chem. 45, 623-658 (1994).

[FeKe95] D. Feller, R. A. Kendall, and M. J. Brightman, "The EMSL Ab Initio Methods Benchmark Report", Pacific Northwest Laboratory, Report number PNL-10481, March 1995.

[KuSt97] Kuhn, Bob and Eric Stahlberg, "Porting Scientific Software to Intel SMPs Under Windows/NT", Scientific Computing & Automation, Vol 14, No 12, pp 31-38, November 1997.

[SaSm98] Stef Salvini, Brian T. Smith, and John Greenfield, "Towards Mixed Mode Parallelism on the New Model F50-Based IBM SP System", Albuquerque High Performance Computing Center, University of New Mexico, Technical Report AHPCC98-003, September 1998.

[EiHa96] Rudolf Eigenmann and Siamak Hassanzadeh, "Benchmarking with Real Industrial Applications: The SPEC High-Perfomance Group", *IEEE Computational Science and Engineering*, Spring 1996.