A HYBRID MPI-OPENMP IMPLEMENTATION OF AN IMPLICIT FINITE-ELEMENT CODE ON PARALLEL ARCHITECTURES

G. Mahinthakumar

NORTH CAROLINA STATE UNIVERSITY, USA

F. Saied

NATIONAL CENTER FOR SUPERCOMPUTING APPLICATIONS, USA

Summary

The hybrid MPI-OpenMP model is a natural parallel programming paradigm for emerging parallel architectures that are based on symmetric multiprocessor (SMP) clusters. This paper presents a hybrid implementation adapted for an implicit finite-element code developed for groundwater transport simulations. The original code was parallelized for distributed memory architectures using MPI (Message Passing Interface) using a domain decomposition strategy. OpenMP directives were then added to the code (a straightforward loop-level implementation) to use multiple threads within each MPI process. To improve the OpenMP performance, several loop modifications were adopted. The parallel performance results are compared for four modern parallel architectures. The results show that for most of the cases tested, the pure MPI approach outperforms the hybrid model. The exceptions to this observation were mainly due to a limitation in the MPI library implementation on one of the architectures. A general conclusion is that while the hybrid model is a promising approach for SMP cluster architectures, at the time of this writing, the payoff may not be justified for converting all existing MPI codes to hybrid codes. However, improvements in OpenMP compilers combined with potential MPI limitations in SMP nodes may make the hybrid approach more attractive for a broader set of applications in the future.

Address reprint requests to G. (Kumar) Mahinthakumar, Department of Civil Engineering, Campus Box 7908, North Carolina State University, Raleigh, NC 27695-7908, USA tel: 919-515-7696, fax: 919-515-7908, e-mail: gmkumar@eos.ncsu.edu

The International Journal of High Performance Computing Applications, Volume 16, No. 4, Winter 2002, pp. 371–393

1 Introduction

A majority of the emerging parallel supercomputers are either symmetric multiprocessor (SMP) clusters (e.g. IBM SP, Compaq/Alpha SC) or distributed shared memory (DSM, also known as non-uniform shared memory or NUMA) (e.g. SGI Origin 2000) machines. This is a marked change from the earlier generation of uniform shared memory parallel vector processors (e.g. Cray J90) and pure distributed memory (Intel Paragon, Cray T3E, etc.) architectures. Programming for optimal performance on SMP cluster and DSM architectures is more challenging than the older architectures due to their inherently more complex memory hierarchy. The memory hierarchy is even more complicated by multiple levels of cache, which are becoming standard on most newer architectures. These trends in architectures are now pushing parallel programmers to adopt more complex programming paradigms such as hybrid models, which use both message passing (e.g. MPI, PVM, MPL, NX, etc.) and threading (e.g. OpenMP, pthreads, Solaris threads, Cray directives, Terra directives, etc.) in the same program (e.g. Bova et al. 2000, Hanebutte 2000). In the hybrid model one implements several OpenMP threads under each MPI process.

In recent years, MPI (Message Passing Interface; see Gropp et al. 1999), a distributed memory programming paradigm, has become the *de facto* standard for communicating data between parallel processors on modern parallel architectures. MPI is widely supported on most parallel architectures in languages such as Fortran and C. More recently, MPI support has been extended to C++ and Java on some freeware MPI libraries. OpenMP was recently introduced as a shared memory programming standard on shared memory multiprocessor systems, intended to replace existing thread protocols such as `pthreads` as a more simplified interface. Currently OpenMP is only supported in Fortran and C. While this study specifically uses OpenMP as the thread protocol, most of the performance results should in principle apply to other thread protocols such as `pthreads` because OpenMP is built on top of these lower-level thread protocols.

A pure MPI approach is best suited for distributed memory architectures even though most vendors now support MPI on shared memory nodes. A pure OpenMP approach is best suited for uniform shared memory nodes even though many vendors are now starting to support OpenMP on non-uniform shared memory nodes. A hybrid MPI-OpenMP approach is thought of as the best approach for SMP cluster and DSM architectures. We believe that this type of programming approach may become increasingly popular with a growing trend in SMP cluster and DSM architectures. In a hybrid MPI-OpenMP approach, OpenMP directives are placed under each MPI process and consequently a certain number of OpenMP threads

```
!========================================================================
!   ORIGINAL PURE MPI CODE
!   pi.f - compute pi by integrating f(x) = 4/(1 + x**2)
!========================================================================
      program main
      include 'mpif.h'
      double precision  mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, ierr
!function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
      call MPI_BCAST(n,1,MPI_INTEGER,0, MPI_COMM_WORLD,ierr)
      h = 1.0d0/n
      sum  = 0.0d0
      do 20 i = myid+1, n, numprocs
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
20    enddo
      mypi = h * sum
! collect all the partial sums
      call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,
    &      MPI_SUM,0,MPI_COMM_WORLD,ierr)
      call MPI_FINALIZE(ierr)
      stop
      end
!========================================================================

!========================================================================
!   HYBRID MPI-OpenMP CODE
!   pi.f - compute pi by integrating f(x) = 4/(1 + x**2)
!========================================================================
      program main
      include 'mpif.h'
      double precision  mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, ierr
!function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr )
      call MPI_BCAST(n,1,MPI_INTEGER,0, MPI_COMM_WORLD,ierr)
      h = 1.0d0/n
      sum  = 0.0d0
!$OMP PARALLEL DO REDUCTION (+:sum) PRIVATE (x)
      do 20 i = myid+1, n, numprocs
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
20   enddo
!$OMP END PARALLEL DO
      mypi = h * sum
! collect all the partial sums
      call MPI_REDUCE(mypi,pi,1, MPI_DOUBLE_PRECISION,
    &      MPI_SUM,0,MPI_COMM_WORLD,ierr)
     call MPI_FINALIZE(ierr)
      stop
      end
!========================================================================
```

Fig.1 A simple Fortran example showing pure MPI and Hybrid Codes.

are invoked under each MPI process. The balance between the number of MPI processes and the number of OpenMP threads can be an important consideration for getting good performance. Adding loop level OpenMP directives to an existing MPI code is very straightforward. In Figure 1 we show a simple Fortran MPI code that calculates π by integrating the function $4/(1+x^2)$. On the upper part we show the original MPI version. The program is parallelized on `numprocs` MPI processes and the work is distributed across the processes based on the different rank of each process (`myid`). The value of `numprocs` is defined by the user while starting the MPI program. On the lower part we show the hybrid version that includes loop level OpenMP directives. In the parallel regions (enclosed by `$OMP PARALLEL` and `$OMP END PARALLEL` directives), each MPI process is split into separate threads, and the work of the loop is distributed among these threads. The `REDUCTION` clause specifies that a private copy of the variable `sum` is to be kept by each thread during the operation of the loop and a summation of the result across all threads be placed in the master thread at the completion of the loop. The `PRIVATE` clause specifies that the scalar variable `x` is private (by default all variables except loop indices are `SHARED`).

The objective of this work is to implement the hybrid model on a full application code and compare its performance on various SMP cluster and DSM architectures. For this purpose we have chosen an implicit finite element code used for simulating multicomponent groundwater transport. An overview of the code is provided in the next section. In general, an implicit finite-element code using sparse iterative solvers can suffer from performance problems on most modern architectures because such codes are generally communication, synchronization, and memory intensive. One of our goals is to evaluate the conditions under which a hybrid MPI-OpenMP approach can be useful in an implicit code. We will investigate the most efficient OpenMP efficient strategies for a hybrid implementation and the optimum balance between the number of MPI processes and the number of OpenMP threads on different architectures.

In recent years there have been several publications on the application of hybrid MPI-OpenMP model (e.g. Henty 2000, Smith and Kent 2000, Capello and Etiemble 2000, Bova et al. 2000, Sarma and Adeli 2001). However, a number of these investigations have focused on using MPI for the embarrassingly parallel portion (e.g. Monte-Carlo or Genetic Algorithm) and OpenMP for the difficult to parallelize portion (e.g. finite-element or finite-difference) of the application. In contrast, our focus is on using the hybrid model in all phases of an implicit finite-element application thereby enabling a direct comparison between the pure MPI, hybrid, and pure OpenMP modes. The unique aspects of this work are: (i) a systematic performance comparison of several popular parallel architectures, (ii) choice of an implicit finite element code that is representative of many production level codes, and (iii) special modifications for improving the performance of the hybrid model for sparse matrix-vector multiplication and matrix assembly operations.

2 Overview of Groundwater Transport Code

The multicomponent groundwater transport code that is used in this investigation has been used in a variety of problems including field applications (e.g. Mahinthakumar and West 1998), parameter estimation studies (e.g. Mahinthakumar et al. 1999), laboratory studies (Shin et al. 2002), and parallel performance studies (e.g. Mahinthakumar and Saied 1999). The general system of equations describing transport of *nc* dissolved components undergoing reactions in saturated porous media is defined by a nonlinear time-dependent coupled partial differential equation (p.d.e) system given by

$$\frac{\partial C_i}{\partial t} = \nabla \cdot (\mathbf{D} \cdot \nabla C_i) + \nabla \cdot (C_i \mathbf{v}) + \frac{q}{\theta}(C_i - C_{0i}) + R_i$$

$$i = 1, 2, 3, \ldots, nc \qquad (1)$$

where $\mathbf{v}$ is the 3×1 velocity field vector, $\mathbf{D}$ is the 3×3 dispersion tensor dependent on $\mathbf{v}$, and C_i is the dissolved concentration of component i. The term $q/(C_i - C_{0i})/\theta$ represents the source term with volumetric flux q, medium porosity θ, and injected concentration C_{0i} (e.g. from injection wells). R_i is the rate of mass loss of component i due to sorption and bioremediation reactions and is the main coupling term for the system of equations. The term R_i may contain many terms and can be nonlinear. For example, if only bioremediation reactions are present then R_i is given by

$$R_i = \mu_{max} F_i X \prod_{\substack{j=1 \\ f_{ji} \neq 0}}^{j=nc} f_{ji} \left(\frac{C_j}{K_j + C_j} \right) \quad i = 1, 2, 3, \ldots, nc \qquad (2)$$

where F_i is the stoichiometric ratio, X is the biomass concentration, μ_{max} is the maximum utilization rate, and f_{ji} is a factor controlling component j's contribution to component i's biodegradation process. If $f_{ji} = 0$ then component j does not participate in component i's biodegradation process.

The system of equations (1) is discretized using the Galerkin finite element method with 8-node linear hexahedral elements. A logically rectangular grid structure is assumed but irregular geometries are supported using

distorted elements. A Crank-Nicolson approximation (central finite-difference) is used for the time derivative terms. A lumped mass formulation (Huyakorn and Pinder 1983) is used for all time-derivative and non-derivative (zeroth spatial derivative) terms. The coupled non-linear system described by (1) is solved using a modified form of the Sequential Iterative Algorithm (SIA) with a semi-explicit method to decouple the system (Mahinthakumar et al. 1999). In each iteration, a full matrix solve of the decoupled system is performed for the linear system arising from equations (1). Iterations are performed until the fully coupled system is satisfied (typically 4-5 iterations per time step for a six component system). A diagonal storage format is used for the global matrices arising for each component of the decoupled system. We have implemented BiCGSTAB, GMRES(k), ORTHOMIN(k), and CGS iterative solvers for the matrix solution and the comparison of these solvers is discussed in Mahinthakumar et al. (1997). Our performance results in this paper are based on the BiCGSTAB solver which performs reasonably well for most problems.

Because we use a lumped mass formulation to describe the coupling terms (e.g. equation (2)), the coupling terms occur only on the main diagonals of the off diagonal blocks of the full matrix. We have achieved considerable memory and computational savings by not replicating the storage or computation of matrix entries that are common to all components and updating only those entries which change from time step to time step. Based on the user input data the code will determine which matrix entries are common to all components and which entries change for the subsequent time step. In most multi-component transport scenarios tremendous savings can be achieved by carefully tracking these needs. For example, for the six component transport system we used for our simulations we were able save a factor of 4 in matrix storage and a factor of 2 in matrix computations compared to the standard approach which caters for the worst case scenario. This implementation has enabled the rapid solution of a large number of components at a much higher resolution than previously possible. The combination of lumped mass formulations, decoupling of the full matrix, and other special treatments for memory and computational savings has one minor drawback: this restricts us from reordering the full sparse matrix to form dense sub blocks which may result in better floating point performance. Therefore we lose some of the advantages that are traditionally associated with a vector p.d.e (multicomponent transport) as opposed to a scalar p.d.e (e.g. single component transport). The codes are mainly written in Fortran (some F90 features and some C) using double-precision arithmetic.

3 Parallelization

For the hybrid model, parallelization is done at two levels: distributed memory MPI parallelism, and shared memory OpenMP parallelism. The original code was already parallelized using MPI, and OpenMP directives were added more recently to complete the hybrid model.

3.1 MPI

The MPI implementation uses a two-dimensional (2D) domain decomposition in the x and y directions (for more details see Saied and Mahinthakumar 1998). A 2D decomposition is generally adequate for groundwater problems because common groundwater aquifer geometries involve a vertical dimension that is much smaller than the other two dimensions. For the finite-element discretization such decomposition involves communication with at most 8 neighboring processors. We note here that a 3D decomposition in this case will generally require communication with up to 26 neighboring processors. We overlap one layer of processor boundary elements in our decomposition to avoid additional communication during the assembly stage at the expense of some duplication in element computations. There is no overlap in node points. In order to preserve the 27-diagonal band structure within each processor submatrix, we perform a local numbering of the nodes for each processor subdomain. This resulted in non-contiguous rows being allocated to each processor in the global sense. For local computations each processor is responsible only for its portion of the rows that are locally contiguous. However, such numbering gives rise to some difficulties during explicit communication and I/O stages. For example, in explicit message passing, non-contiguous array segments had to be gathered into temporary buffers prior to sending. These are then unpacked by the receiving processor. This buffering contributes somewhat to the communication overhead. When the solution output is written to a file we had to make sure that the proper order is preserved in the global sense. This required non-contiguous writes to a file resulting in some I/O performance degradation particularly when a large number of processors were involved. All explicit communications between neighboring processors are performed using asynchronous MPI calls (`MPI_Irecv` and `MPI_Send`). MPI collective communication calls (`MPI_Allreduce`) were used for global communication operations such as those used in dot products. Parallel I/O is performed using MPI-IO calls (ROMIO) or optionally have a single processor read/write data through message passing and a temporary buffer. The single processor I/O option was built for portability on systems where MPI-IO could not be used with native MPI. Parallel I/O using MPI-IO has

been tested on the Origin 2000 and Intel Paragon systems (Mackay et al. 1998).

3.2 OPENMP

Although our OpenMP implementation is based on straightforward loop level parallelism, several modifications were performed to the compute intensive loops in an attempt to improve the OpenMP performance. These modifications included explicit thread decomposition of loops, simple red-black type color codings to prevent thread conflicts during writes, and interchanging loop orderings (described below in the sub sections). One of the advantages of OpenMP is the ability to approach the parallelism in a step-by-step incremental fashion. For example, we can first try automatic compiler parallelization (step 1), then insert loop level directives without any code modification (step 2), then modify poorly performing loops to improve performance (step 3), and finally perform a full-fledged memory decomposition (step 4). If OpenMP scalability for a large number of processors is not an issue, then we can get satisfactory performance by stopping at step two or three. We note here that large-scale OpenMP scalability is not generally important for hybrid implementations targeted for SMP cluster architectures because the shared memory nodes on these architectures typically consist of a small number of processors (anywhere from 2 to 16). In this study our OpenMP parallelization is limited to the first three steps. The fourth and final step usually requires extensive code modification and is somewhat similar to decomposing the memory for MPI parallelism. In this mode all the major arrays will be local (or `PRIVATE` as opposed to the default `SHARED` mode) to each thread and data exchange between threads can be accomplished by copying the exchange data into shared buffer arrays.

Our OpenMP parallelism was targeted for all the major components of the code: matrix computation and assembly, matrix solution, and all other loops involving initialization and updates. Automatic compiler parallelization gave reasonable performance on most of the loops only on the Origin 2000. However, on the other architectures the auto parallelization performance was poor.

We approached the OpenMP parallelization in a systematic step-by-step process. The first attempt was a quick automatic parallelization by the compiler. Although auto parallelization worked to some extent on the Origin 2000, on the other architectures this approach was disastrous. Next we identified compute intensive loops in the program and inserted standard OpenMP directives. Then we performed simple speedup tests to identify types of loops where loop modifications may improve OpenMP performance. Based on our tests, loop modifications were targeted toward three main types of compute intensive loops in the program: (1) matrix assembly, (2) sparse matrix vector multiplication, and (3) vector operations such as saxpys, dot products, and vector updates. Most of the initialization loops fell into the latter category. In the subsections that follow we discuss OpenMP parallelization strategies for each major operation, and some architecture specific issues. It should be noted that in these discussions, the number of elements and the number of grid points (equal to the number of rows in the global matrix) refer to per MPI process values after MPI decomposition and not the total number of elements or grid points.

3.2.1 Matrix Assembly. The finite-element matrix assembly operation involves the following: computation of local element matrices and addition to the global matrix. In our implementation, the outermost loops loop over the number of elements in *x*, *y*, and *z* directions, and the inner loops loop over the number of nodes in each element. The element matrices are computed by a call to a subroutine inside the number of elements loop. Each entry in the element matrices is then added to the global matrix at appropriate locations by looping over the entries in each element matrix. Since the order of the element matrices is 8×8 for the linear hexahedral elements adopted here, the number of entries in each element matrix is 64. The most straightforward way of parallelizing this operation is to place a parallel directive outside the outermost loop. In this way, each thread can operate on a set of elements thus dividing the most intensive computations among the threads. However, there is a bottleneck associated with this strategy: thread conflicts during the update of the global matrix. This is because elements belonging to different threads can share a node in the finite-element mesh. Smarter OpenMP compilers automatically place a synchronization lock on the global matrix update operation where multiple threads can potentially write to the same location at the same time. To be on the safe side, programmers should designate this code segment as a critical section by enclosing this segment within `$OMP_CRITICAL` directives as shown in Algorithm 1a in Figure 2 (for the purpose of illustration, the loops shown in Algorithm 1 have been simplified from the actual code). Critical sections are serial segments in a parallel region that can only be executed by one thread at a time. As a rule of thumb, critical segments should be avoided within parallel loops as they can cause big performance problems. We can avoid thread conflicts during global matrix update by using a red-black type scheme where all the threads first operate on the red elements and then the black elements as shown in Figure 3. The key here is to make sure that any two red elements or any two black elements will not have a common node. Because we are dealing with a structured finite-element mesh we can

```
! ========================================================================
! Algorithm 1a: Matrix Computation and Assembly
! Original Loop Ordering
! ========================================================================
!$OMP PARALLEL DO
!loop over number of elements in x,y,z directions
       Do i=1,nelemx
         Do j=1,nelemy
            Do k=1,nelemz
             ...
             Compute 8x8 Local_Element_Matrix for element(i,j,k)
             ...
             Do ii = 1,8
                Do jj = 1,8
                ...
                Compute location (in,jd) in global matrix
                ...
!$OMP CRITICAL
! note that (in,jd) can be the same for different (i,j,k,ii,jj)
              Global_Matrix(in,jd) = Global_Matrix(in,jd) +
     &        Local_Element_Matrix(ii,jj)
!$OMP END CRITICAL
               ...
              Enddo
            Enddo
          Enddo
        Enddo
       Enddo
!$OMP END DO NOWAIT
! ========================================================================

! ========================================================================
! Algorithm 1b: Matrix Computation and Assembly
! Red-Black Loop Ordering
! ========================================================================
       Do m=1,2    !loop over colors 1=red, 2=black
!$OMP PARALLEL DO
          Do i=m,nelemx,2
            Do j=1,nelemy
              Do k=1,nelemz
                ...
                Compute 8x8 Local_Element_Matrix for element(i,j,k)
                ...
                Do ii = 1,8
                  Do jj = 1,8
                  ...
                  Compute location (in,jd) in global matrix
                   ...
                   Global_Matrix(in,jd) = Global_Matrix(in,jd) +
      &              Local_Element_Matrix(ii,jj)
                ...
               Enddo
              Enddo
            Enddo
          Enddo
        Enddo
!$OMP END DO NOWAIT
       Enddo
! ========================================================================
```

Fig. 2 OpenMP parallelization using normal (Algorithm 1a) and red-black (Algorithm 1b) algorithms for matrix assembly.

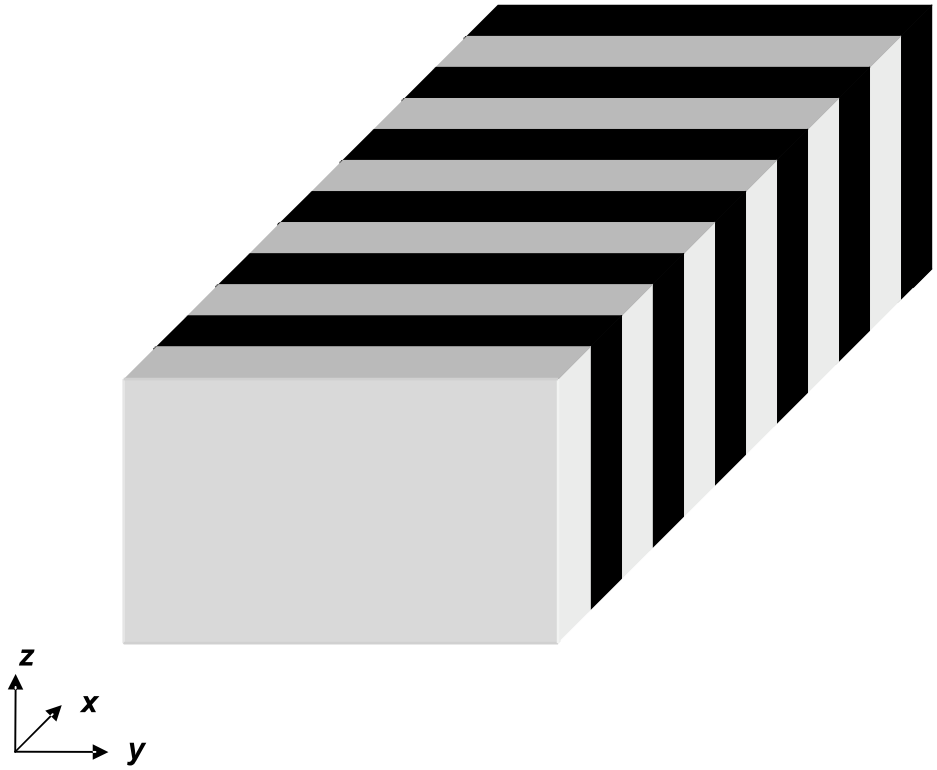


Fig. 3 Red-black ordering in the *y-z* plane. Data dependencies between adjacent planes are eliminated by computing the red planes first and the black planes second.

use a simple plane red-black decomposition where the red denotes the odd numbered x indices (y-z planes) and black denotes the even numbered x-indices. We note here that in the case of an unstructured mesh, a more complicated multi-color scheme can be used (Kumar et al. 1994). The modified loop structure corresponding to the red-black ordering is shown in Algorithm 1b. Note that the critical segment is not needed in the red-black version and should result in better performance.

3.2.2 Sparse Matrix-Vector Multiplication. The sparse matrix-vector multiplication (`spmatvec`) is one of the most compute intensive operations in the code (typically 50% of the total time). It is therefore important to parallelize this portion in an efficient manner. The `spmatvec` operation has two primary loops: a loop over the number of rows, and a loop over the number of entries per row (≤ 27). In the original code, the outer loop consisted of the number of entries per row and the inner loop consisted of the number of rows per processor (see Algorithm 2a in Figure 4). We note here that the number of rows (typically 1000 to 100000) is generally much larger than the number of entries per row (≤ 27 for 3D structured mesh). Although this loop ordering has good vectorization properties and is generally recommended for high memory bandwidth (e.g. Cray J90) or large cache (e.g. Origin 2000, newer IBM SPs) architectures, it can have performance problems on cache starved (e.g. older IBM SPs, Cray T3E) or low memory bandwidth (e.g. Pentium) architectures. For this reason, an interchanged loop ordering where the number of rows is the outer loop and the number of entries per row is the inner loop (i.e., shorter loop on the inside) is becoming increasingly popular in `spmatvec` operations (e.g. Anderson et al. 1999) as illustrated by Algorithm 2c in Figure 4. Note that this interchanged loop ordering also requires that the matrix storage scheme be reversed [e.g. in Fortran, Amatrix(no. of rows, 27)] to allow contiguous memory access to the matrix entries. Here we exploit

```
! ========================================================================
! Algorithm 2a: Sparse Matrix Vector Multiplication b=Ax
! Normal Loop Ordering
! ========================================================================
!loop over number of entries per row (ndiags= 27)
        Do j=1,ndiags
!loop over local number of rows per MPI process
!$OMP PARALLEL DO
           Do i=istrt,iend
               b(i)=b(i)+A(i,j)*x(i+nda(j))
           enddo
        enddo
!$OMP END DO NOWAIT
! ========================================================================


! ========================================================================
! Algorithm 2b: Sparse Matrix Vector Multiplication b=Ax
! Red-black Loop Ordering
! ========================================================================
        ncolors=2
! loop over number of colors (red=0, black=1)
        Do icolor=0,ncolors-1
!$OMP PARALLEL DO PRIVATE(istc,ienc)
! loop over alternate y-z planes along x-axis
          Do ix=1+icolor,nnpx,ncolors
            istc=(ix-1)*nspy+istrt
            ienc=istc+nspy-nv
! loop over number of entries per row (ndiags=27)
            Do j=1,ndiags
! loop over rows in each y-z plane
               Do i=istc,ienc
                    b(i)=b(i)+A(i,j)*x(i+nda(j))
               enddo
            enddo
          enddo
!$OMP END DO NOWAIT
        enddo
! ========================================================================


! ========================================================================
! Algorithm 2c: Sparse Matrix Vector Multiplication b=Ax
! Reversed Loop Ordering
! ========================================================================
!$OMP PARALLEL DO
!loop over local number of rows per MPI process
        Do i=istrt,iend
!loop over number of entries per row (ndiags= 27)
             Do j=1,ndiags
                b(i)=b(i)+A(j,i)*x(i+nda(j))
             enddo
        enddo
!$OMP END DO NOWAIT
```

Fig. 4 OpenMP parallelization using standard (Algorithm 2a), red-black (Algorithm 2b) and interchanged (Algorithm 2c) loop orderings for sparse matrix-vector multiplication.

CPP (C Preprocessor) macros to conveniently implement the alternative matrix storage scheme in the same code (see Section 3.2.5). For OpenMP parallelization, Algorithm 2a requires that the `PARALLEL DO` directive be placed at the inner loop in order to avoid thread conflicts during update of the result vector. While this is not a bad strategy, especially since the inner loop is much longer than the outer loop, a rule of thumb in OpenMP is to place the `PARALLEL DO` directive above the outermost loop if possible. We can achieve this by avoiding thread conflicts using a red-black type loop restructuring as shown in Algorithm 2b (Figure 4). The additional loops in Algorithm 2b will result in some overhead but this overhead should pay for itself for larger number of threads. Interchanging the loop ordering (smaller loop inside) as shown in Algorithm 2c lends itself better for OpenMP parallelization as there are no thread conflicts and the `PARALLEL DO` directive can be placed above the outermost loop. This algorithm also should give better MPI performance on cache-starved architectures such as the IBM SP. In our implementation, using Algorithm 2c caused some overhead on the MPI side because the matrix storage scheme is reversed and hence our standard MPI update boundary routines could not be used for updating the matrix entries without some extra memory copy operations. This overhead is generally small since in most practical problems the processor boundary entries of the global matrix had to be updated only in the first time step (in most cases only the diagonal entries of the global matrix that are local to each processor change during time stepping).

3.2.3 Vector-Vector Operations. A large number of loops in our code fall into this category. These include most of the initialization loops, update of solution vectors, saxpy's, dot products and so on. Most of these are simple single loops involving vectors of fixed size (in most cases, the size of the vector = the number of grid nodes per processor). For example, `x(1:n)=x(1:n)+z(1:n)*y(1:n)`. Since the vectors are generally long, and the operations are relatively simple, the performance is highly limited by cache and memory bandwidth of a single processor. When operations such as these are threaded using OpenMP, each thread will be competing for the bandwidth and the cache. On some architectures, this can have serious performance implications. It has been suggested that explicitly decomposing these loops for each thread may reduce some of the burden (Hanebutte 2000). In Algorithm 3a (Figure 5) we show the original loop for a `daxpy` operation and in Algorithm 3b we show the decomposed loop. While most modern OpenMP compilers will perform this task automatically (under a `STATIC` scheduling of loops), Hanebutte observed that OpenMP performance of vector loops can be improved for each thread by explicitly decomposing each loop beginning with all the initialization loops. We note here that Hanebutte's observations were on an older version of IBM SP (Nighthawk I – 220 MHz) using the C version of the OpenMP compiler. We also note here that in the early days of thread programming, many programmers had to attempt these kinds of explicit decompositions because the compilers were not mature enough or were less consistent across architectures.

3.2.4 Some architecture specific OpenMP issues. Because our first attempt on OpenMP parallelization was on the Origin 2000, we had to make sure we parallelized all the initialization loops since the threads are assigned according to the "first-touch" rule (Bova et al. 2000). According to the first-touch rule, the threads that first touch a piece of data (typically, a segment of an array) retain an affinity (data access is faster by that thread) to this data throughout the program execution. If the initialization loops are not parallelized then only the master thread will retain affinity toward the data. The first-touch feature is only enabled if the `OMP_DYNAMIC` environment variable is set to `FALSE` (usually the default). If the `OMP_DYNAMIC` environment variable is set to `TRUE`, then this feature is not invoked. On the Origin2000, initializing the arrays in parallel using the same loop structure as the compute intensive loops improved the performance slightly (though not to the extent reported by Bova et al. 2000). The same directives were kept for other architectures and no further testing was performed in this regard. The system developers for IBM SP provided a routine called `thread_bind()`, that was recommended to be compiled and linked in with the application code for programs run in a hybrid mode. Using the `thread_bind()` routine ensures that each thread is bound to a fixed processor throughout the program execution and that the threads do not migrate from one processor to the other within a node. Using `thread_bind()` resulted in some overhead for single MPI process (multiple thread) runs but resulted in improved performance for multiple MPI process runs.

3.2.5 Some common OpenMP optimizations. The OpenMP loop optimizations adopted previously only entailed those that are somewhat specific to the code that we analyzed. These included loop interchange and fusion (Algorithm 2c), static scheduling of loops (Algorithm 3b), and loop restructuring (Algorithm 1b). Although not explicitly mentioned earlier, some of the well known OpenMP optimizations such as barrier removal, loop consistency, and ordered serial loops were already incorporated in our implementation. A discussion of these can be found in Hoeflinger et al. (2001). Barrier removal is enforced in OpenMP using the `$OMP END NOWAIT` clause at the end of a parallel loop. Barrier removal reduces synchronization overheads. Loop consistency can

```
!============================================================
! Algorithm 3a: original loop
!============================================================
!$OMP PARALLEL DO
        Do i=1,n
           z(i)=sa*x(i)+y(i)
        Enddo
!$OMP END DO NOWAIT
!============================================================


!============================================================
! Algorithm 3b: thread decomposed loop
!============================================================
!$OMP PARALLEL DO PRIVATE(is,ie)
        Do j=1,nthreads
        is=istb(j)
        ie=ietb(j)
        Do i=is,ie
           z(i)=sa*x(i)+y(i)
        Enddo
        Enddo
!$OMP END DO NOWAIT
```

Fig. 5 OpenMP parallelization using standard (Algorithm 3a) and explicit thread decomposition (Algorithm 3b) for vector operations. The `daxpy` operation is shown as an example.

be maintained by using the same loop bounds for successive do loops if possible. Loop consistency promotes better cache usage. For improved cache performance, serial loops can be implemented using an ordered section (using the `ORDERED` directive) instead using the `MASTER` or `SINGLE` directive. We limit our discussion of these optimizations because these optimizations make a difference only when larger scale OpenMP scalability is desired (Luecke and Lin 2001, Wallcraft 2000, Hoeflinger et al. 2001); they make very little impact on OpenMP implementations that are targeted for a moderate number of threads (< 8) as in the case of SMP cluster architectures.

3.2.6 Implementation. In our implementation, we have used C preprocessor (CPP) macros to compile the code with any combination of the algorithms (1a, 1b, 2a, 2b, 2c, 3a, 3b) and with pure OpenMP, pure MPI, or hybrid modes. This facilitates easy testing of code for any algorithm combination.

4 Performance Results

In this section we report OpenMP, MPI, and hybrid performance for the different architectures. Our performance results are based on a test problem with model parameters taken from a field scale bioremediation problem reported by Semprini and McCarty (1991) involving the transport and aerobic biodegradation of six organic components (TCE, VC, t-DCE, c-DCE, DO, and CH_4). The base problem size is 41×41×11 (16000 elements, 18491 nodes, 6 components) involving about 110,946 unknowns. For the scalability analyses we fixed this as the problem size per processor. The problem sizes chosen here are purely for performance analysis purposes and are not representative of the field problem. For timing purposes, all simulations are performed for 100 time steps. For tests performed here we have used options so that very minimal I/O is performed (no check pointing and concentration field output only at the final time step).

Four parallel architectures were used in this study: (1) IBM SP with 184 4-way SMP nodes (Winterhawk I or WH I) each with four 375 MHz Power 3 processors located at Oak Ridge National Laboratory (ORNL), (2) IBM SP with 144 8-way SMP nodes (Nighthawk II or NH II) each with eight 375 MHz Power 3 processors located at San Diego Supercomputing Center (SDSC), (3) Compaq/Alpha SC server with 64 4-way SMP nodes each with four 667 MHz CPU's located at ORNL, (4) SGI Origin 2000 with 256 250 MHz processors located

at the National Center for Supercomputing Applications (NCSA). The IBM SP at SDSC was upgraded from the 222 MHz 8-way Nighthawk I nodes to the 375 MHz 8-way Nighthawk II nodes.

In order to limit the focus of this paper to the hybrid MPI-OpenMP performance we have not explicitly shown a comparison of pure OpenMP performance to pure MPI performance within a single SMP node. However, the performance analyses that were performed in this context showed that the pure MPI performance was almost always better than the pure OpenMP performance for all architectures except for the case of two processors in which case the performances were close. This limitation in the pure OpenMP model also extends to the hybrid model which performs best only when two OpenMP threads are used (more on this in Section 4.2.1).

4.1 PURE OPENMP WITHIN A SINGLE NODE

The pure OpenMP performance can only be measured within a single SMP node since most of our architectures (except Origin 2000) do not support OpenMP outside a node. In this section we compare our different OpenMP implementations for the matrix assembly and solver for all architectures. We used a fixed problem size (41×41×11) for these tests (6 components, 18941 nodes).

4.1.1 Matrix Assembly. For the matrix assembly, 2 algorithms were tested (see Section 3.2.1). Algorithm 1a is the original loop structure and Algorithm 1b is the modified red-black implementation. The results are shown in Figure 6. It is clear that while Algorithm 1b incurs some overhead for the single CPU case, it gives better performance for increasing number of CPU's. This is mainly because in Algorithm 1b we do not have the expensive `OMP_CRITICAL` section that was required in Algorithm 1a.

4.1.2 Matrix Solution. The matrix solver (BiCGSTAB) consists of the `spmatvec` operation and some vector operations (dot products, saxpy's, vector copies, scaling, etc.). For the `spmatvec` operation we tried three algorithms (see Section 3.2.2). Algorithm 2a is the original algorithm, 2b is the red-black implementation, and 2c is the reordered loop implementation. For the vector operations we tried two algorithms (see Section 3.2.3). Algorithm 3a is the original algorithm and 3b is the explicit loop decomposition. All our tests indicated that Algorithm 3b did not provide any improvement over 3a. This indicated that the compiler already performs this decomposition by default (this is the default STATIC schedule). Therefore, we focus our solver performance results on the different `spmatvec` algorithms (2a, 2b and 2c). The results are shown in Figure 7. In all cases we see that 2b and 2c outperform 2a. This is mainly because the non-blocking nature of 2b and 2c allows us to place the OpenMP directives outside the outer loop. However, 2b performs better than 2c for the IBM SP's, while 2c performs better for the Origin 2000 and the Compaq. This is primarily due to the cache sizes available on these machines. Algorithm 2c has better cache properties while 2b has better vectorization properties. On the IBM SP's L2 cache is 8 Mb while it is 2Mb on the Compaq and 4 Mb on the Origin 2000. Therefore 2c performs better than 2c on machines which are somewhat limited by the cache size.

4.2 SCALABILITY OF HYBRID VS PURE MPI

In this section we compare performance of Hybrid and Pure MPI implementations for increasing number of processors (from 1 to 128). The problem size per processor was approximately fixed at 41×41×11. That is, the problem size is 41×41×11 for 1 processor, 81×41×11 for 2 processors, 81×81×11 for 4 processors, and so on up to 641×321×11 for 128 processors.

4.2.1 Solver Performance. For each architecture the best hybrid and best MPI performance is compared for the BiCGSTAB solver. As indicated earlier, for all architectures the best hybrid performance was obtained when the number of OpenMP threads was limited to just two. When more than two OpenMP threads are used the performance of the hybrid model drops off. This behavior is illustrated in Table 1 for solver performance using Algorithm 2b for the IBM SP at ORNL (4-Way Winterhawk I nodes). For example, for 64 processors, the best hybrid performance corresponds to a hybrid model using 2 OpenMP threads and 32 MPI processes. For both hybrid and MPI timings, the best performing `spmatvec` algorithm was used for a given architecture. For the hybrid model, this corresponded to 2b for the two IBM SP's and 2c for the Compaq and the Origin2000. The scalability results are shown in Figure 8. It is evident that the pure MPI approach outperforms the hybrid model for all architectures except the 8-Way IBM SP. The reason is that on the 8-way Nighthawk II nodes, only up to 4 MPI processes are supported through the faster US (user space) interface. To run 8 MPI processes, the slower IP mode had to be used. While the use of IP mode is acceptable for small node configurations, the overhead for larger node configurations make a pure MPI mode less efficient. To facilitate a cross-architecture comparison of the solver performance, we show performance relative to a single processor of the Origin 2000 in Table 2. Single CPU Origin 2000 performance corresponds to about 94 Mflops (= 1 unit). From Table 2, we can see that the

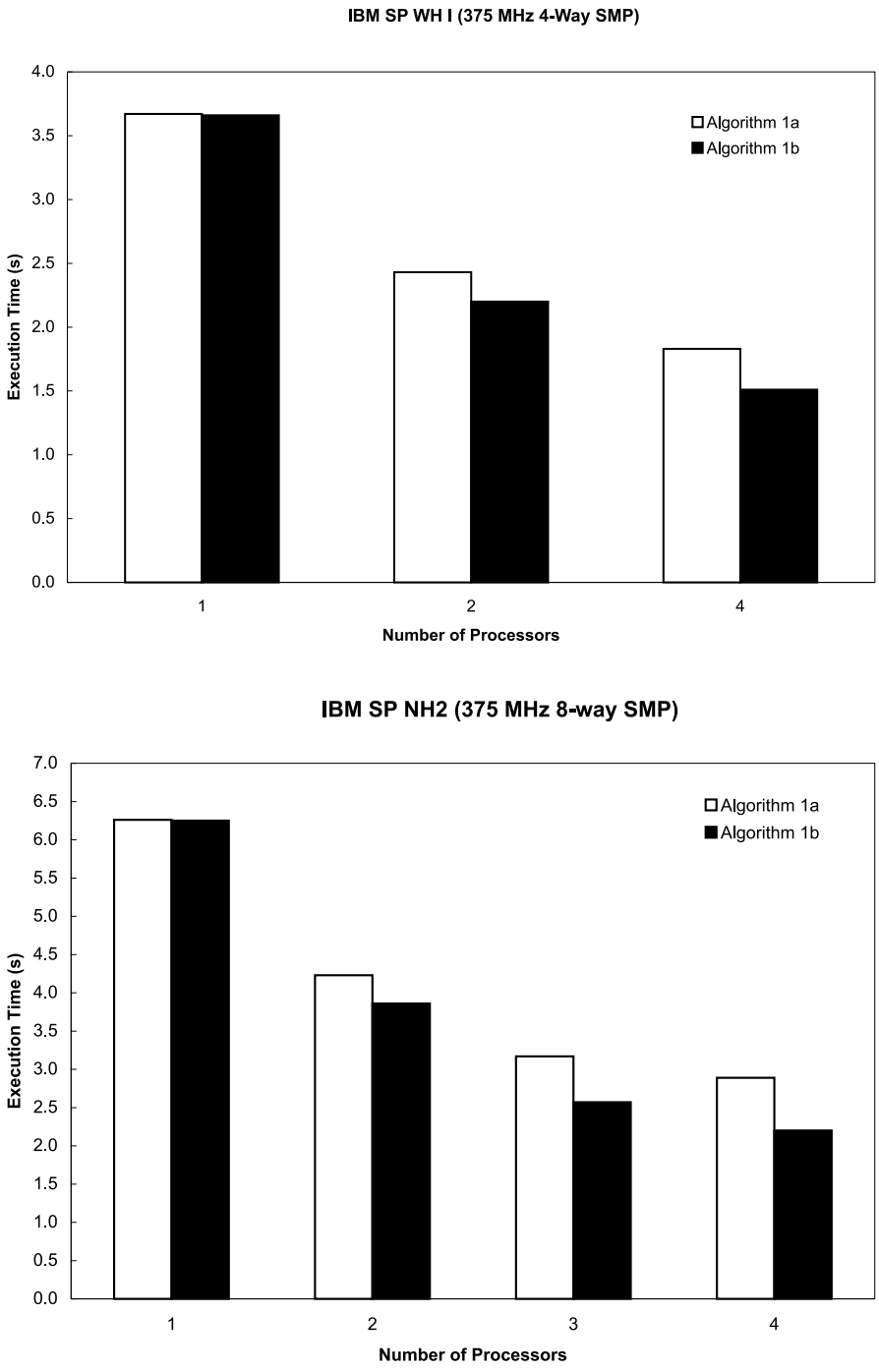


Fig. 6(a–b) Pure OpenMP timing comparison of standard (Algorithm 1a) and red-black ordering (Algorithm 1b) for matrix assembly. (a) IBM SP WH I, (b) IBM SP NH II.

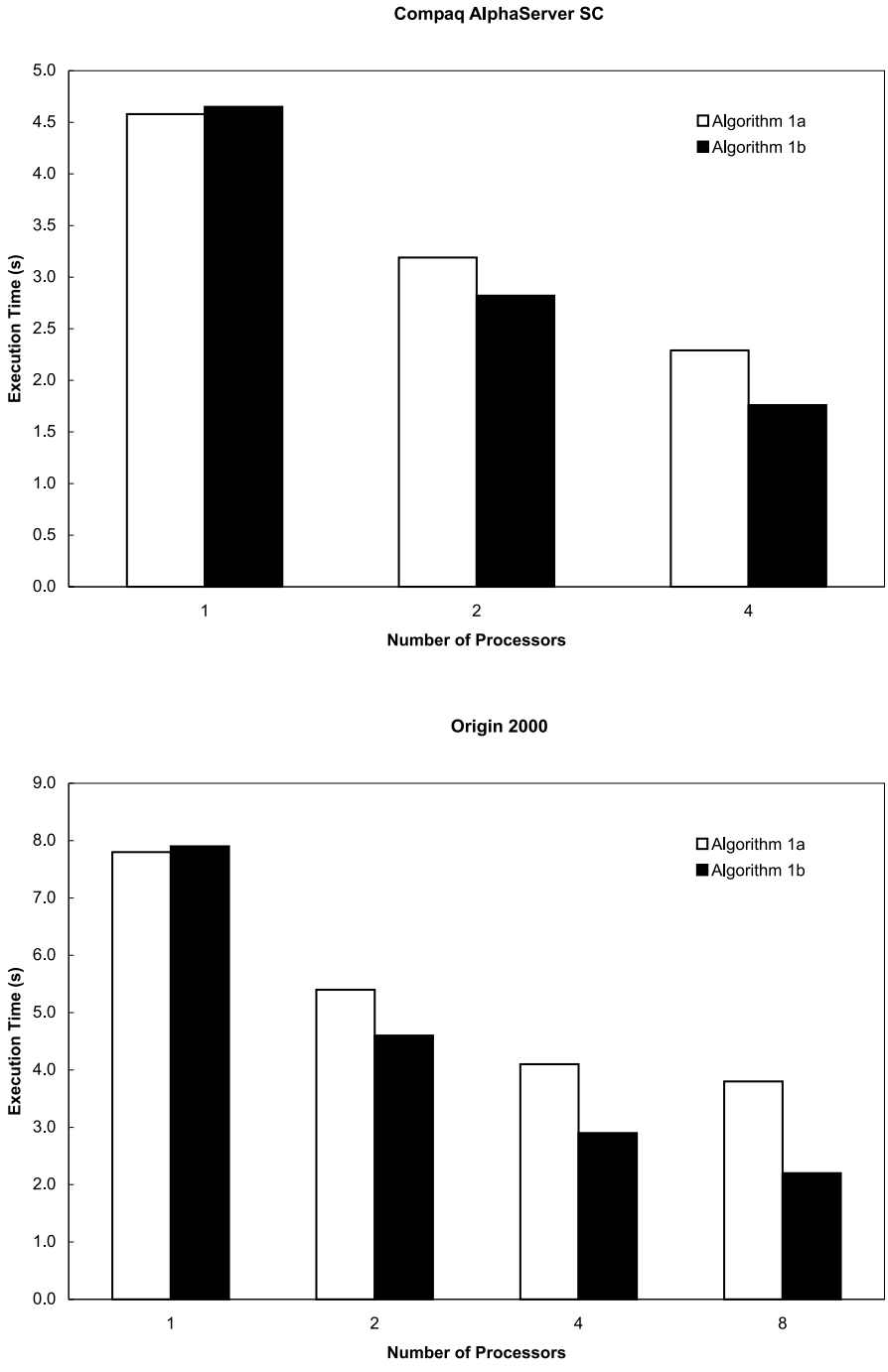


Fig. 6(c–d) Pure OpenMP timing comparison of standard (Algorithm 1a) and red-black ordering (Algorithm 1b) for matrix assembly. (c) Compaq/Alpha SC, (d) Origin 2000.

Table 1
Solver performance comparison for the hybrid and pure MPI modes on the IBM SP (4-Way Winterhawk I nodes) at ORNL. Algorithm 2b is used for the sparse matrix vector multiplication operation

Processors		HYBRID			Pure MPI	
NSMP	NCPU	OMP	MPI	Mflops	MPI	Mflops
1	1	1	1	229	1	233
	2	2	1	392	2	450
	4	4	1	550	4	914
		2	2	688		
2	8	4	2	945	8	1545
		2	4	1193		
4	16	4	4	1787	16	2312
		2	8	2065		
8	32	4	8	3312	32	4683
		2	16	3845		
16	64	4	16	5915	64	8341
		2	32	6901		
32	128	4	32	7402	128	13626
		2	64	9269		

NSMP = Number of SMP nodes, NCPU = Number of processors

peak performance of about 255 units (= 24 Gflops) is delivered by Compaq/Alpha using 128 processors for the pure MPI version.

4.2.2 Overall Scalability. The overall scalability of the code is examined using the total execution time. The total time is broken into 3 components: assembly, solver, and communication. The communication time includes total time spent on communication including the communication time spent in the assembly and solver components. The communication time also includes the gather and scatter operations that are required in point-to-point communication operations (see Section 3.1). We limit our comparison to two architectures: IBM SP (8-way NH II) at SDSC and the Compaq at ORNL. The comparison is shown in Figure 9. Since we are keeping the problem size per processor fixed, for perfect scalability, the total execution time should remain constant for a fixed number of time steps. Here we assume that the total number of solver iterations remain approximately constant with increasing problem size. In the actual simulations we observed a slight decrease in the number of iterations with increasing problem size. For example, for the smallest problem size of 41×41×11 (1 processor) the total number of BiCGSTAB iterations over 100 time steps was 863 and for the largest problem size of 641×321×11 (128 processors) it was 791.

In general, the Compaq scales better than the IBM SP. If we compare the hybrid version and the pure MPI version within each architecture, then on the IBM SP, the hybrid version is more scalable than the pure MPI version, and on the Compaq, the opposite is true. We can attribute this to the slower IP communication mode used on the IBM SP (NH II) for the pure MPI mode. On both platforms, the increase in communication time contributes to the scalability loss going from 1 to 128 processors. Further analysis of the communication time indicated that a significant portion (about 50%) of the communication time comes from the `MPI_Allreduce` operations performed in the dot product and norm computation steps in the BiCGSTAB solver. This is especially true for the larger number of processors on the IBM SP (NH II). For example, on the IBM SP (NH II) for 128 processors using pure MPI, the communication time is 46.5 seconds of which 27.3 seconds come from the

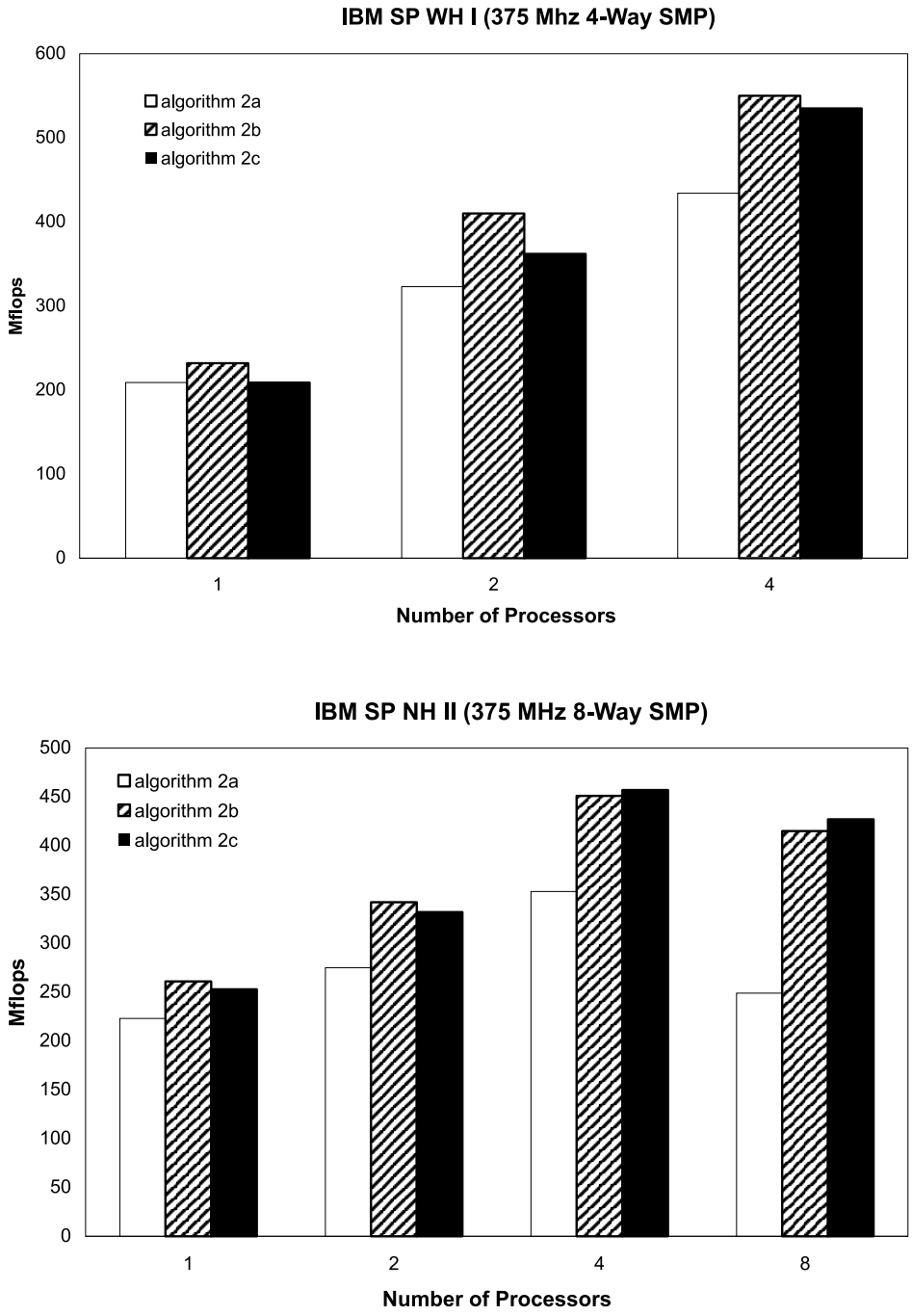


Fig. 7(a–b) Comparison of pure OpenMP solver performance based on the standard (Algorithm 2a), red-black (Algorithm 2b), and interchanged (Algorithm 2c) loop orderings for the sparse matrix vector multiplication. (a) IBM SP WH I, (b) IBM SP NH II.

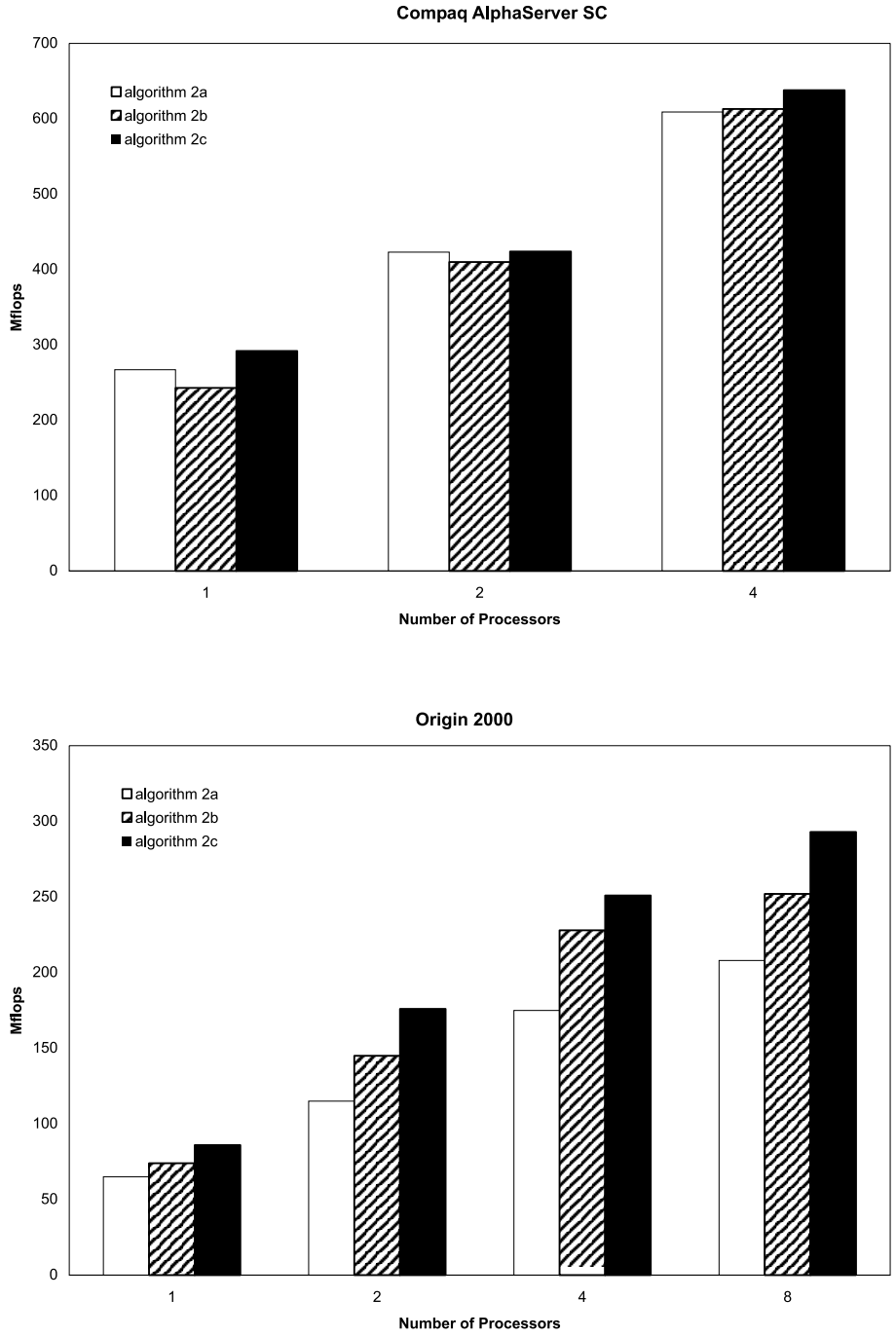


Fig. 7(c–d) Comparison of pure OpenMP solver performance based on the standard (Algorithm 2a), red-black (Algorithm 2b), and interchanged (Algorithm 2c) loop orderings for the sparse matrix vector multiplication. (c) Compaq/Alpha SC, (d) Origin 2000.

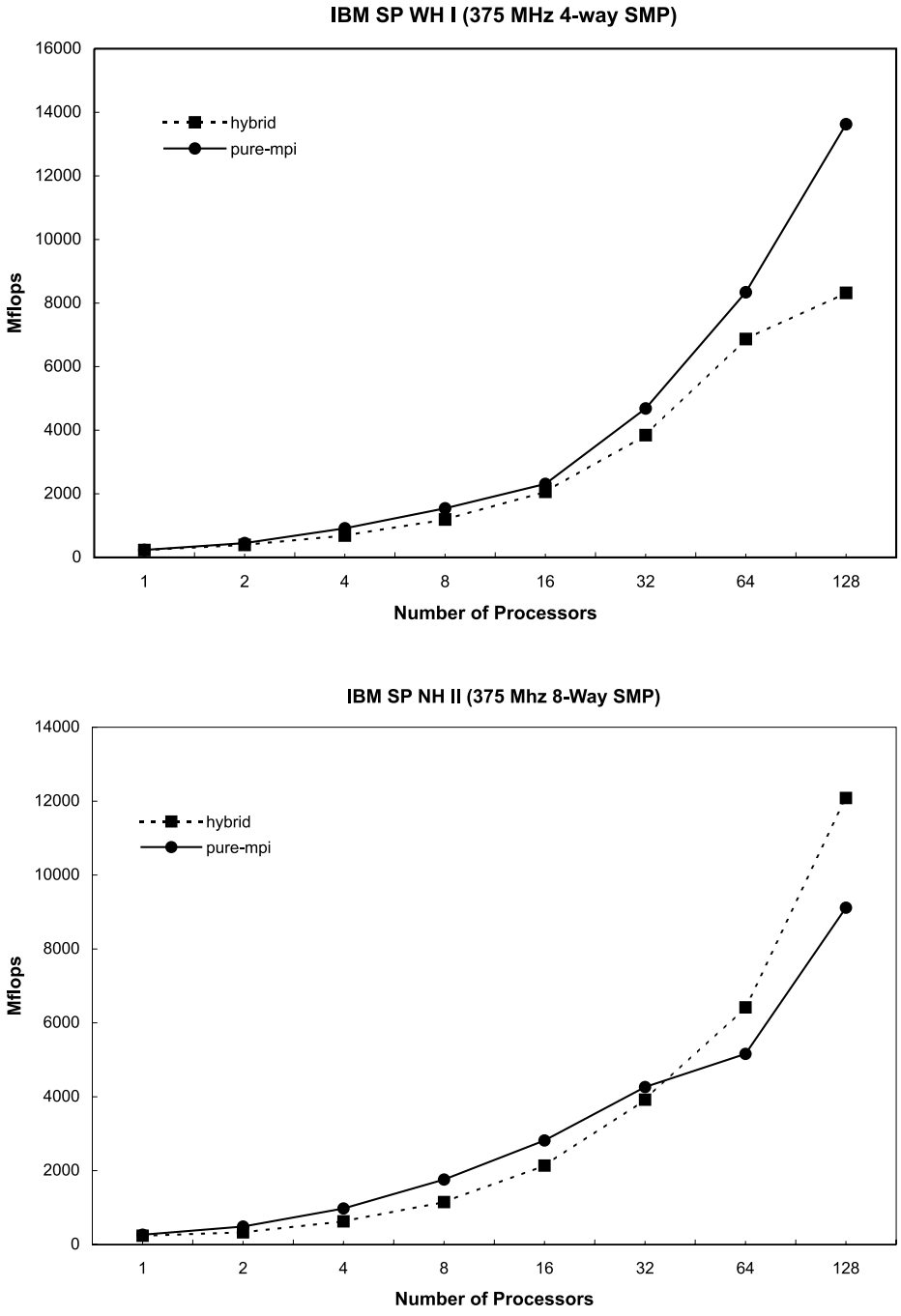


Fig. 8(a–b) Scalability of solver performance for pure MPI and the hybrid modes. The hybrid model uses two OpenMP threads in all cases. (a) IBM SP WH I, (b) IBM SP NH II.

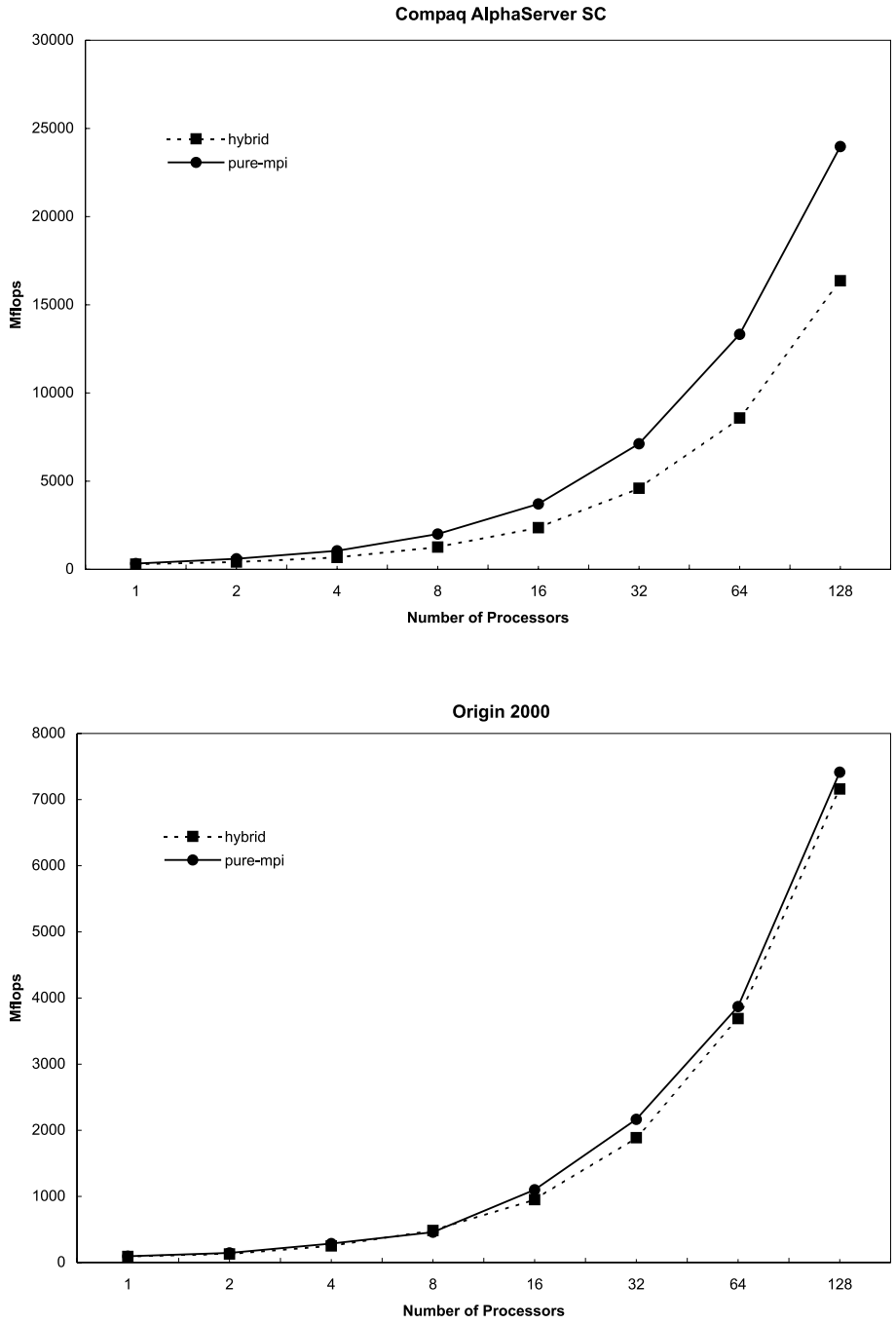


Fig. 8(c–d) Scalability of solver performance for pure MPI and the hybrid modes. The hybrid model uses two OpenMP threads in all cases. (c) Compaq/Alpha SC, (d) Origin 2000.

Table 2
Performance of solver on all platforms relative to single CPU pure MPI performance on Origin 2000

NP	IBM SP NH II		IBM SP WH I		Compaq/Alpha		Origin 2000	
	hybrid	pure-mpi	hybrid	pure-mpi	hybrid	pure-mpi	hybrid	pure-mpi
1	2.6	2.9	2.4	2.5	3.1	3.4	1.0	1.0
2	3.5	5.2	4.2	4.8	4.4	6.3	1.4	1.5
4	6.7	10.4	7.3	9.7	7.2	11.1	2.7	3.0
8	12.3	18.7	12.7	16.4	13.4	21.2	5.2	4.9
16	22.7	30.0	22.0	24.6	25.1	39.4	10.1	11.7
32	41.7	45.4	40.9	49.8	48.9	75.7	20.1	23.0
64	68.3	54.9	73.1	88.7	91.2	141.7	39.2	41.2
128	128.6	97.0	88.5	145.0	174.0	255.0	76.2	78.9

1 unit = 94 Mflops = Performance of 1 Origin 2000 CPU

`MPI_Allreduce` operations in the solver. The dot product communication time will also include synchronization overheads that are likely to be significant for a large number of processors. While these numbers are not unusual for the slower IP communication mode used on the IBM SP they are not acceptable. Even with the faster US communication mode used in the hybrid case, the dot product communication time (9.4 s) is approximately 50% of the total communication time (20.8 s) for 128 processors. On the Compaq/Alpha SC server also the dot product communication time (4 s) is about 50% of the total communication time (8.1 s) for 128 processors. These numbers underscore the need for better communication software and hardware that can handle global communication operations more efficiently. The authors want to note here that despite the presence of global communication operations, the same code (pure MPI version) exhibited very good scalability up to a large number of processors (greater than 256 processors) on older architectures such as the Intel Paragon and Cray T3E (Mahinthakumar and Saied 1999). Recent efforts (Vadhiyar et al. 2000) on automatically tuning the MPI collective communication operations can greatly benefit implicit applications such as the one studied here.

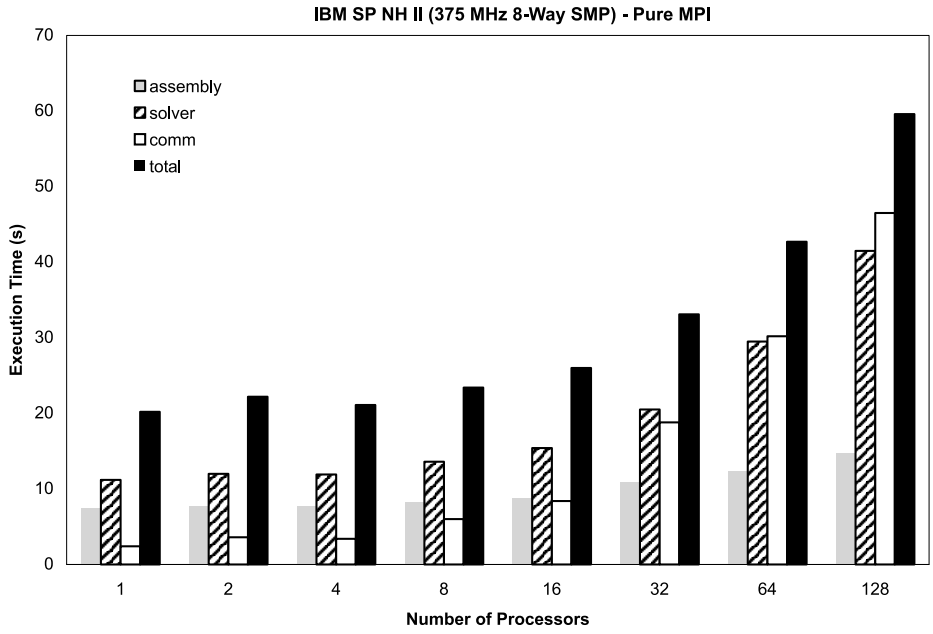
If we look at the timing breakdown, on both platforms, particularly for larger processor configurations, matrix assembly takes about 30% of the total time, and the matrix solver 60%. All the other operations such as initializations, I/O, and updates take around 10%. For most problems, the percentage of time spent in the solver should increase with increasing number of time steps. This is because, for most problems, the assembly operation after the first time step is restricted to assembling the main diagonal and the right hand side.

5 Concluding Remarks

In this work we have implemented a hybrid MPI-OpenMP model of an implicit finite element application and compared its performance against a pure MPI model on various parallel architectures. The MPI parallelization is based on domain decomposition and the OpenMP parallelization is based on loop level directives. In order to improve the OpenMP performance, several loop modifications were attempted in the compute intensive loops. Even with these modifications, loop level OpenMP parallelization does not perform as well as MPI beyond a few processors. However, the additional work involved in going beyond loop level parallelism may not be justified if OpenMP parallelism is targeted for only a few processors as in the case of an SMP node.

Although the finite element application considered here uses a structured grid, many of the strategies adopted here can be extended to unstructured finite element applications. For example, the red-black scheme can be replaced by a multi-colored scheme, and the sparse matrix-vector multiplication can be performed with the “larger” number of rows on the outer loop and the “smaller” number of non-zeros per row in the inner loop.

On most of the architectures tested, the pure MPI version outperformed the hybrid version. The exception was



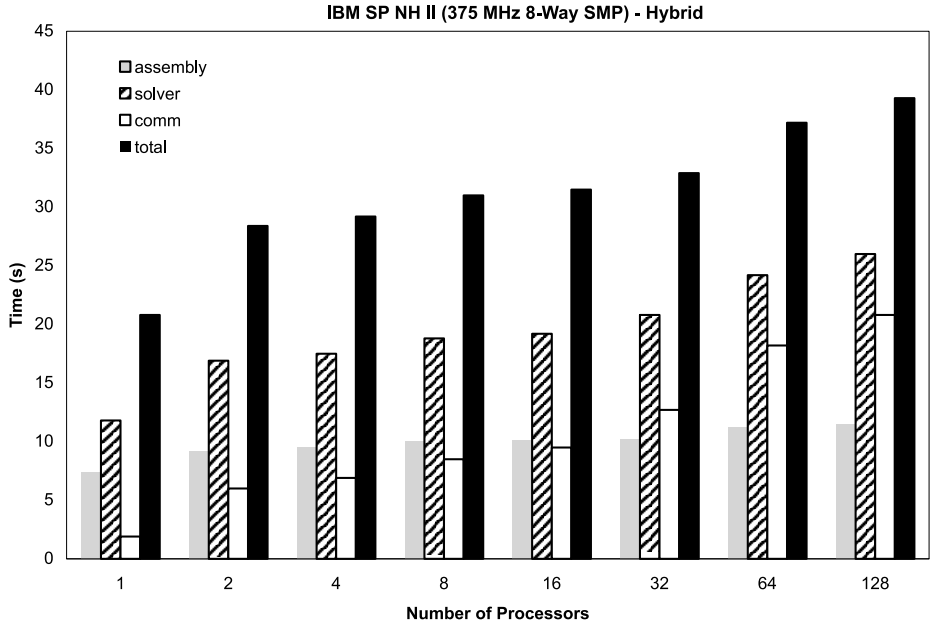
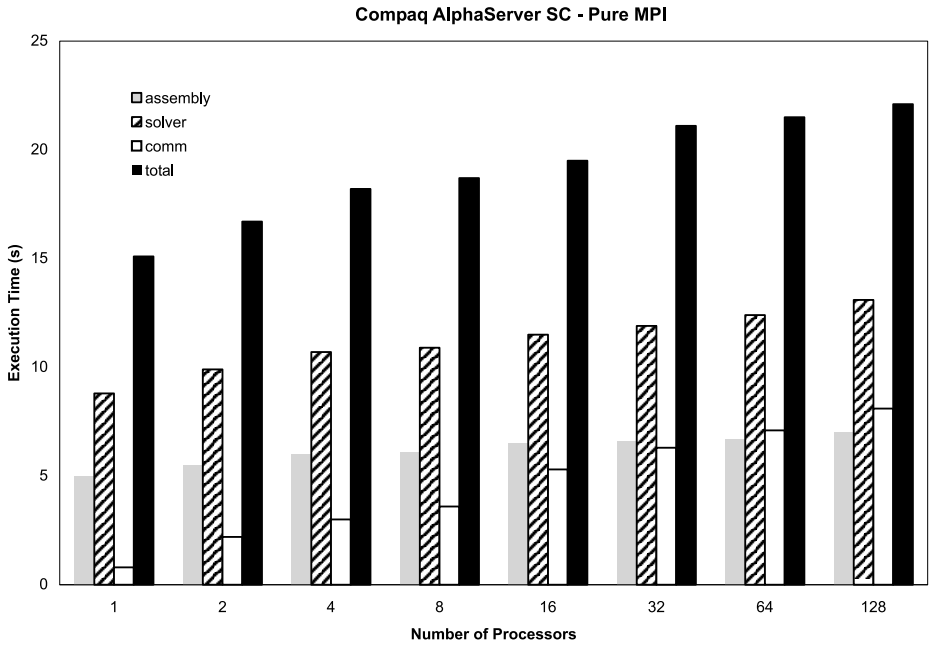


Fig. 9(a–b) Scalability comparison in terms of total execution time for the IBM SP NH II and Compaq SC. The hybrid model uses two OpenMP threads in all cases. (a) Pure MPI on IBM SP NH II, (b) Hybrid on IBM SP NH II.



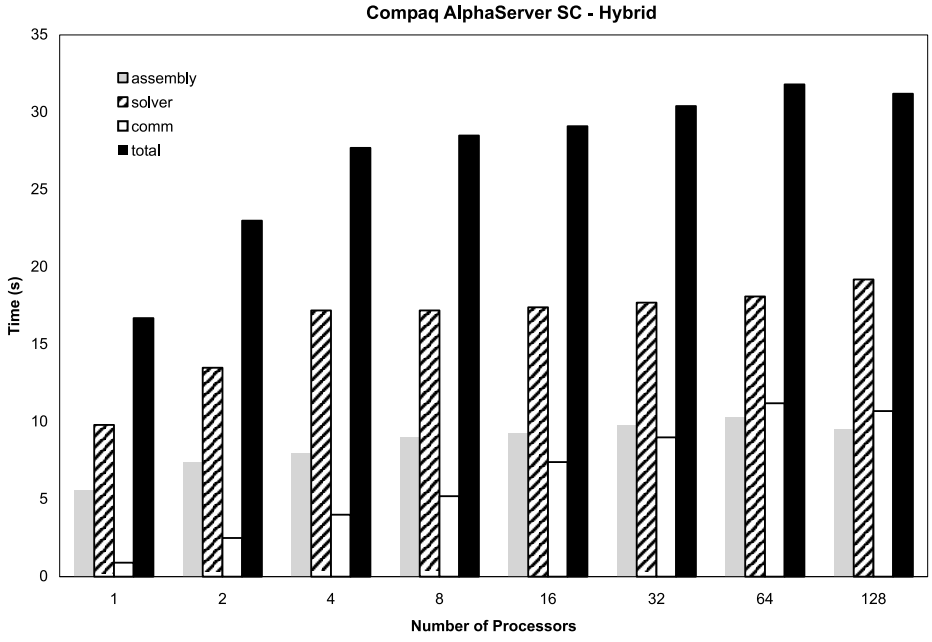


Fig. 9(c–d) Scalability comparison in terms of total execution time for the IBM SP NH II and Compaq SC. The hybrid model uses two OpenMP threads in all cases. (c) Pure MPI on Compaq/Alpha SC, (d) Hybrid on Compaq/Alpha SC.

the IBM SP with the 8-way Nighthawk II nodes that did not support faster MPI communication beyond 4 processors of each node. There may be two possible explanations as to why the pure MPI model outperforms the hybrid model in general. First, when the OpenMP implementation is limited to loop level directives, multiple threads access the same shared array. This can cause synchronization overheads and can disrupt cache optimization by the compiler. For example, when two threads access the same cache line, performance penalties can incur while maintaining cache coherency. Second, in the hybrid model, the MPI communication operations may only be performed by a single thread (e.g., master thread) while the other threads are sleeping. This results in larger message sizes. Moreover, the entire message needs to be accessed by the thread performing the communication operation despite the fact that part of the message has been owned by other threads prior to the communication operation. This may result in unfavorable cache effects depending on how communication buffers are handled in the MPI library.

Poor performance in MPI collective communication operations such as `MPI_Allreduce` is a major factor in limiting scalability for large number of processors in both MPI and hybrid modes. This problem is more severe on the two IBM SP architectures. This observation underscores the need for more optimized collective communication libraries.

Modifying compute intensive loops to reduce dependencies generally improves loop level OpenMP parallelization. For example, the red-black ordering scheme adopted in the matrix assembly and sparse matrix-vector multiplication routines improved OpenMP performance. In the sparse matrix-vector multiplication operation (the single most compute intensive operation in the code), placing the "larger" number of rows loop on the outside and the "smaller" number of diagonals loop on the inside improved both OpenMP and MPI performances. The OpenMP performance improved because loop dependencies were eliminated thus enabling parallelization at the outer loop. The MPI performance improved on the architectures with limited cache because of favorable cache effects.

Although the hybrid MPI-OpenMP parallelization model is a promising new paradigm for emerging architectures, at the time of this writing, we do not believe that it should be adapted to all existing MPI codes. However, we also believe that with future improvements in OpenMP compilers combined with potential limitations in the support of MPI within future SMP nodes, the use of hybrid approach (even for codes that currently scale well with pure MPI) should continue to be explored.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the supercomputing resources and support given by Oak Ridge National Laboratory, National Center for Supercomputing Center, and the San Diego Supercomputer Center in carrying out the development and performance analysis aspects of this paper. The authors are also thankful to the constructive comments provided by the reviewers that contributed to the improvement of this paper.

AUTHOR BIOGRAPHIES

Kumar Mahinthakumar received his B.S. degree in Civil Engineering from the University of Peradeniya in Sri Lanka, M.E. degree in Environmental Engineering from Asian Institute of Technology, Thailand, M.S. degree in Applied Mathematics from Claremont Graduate School, CA, and his Ph.D. degree in Civil Engineering from University of Illinois at Urbana-Champaign. Subsequently, he spent 5 years as a research staff member at the Oak Ridge National Laboratory's Center for Computational Sciences. Currently, he is an assistant professor of Civil Engineering at the North Carolina State University in Raleigh, North Carolina. His research areas are High performance computing, Groundwater flow and transport, Computational fluid dynamics, and Inverse problems.

Faisal Saied received his B.S. degree in Mathematics from Trinity College, Cambridge, a Diplom Mathematik from Göttingen University, and his Ph.D. degree in Computer Science from Yale. Subsequently, he was an Assistant Professor of Computer Science at the University of Illinois at Urbana-Champaign. He is currently a Senior Research Scientist at NCSA, at the University of Illinois, where he manages the Performance Engineering group. His research areas are Numerical Analysis, Parallel Numerical Algorithms, and Performance Analysis on Parallel Architectures.

REFERENCES

Anderson, W.K., Gropp, W.D., Kaushik, D.K., Keyes, D.E., and Smith, B.F. 1999. Achieving high sustained performance in an unstructured mesh CFD application. Proceedings of Supercomputing 99; also available as Argonne preprint ANL/MCS-P776-0899.

Bova, S.W., Breshears, C.P., Cuicchi, C.E., Demirbilek, Z., and Gabb, H.A. 2000. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. *The International Journal of High Performance Computing Applications* 14(1): 49–64.

Cappello, F., and Etiemble, D. 2000. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. *Supercomputing 2000, November 4 –10, Dallas, TX*.

Gropp, W., Lusk W., and Skjellum A. 1999. *Using MPI: Portable parallel programming with the message-passing pnterface*, 2nd Edition, The MIT Press, Cambridge, MA.

Hanebutte, U.R. 2000. Performance of MPI/OpenMP Hybrid Algorithms. Unpublished Report, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA.

Henty, D.S. 2000. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. *Supercomputing 2000, November 4–10, 2000, Dallas, TX*.

Hoeflinger J., Alavilli, P., Jackson, T., and Kuhn, B. 2001. Producing scalable performance with OpenMP: Experiments with two CFD applications. *Parallel Computing* 27: 391–413.

Huyakorn, P.S. and Pinder, G.F. 1983. *Computational Methods in Subsurface Flow*. Academic Press, New York.

Kumar V.P., Grama, A., Gupta, A., and Karypis, G. 1994. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Co., Ltd., Redwood City, CA, p. 597.

Luecke, G.R. and Lin, W.-H. 2001. Scalability and performance of OpenMP and MPI on a 128-processor SGI Origin 2000. *Concurrency and Computation: Practice and Experience* 12:905–928.

Mackay, D., D'Azevedo, E.F., and Mahinthakumar, G. 1998. A study of I/O in a parallel finite-element groundwater transport code. *International Journal of High Performance Computing Applications* 12(3):307–319.

Mahinthakumar, G., Gwo, J.P., Moline, G.R., and Webb, O.F. 1999. Subsurface biological activity zone detection using genetic search algorithms. *ASCE Journal of Environmental Engineering* 125(12):1103–1112.

Mahinthakumar, G., and Saied, F. 1999. Implementation and performance analysis of a parallel multicomponent groundwater transport code. *CD-ROM Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, March 22–24, San Antonio, TX*, Society for Industrial and Applied Mathematicians (SIAM), Philadelphia, PA.

Mahinthakumar, G., Saied, F., and Valocchi, A.J. 1997. Comparison of some parallel Krylov solvers for large scale contaminant transport simulations. *High Performance Computing 1997*, ed. A. M. Teter, pp. 134–139.

Mahinthakumar, G., and West, O.R. 1998. High resolution numerical experiments in support of the ISCOR experiment at Portsmouth. *Unpublished ORNL Report submitted to PORTS*, Nov. 1998 (http://www4.ncsu.edu/gmkumar/ports.pdf).

Saied, F., and Mahinthakumar, G. 1998. Efficient parallel multigrid based solvers for large scale groundwater flow aimulations. *Computers Math. Applic.* 35(7):45–54.

Sarma, K.C., and Adeli, H. 2001. Bilevel parallel genetic algorithms for optimization of large steel structures. *Computer-Aided Civil and Infrastructure Engineering* 16(5): 295–304.

Semprini, L., and McCarty, P.L. 1991. Comparison between model simulations and field results for in-situ biorestoration of chlorinated aliphatics: Part 1. Biostimulation of Methanotropic Bacteria. *Groundwater* 29(3):365–374.

Shin, W.T., Garanzuay, X., Yiacoumi, S., Gu, B., Tsouris, C. and Mahinthakumar, G. 2002. Kinetics of soil ozonation: An experimental and numerical investigation. Submitted to *Journal of Contaminant Hydrology*.

Smith, L., and Kent P. 2000. Development and performance of a mixed OpenMP/MPI quantum Monte-Carlo code. *Concurrency-Practice and Experience* 12(12):1121–1129.

Vadhiyar, S.S., Fagg, G.E., and Dongarra, J. 2000. Automatically tuned collective communications, *Proceedings of Supercomputing 2000, Dallas, TX* (http://www.supercomp.org/sc2000/Proceedings/techpapr/papers/pap270.pdf).

Wallcraft, A.J. 2000. SPMD OpenMP versus MPI for ocean models. *Concurrency and Computation: Practice and Experience* 12:1155–1164.