Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Parallel Computing 30 (2004) 1–20

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Dual-level parallelism for high-order CFD methods ☆

Suchuan Dong, George Em Karniadakis *

Division of Applied Mathematics, Center for Fluid Mechanics, Brown University, Box 1966, 307 Manning street, Providence, RI 02912, USA

Received 23 September 2002; received in revised form 7 May 2003; accepted 7 May 2003

Abstract

A hybrid two-level parallel paradigm with MPI/OpenMP is presented in the context of high-order methods and implemented in the spectral/*hp* element framework to take advantage of the hierarchical structures arising from deterministic and stochastic CFD problems. We take a coarse grain approach to OpenMP shared-memory parallelization and employ a workload-splitting scheme that reduces the OpenMP synchronizations to the minimum. The hybrid algorithm shows good scalability with respect to both the problem size and the number of processors for a fixed problem size. For the same number of processors, the hybrid model with 2 OpenMP threads per MPI process is observed to perform better than pure MPI and pure OpenMP on the SGI Origin 2000 and the Intel IA64 Cluster, while the pure MPI model performs the best on the IBM SP3 and on the Compaq Alpha Cluster. A key new result is that the use of threads facilitates effectively *p-refinement*, which is crucial to adaptive discretization using high-order methods.

Keywords: Dual level parallelism; Message passing interface; OpenMP; Spectral element method; CFD

1. The hybrid model

The hybrid programming (MPI/OpenMP or MPI/Pthreads) paradigm combining message passing and shared-memory parallelism has become popular in the past few years and has been applied in many applications, ranging from costal wave analysis

☆ An abbreviated version appears in Supercomputing Conference 2002 (SC2002).

* Corresponding author. Tel.: +1-401-863-1217; fax: +1-401-863-3369.
E-mail address: gk@dam.brown.edu (G.E. Karniadakis).

0167-8191/$ - see front matter
doi:10.1016/j.parco.2003.05.020

[1,11] to atmospheric research [10], to molecular dynamics analysis [6]. The adoption of this model is facilitated by both the architectural developments of modern supercomputers and the characteristics of a wide range of applications.

The hybrid model can potentially exploit modern supercomputer architectures consisting of shared-memory multi-processor (SMP) nodes more effectively. Modern HPC machines are usually based on distributed memory architecture for scalable performance. However, to address issues of cost-effective packaging and power, manufacturers incorporate shared-memory parallelism at the node level. Thus, most HPC platforms, including the top ten supercomputers in the world (www.top500.org) at the time of this writing, are essentially clusters of shared-memory multiprocessors. The challenge presented to application developers by such machines is that they exhibit hierarchical parallelism with increasingly complex non-uniform memory access data storage. Flat message passing has been the dominant model on these systems. However, compared to the direct data access in shared-memory parallelism, message passing within the SMP node involves the overhead of memory-to-memory data copies and the latencies. A hybrid model combining these two approaches, employing shared-memory parallelism within and message passing across the nodes, seems a natural alternative on such hierarchical systems.

The characteristics of an application also influence the programming paradigm. Irregular applications like molecular dynamics and adaptive mesh refinement, characterized by unpredictable communication patterns and load imbalance, have been the focus of several research efforts with different programming models. For example, OpenMP has been applied to improve the load balance within the SMP node in molecular dynamics analysis [6], mitigating the excessively fine data decompositions in pure MPI computations. The hybrid approach has also been encouraged by the observations that even for irregular codes the shared-memory implementation seem to be able to perform as well as MPI through techniques like runtime performance monitoring [12] and with substantial ease of programming [13].

The hybrid programming model with MPI/threads has been analyzed previously with kernel calculations or simplified model problems [2,6]. These analyses, while providing insights into the hybrid-programming model, are based upon the fine grain shared-memory parallelism, i.e., the shared-memory parallelization using OpenMP or POSIX threads is applied only to major loops [2,6] or sub-routines [5,11]. While feasible for model problems and some applications, this approach is generally not adequate for complex applications, in which either no major loop is readily identifiable or the complicated operations prohibit such loop-level parallelization directives. For example, in the spectral element implementation in the current work the spectral elements are organized using linked lists. The loops for traversing the element lists account for a significant portion of the operations. However, these loops are not readily parallelizable with loop-level directives because of their structures. The pointer operations within such loops also preclude the loop-level directives. Fine grain parallelization also suffers from the drawback of frequent thread creations, destructions and associated synchronizations.

Alternatively, we take a coarse grain approach to the shared-memory parallelization in a fashion similar to MPI data parallel programs. Loft et al. [10] reported such

an approach for atmospheric modeling, and observed a better performance with the hybrid model in some cases. However, the performance of the coarse grain hybrid model relative to the shared-memory and the unified MPI approach, especially in the context of high-order methods, is still not quite clear, while the effect of the hybrid paradigm on dynamic p-refinement has not been studied before.

2. Current objectives

The objective of this work is to design a hybrid approach to take full advantage of the hierarchical structures arising from the spectral/*hp* discretizations of deterministic and stochastic CFD problems.

Deterministic and stochastic CFD systems demonstrate inherent hierarchical structures when discretized with a spectral/*hp* element method. For deterministic Navier–Stokes equations the flow velocity is represented by

$$u(x,y,z,t) = \sum_k \hat{\boldsymbol{u}}_k^*(x,y,t)\mathrm{e}^{\mathrm{i}kz}.$$

This representation applies to 3D unsteady flow problems on geometries with one homogeneous direction while the non-homogeneous 2D domain is of arbitrary complexity. A combined spectral element-Fourier discretization [8] can be employed to accommodate the requirements of high-order as well as the efficient handling of multiply connected computational domain in the non-homogeneous planes. Spectral expansions in the homogeneous direction employ Fourier modes that are decoupled except in the non-linear terms. Each Fourier mode can be solved with the spectral element approach.

Similar hierarchical structures arise from the stochastic analysis using *generalized polynomial chaos* [14]. The key idea of polynomial chaos is to represent stochasticity spectrally with polynomial functionals, first introduced by Wiener for Gaussian random processes. The randomness is absorbed by a suitable orthogonal basis function from the Askey family of polynomials [14]. Subsequently, the Navier–Stokes equation is projected onto the space spanned by the same orthogonal polynomial functions, leading to a set of deterministic differential equations. The flow velocity is thus represented by

$$u(x,y,z,t;\theta) = \sum_m \hat{\boldsymbol{u}}_m^*(x,y,z,t)H(\xi(\theta)),$$

where $H(\cdot)$ is the Hermite polynomial functional, ξ is the Gaussian variable and θ is a random parameter. As a result, the Navier–Stokes equations are reduced to a set of equations for the expansion coefficients (called random modes), which are 3D deterministic functions of both space and time. The random modes are decoupled except in the non-linear terms, and can be solved with the spectral/*hp* element method [8].

The *inherent* hierarchical structures in CFD problems suggest a multi-level parallelization strategy. At the top-most level are groups of MPI processes. Each group

computes one random mode. At the next level, the 3D domain of each random mode is decomposed into sub-domains, each consisting of a number of spectral elements. Each MPI process within the group computes one sub-domain. At the third level, multiple threads are employed to share the computations within the sub-domain. Compared with the flat message-passing model, this multilevel parallelization strategy reduces the network latency overhead because a greatly reduced number of processes are involved in the communications at each level. This enables the application to scale to a large number of processors more easily.

The pure MPI approach for the deterministic problems has been documented in [3,4,7]. In this approach a straight-forward mapping of the Fourier modes onto the processors is employed, resulting in an efficient and balanced computation where the 3D problem is decomposed into 2D problems using multiple 1D FFTs. One drawback of this approach (and 1D domain decompositions in general), however, is that the number of Fourier planes in the homogeneous direction imposes an upper limit on the number of processors that can be employed. When the maximum number of processors is used, the wall clock time per time step is solely determined by the speed of computations of the 2D plane. Increasing the number or the order of the elements in the non-homogeneous 2D domain, which is often desired for resolving vortical flows, cannot be further balanced by increasing the number of processors accordingly. However, this limit can be eliminated by: (1) further decomposing the 2D plane through e.g. METIS [9] or (2) using shared-memory parallelism.

Both approaches can be effective but here we follow the second one, which is relatively less complicated from the data structure standpoint. Specifically, we parallelize the computations in the non-homogeneous 2D plane via a coarse grain shared-memory parallelism with OpenMP. The resulting hybrid MPI/OpenMP approach eliminates the limit on the number of processors that can be exploited and dramatically expands the capability of the pure MPI model. The objectives of this paper are:

- to present the coarse grain approach for the hybrid MPI/OpenMP programming paradigm,
- to investigate the influence of multi-threading on the p-type refinement in high-order methods, and
- to examine the performance of different programming models in the context of high-order methods.

3. Spectral/*hp* element method

A spectral/*hp* element method [8] is employed to discretize in space, a semi-implicit scheme in time, and polynomial chaos in the random direction. These algorithms are implemented in C++ in *NekTar*, a general-purpose high-order CFD code for incompressible, compressible and plasma unsteady flows in 3D geometries. The code uses meshes similar to standard finite element and finite volume meshes, consisting of structured or unstructured grids or a combination of both. The formulation is also similar to those methods, corresponding to Galerkin and discontinuous Galerkin

projections for the incompressible, compressible and plasma governing equations, respectively. Flow variables are represented in terms of Jacobi polynomial expansions. This new generation of Galerkin and discontinuous Galerkin high-order methods implemented in *NekTar* extends the classical finite element and finite volume methods [8]. The additional advantage is that not only the convergence of the discretization but also the solution verification can be obtained with *p-refinement* (refinement over the interpolation orders). However, the computational complexity associated with p-refinement ($O(p^{d+1})$, where d is the spatial dimension) has limited its use in large-scale simulations to date.

4. Coarse grain shared-memory parallelization

Fig. 1 provides a schematic of the parallel hybrid paradigm. The flow domain is decomposed in the homogeneous z-direction. At the outer level multiple MPI processes are employed, with each process computing one sub-domain. At the inner level, within each MPI process multiple OpenMP threads conduct the computations in the sub-domain in parallel. Data exchange across sub-domains is implemented with MPI. Within each process, access to shared objects by multiple threads is coordinated with OpenMP synchronizations.

Specifically, a single parallel region encloses the time-integration loops at the topmost level to enable the coarse grain approach. This avoids the overhead associated with frequent thread creations and destructions inherent in fine grain programs. The number of threads in different MPI processes can vary to offset the load imbalance in different sub-domains. The OpenMP threads share the computations within a

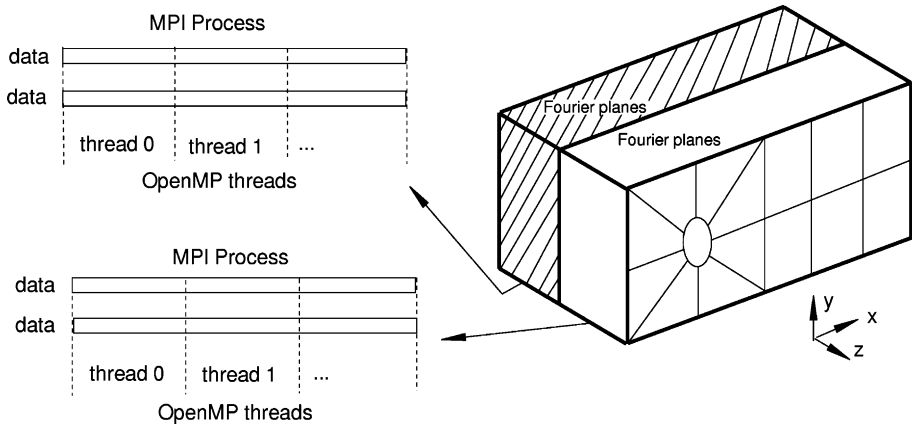

Fig. 1. Schematic showing domain decomposition and the OpenMP threads within MPI processes. The flow region is decomposed in the homogeneous direction (z). Each MPI process is assigned to one subdomain. Multiple OpenMP threads within the MPI process work on disjoint sections of the data in parallel.

sub-domain by working on disjoint groups of elements or disjoint sections of the vectors (of roughly the same size). The vector length, the element number, and the number of entries in the linked lists are split based on the number of threads. This computation is done only once at the pre-processing stage, and the results are stored in a shared table, which can be referenced based on thread IDs. In this configuration each thread works on a large section of a vector with contiguous memory, improving the cache-hit rate. The MPI calls are handled by only one thread within each process. Advantageous over pure MPI programs on SMP nodes, this configuration assembles the nodal messages into a single large message and thus reduces the network latency overhead. The OpenMP barriers surrounding the MPI call constitute the associated overhead.

OpenMP barrier is the main type of OpenMP synchronization involved. The barriers exist at (1) the beginning of time integration sub-steps due to data dependence, (2) the beginning and end of MPI calls to ensure the data completeness, (3) the inner products in the conjugate gradient iterative solver for boundary-mode solves, and (4) the switching points between global and local operations. The majority of OpenMP barriers occur at the switching points between global and local operations. The workload splitting dictated by OpenMP loop-level directives would result in excessive such barriers and compromise the performance. We first illustrate this situation and then introduce a workload splitting scheme that completely eliminates these barriers.

The following code segment from the pure MPI implementation is for calculating the non-linear term $U\frac{\partial U}{\partial x} + V\frac{\partial U}{\partial y} + W\frac{\partial U}{\partial z}$ in Navier–Stokes equations.

```
... // dU/dz is calculated here
dvmul(nq, W->base_h, 1, Uz->base_h, 1, grad_wk, 1); // grad_wk = W*dU/dz
U->Grad_h(U->base_h, Ux->base_h, Uy->base_h, NULL, trip); // calculate dU/dx, dU/dy
dvvtvp(nq, U->base_h, 1, Ux->base_h, 1, grad_wk, 1, grad_wk, 1); // grad_wk += U*dU/dx
...
```

The term $W\frac{\partial U}{\partial z}$ is computed on the first line and stored in the temporary vector `grad_wk`. On the second line the derivatives $\frac{\partial U}{\partial x}$ and $\frac{\partial U}{\partial y}$ are computed in the function `Grad_h( )` and stored in `Ux` and `Uy`, respectively. Inside `Grad_h( )` the derivatives are computed element by element (local operations). In the implementation the structures `Uz` and `Uy` point to the same object in memory. On the third line the term $U\frac{\partial U}{\partial x}$ is computed and added to the temporary vector `grad_wk`. The vector length, `nq`, is the product of the number of Fourier planes on the current processor and the total number of quadrature points in the `x`–`y` plane. The two LAPACK-style vector routines, `dvmul` and `dvvtvp`, manipulate the entire vector (global operations).

In the hybrid model, the straightforward approach is to adopt loop-level OpenMP directives in functions `dvmul`, `Grad_h` and `dvvtvp`, which unfortunately would result in excessive barriers. With loop-level directives the vector length, `nq`, would be split based on the number of threads. So each thread computes a section of the vectors with length approximately $[\frac{nq}{P}]$, where P is the number of OpenMP threads. In-

side the function `Grad_h( )` the workload could be split based on the number of elements, `nel`, since the computations are performed element by element. Thus, each thread works on about $[\frac{nel}{P}]$ elements. Because `Uz` and `Uy` point to the same object in memory an OpenMP barrier is required between the first and the second lines to ensure that all the operations on `Uz` have completed before the operations on `Uy` canstart. By the same token, another barrier is needed between the second and the third lines to ensure that all the results about $\frac{\partial U}{\partial x}$ have been written to `Ux->base_h` before they can be used on the third line. With loop-level parallelization directives, these barriers cannot be avoided no matter what scheduling policies (static, dynamic, guided) are employed inside `dvmul`, `dvvtvp` and `Grad_h`. These barriers persist because they are at the switching points between global and local operations. An additional complexity is that in function Grad_h( ) the loop for traversing the element list is not readily parallelizable with OpenMP directives.

To eliminate these barriers a consistent workload splitting needs to be employed across the global and local operations. We note that in the implementation the global vector (e.g. `Uz->base_h`) with length `nq` is a concatenation of the local vectors on all the elements. Therefore we can still split the number of elements into groups with size $[\frac{nel}{P}]$ inside the function `Grad_h( )`. The global vectors, however, should be split in a fashion such that each section of the vectors is a concatenation of the local vectors of that particular group of elements the thread works on inside `Grad_h( )`. With this scheme each thread operates on its own group of elements and the corresponding sections of the global vectors. These barriers are thus completely eliminated.

5. Benchmarking procedure

The performance of the hybrid paradigm is tested with simulations of turbulent flow past a circular cylinder at Reynolds number $Re = 500$ based on the inflow velocity and the cylinder diameter. The dimension of the flow domain is π in z direction (homogeneous). We use 412 triangular elements in the x–y plane and 16 modes (32 planes) in z direction (Fig. 2). A uniform inflow is prescribed at the inlet. Outflow boundary conditions are applied at the outlet and the upper/lower boundaries of the flow domain. Periodic conditions are imposed in the homogeneous direction. The initial flow is a 3D fully developed turbulent flow generated in a separate run. A third-order stiffly-stable scheme is used for time-integration [8].

Benchmarks are performed on the following four platforms:

- IBM SP3 (BlueHorizon) at SDSC (375 MHz Power3);
- SGI Origin 2000 at NCSA (250 MHz MIPS R10000);
- Compaq Alpha Cluster (LeMieux) at PSC (1 GHz Alpha EV68);
- Intel IA64 Cluster (Titan) at NCSA (800 MHz Itanium).

We use compiler optimization options "-Ofast" on NCSA Origin, "-O3 -qarch = pwr3 –qtune = pwr3" at SDSC, "-fast" at PSC, and "-O3 -ip -IPF_fma"

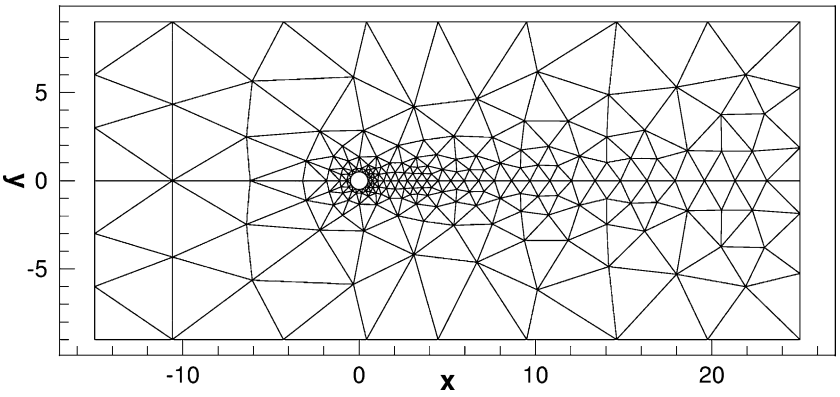

Fig. 2. Unstructured mesh in a 2D slice x–y plane (412 elements) for the test flow problem.

on NCSA IA64 Cluster. On the NCSA Origin the timing results are collected in the dedicated mode on a 128-CPU host. At SDSC and PSC, and on the NCSA IA64 cluster the benchmarks are performed with exclusive access to the nodes allocated to the job. Wall clock time is collected by the `MPI_Wtime( )` calls embedded in the code. Timings are accumulated across all MPI processes and divided by the number of processes to give the mean values per processor. The command "`dplace`" has been employed to optimize memory placement policies.

The first group of tests is designed to examine the scaling with respect to the number of processors for a fixed problem size. In particular, a fixed (spectral) polynomial order of $N_{\text{order}} = 15$ is used for these cases, while the number of MPI processes is varied from 1 through 16 (1, 2, 4, 8, 16), and the number of OpenMP threads per process is varied between 1 and 8 (1, 2, 4, 6, 8) on the NCSA Origin and at SDSC, between 1 and 4 (1, 2, 3, 4) at PSC, and between 1 and 2 on the NCSA IA64 Cluster. The total number of processors (number of processes $\times$ number of threads per process) varies from 1 to 128. The second group of tests is to check the scalability with respect to the problem size. The same mesh as in the first group is used while the order of the interpolating polynomials and the number of threads per process are varied.

6. Performance results

6.1. Fixed problem size

Fig. 3 shows the wall clock time per step (top row) and the parallel speedup (bottom row) versus the total number of processors for the pure MPI, pure OpenMP, and the hybrid runs on the SGI Origin. Since there are 32 Fourier planes in the homogeneous direction, a maximum of 16 processors can be used in the MPI runs. Up to 8 threads per process are used for the OpenMP and hybrid runs because the

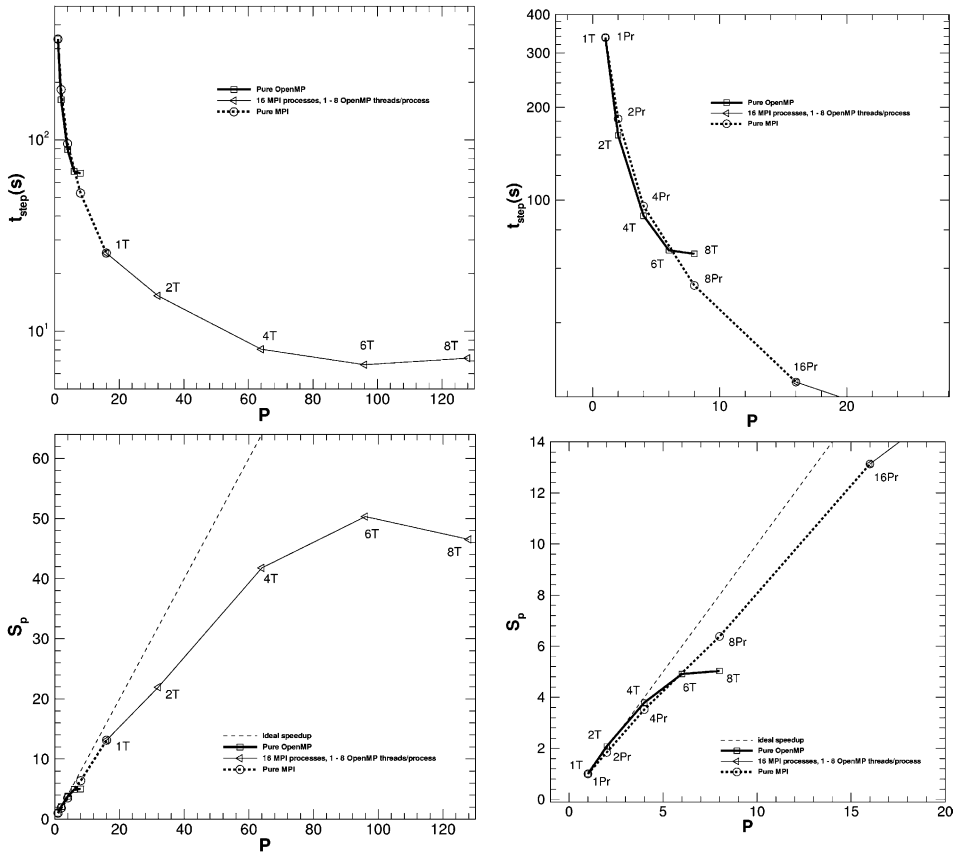

Fig. 3. Wall clock timing (top row) and parallel speedup (lower row) for three programming models (pure MPI, pure OpenMP, hybrid) on NCSA SGI Origin 2000: top left, time per step versus total number of processors; top right, enlarged view of the figure on top left; bottom left, speedup factor versus total number of processors; bottom right, enlarged view of the figure on bottom left. The total number of processors is the multiplication of the number of MPI processes and the number of threads per process. Pure MPI: 1 thread per MPI process; Pure OpenMP: only 1 process; Hybrid: multiple MPI processes, multiple threads per process. Meanings of labels: 4T means 4 threads per process; 8Pr means 8 MPI processes.

performance deteriorates beyond this number for the problem size in consideration. For up to 4 processors, the pure MPI and pure OpenMP demonstrate comparable performance, with the OpenMP slightly better. The pure OpenMP run demonstrates a super-linear speedup for 2 processors. This is attributed to the two factors: (1) the large dataset size the OpenMP handles (the whole flow domain), and (2) the proximity of the two processors to the memory. On the SGI Origin 2000 a node consists of two processors and one piece of memory. Two such nodes are attached to a router, and the routers form a hypercube for up to 64 processors. Beyond 64 processors a hierarchical hypercube is employed. It follows that on this architecture the application should still achieve a good performance with up to four OpenMP threads if

these threads are scheduled on the two nearby nodes attached to the same router. This is indeed the case for the pure OpenMP and all the hybrid runs. Beyond 4 threads the performance of the OpenMP quickly deteriorates due to the overhead associated with remote memory access (NUMA). The pure MPI demonstrates a near-linear speedup up to the maximum number of processors (16) that can be employed.

The hybrid runs demonstrate a good speedup factor for up to 4 threads per process. The speedup drops from 96 processors (16 processes, 6 threads/process) to 128 processors (16 processes, 8 threads/process). These observations are consistent with our analysis in the previous paragraph. We would like to emphasize that the hybrid model extends substantially the performance curve of the pure MPI. The hybrid model reduces the minimum wall clock time that the pure MPI can achieve by a factor of 3.2 with 4 OpenMP threads per process, and a factor of 3.8 with 6 OpenMP threads per process.

Fig. 4 shows the wall clock time per step (top row) and the parallel speedup (bottom row) versus the number of processors for the three programming models on the IBM SP3 at SDSC. The results demonstrate the same trend as the ones observed on the Origin. However, compared with the Origin results the wall clock time on the SP is smaller by a factor of 2–3. With the 8-way SMP nodes on the SP, the applications cannot benefit further from the shared-memory parallelism with more than 8 threads per process. Pure MPI performs consistently better than the pure OpenMP, and the performance gap widens as the number of processors increases. The pure MPI run achieves a super-linear speedup for 4, 8 and 16 processors. This usually indicates that the application benefits from the larger aggregate cache size as the number of processors increases. Compared to the maximum speedup achieved by pure MPI, the hybrid model increases the value by a factor of 2.6 with 4 threads per node, and a factor of 3.1 with 6 threads per process. The scaling of the hybrid model is good for up to 6 threads per process for this problem size. Then it flattens as the number of threads per process further increases.

In Fig. 5 we plot the wall clock time per step (top row) and the parallel speedup (bottom row) versus the total number of processors on the Compaq Alpha cluster at PSC. Since the cluster consists of 4-processor nodes, a maximum of 4 OpenMP threads per process is deployed in these tests. The data shows that the wall clock time on the PSC Compaq cluster is nearly half of that on the SP at SDSC, and an even smaller fraction of that on the Origin. The pure MPI runs demonstrate near-linear speedup. The pure OpenMP run with 2 threads shows a performance comparable to the pure MPI run with two processes, although the latter performs slightly better. The pure MPI run with 4 processes performs considerably better than the pure OpenMP run with 4 threads. This performance gap between the pure MPI and OpenMP is attributed to the memory bandwidth contention between multiple threads within the node in OpenMP runs. Since the threads within the node share the memory bandwidth, severe contentions may occur with a fair number of threads, leading to a reduced effective memory bandwidth. In contrast, multiple MPI processes are started on different nodes in pure MPI runs, with no memory bandwidth contention on any node. Similar to what has been observed on other systems, the

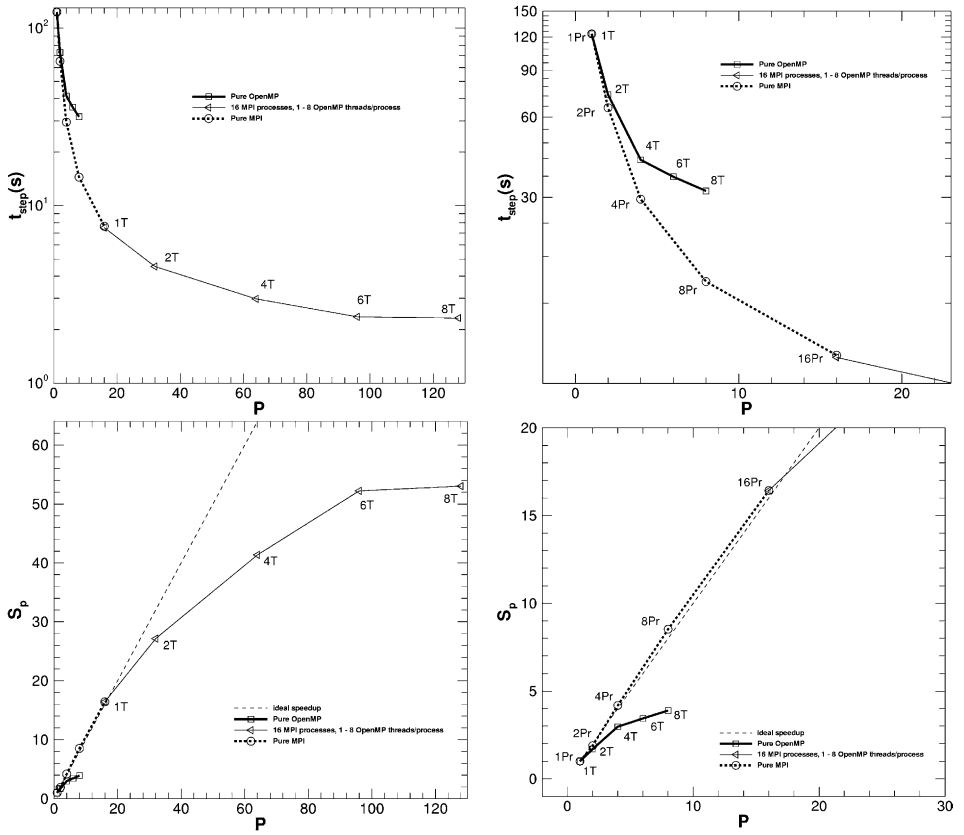

Fig. 4. Wall clock timing (top row) and parallel speedup for the three programming models (pure MPI, pure OpenMP, hybrid) on SDSC IBM SP: top left, time per step versus total number of processors; top right, enlarged view of the figure on top left; bottom left; speedup factor versus total number of processors; bottom right, enlarged view of the figure on bottom left. See caption of Fig. 3 for meanings of the labels.

hybrid runs greatly extend the performance curve of the pure MPI runs. With 2 and 4 threads per MPI process speedup factors of 1.7 and 2.2 are observed, respectively. The maximum number of processors in this test on Compaq Alpha (64 processors) produces a lower parallel speedup (32) compared with those on IBM SP (speedup 41) and SGI Origin (speedup 42).

Fig. 6 shows the wall clock timing (top row) and the parallel speedup (bottom row) collected on the IA64 cluster at NCSA. A maximum of two OpenMP threads per MPI process is deployed for the hybrid and pure OpenMP runs on the dual-processor nodes of the cluster. Like on the other platforms, the MPI runs demonstrate near-linear speedup. An intriguing observation is that the OpenMP run and the hybrid runs demonstrate super-linear speedup. Deploying two threads per process reduces the wall clock time of the corresponding MPI run (same number of MPI processes, single thread per process) by more than half in all the cases. The high

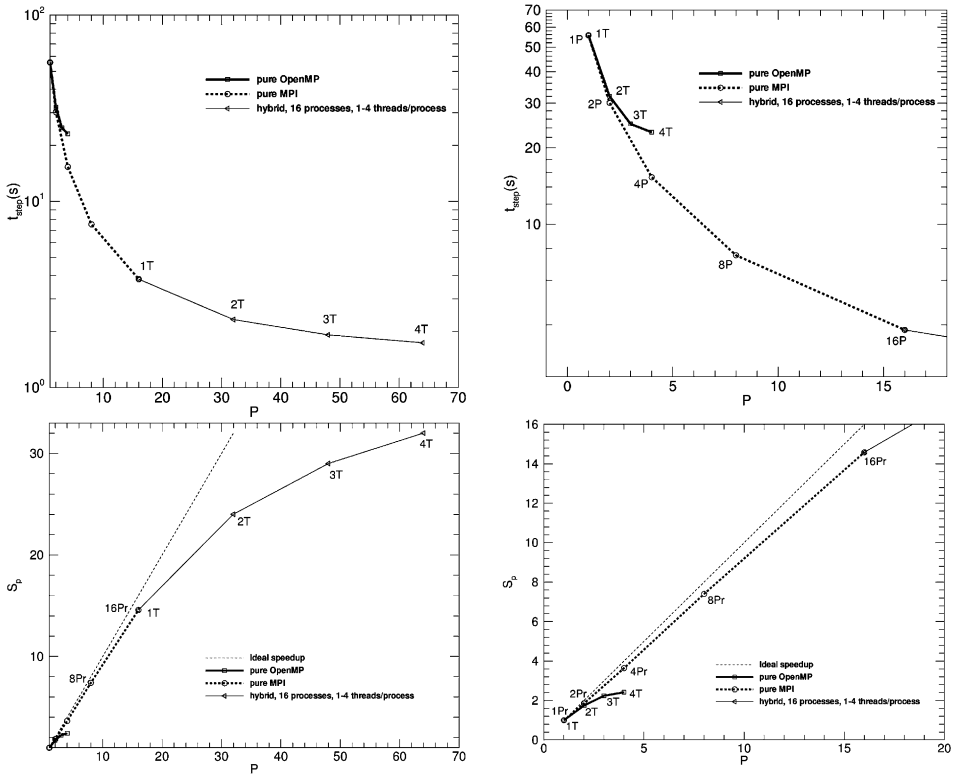

Fig. 5. Wall clock (top row) and parallel speedup for the three programming models (pure MPI, pure OpenMP, hybrid) on PSC Compaq Cluster: top left, time per step versus total number of processors; top right, enlarged view of the figure on top left; bottom left, speedup factor versus total number of processors; bottom right, enlarged view of the figure on bottom left. See caption of Fig. 3 for the meanings of the labels.

scalability of the IA64 cluster is attributed to its high memory bandwidth as compared to the other three platforms and the large L3 cache of the Itanium processor.

To better understand the scaling of the three programming models, we decompose the total execution time of each MPI process into inter-process communication time, thread serial computation time, thread synchronization (mainly OpenMP barriers) time, and thread parallel computation time. Since the MPI inter-process communications are handled by one thread within each process, they will be classified as serial operations in terms of the threads. We would call the sum of the inter-process *communication time* and the *thread serial computation time* the total serial operation time, which can be easily measured. Because the OpenMP barriers are scattered it is difficult to measure this synchronization time accurately. Fig. 7 shows the percentage of the communication cost and the total serial operation cost against the total execution time versus the total number of processors on the four platforms. On all four platforms the communication cost accounts for the major portion of the serial

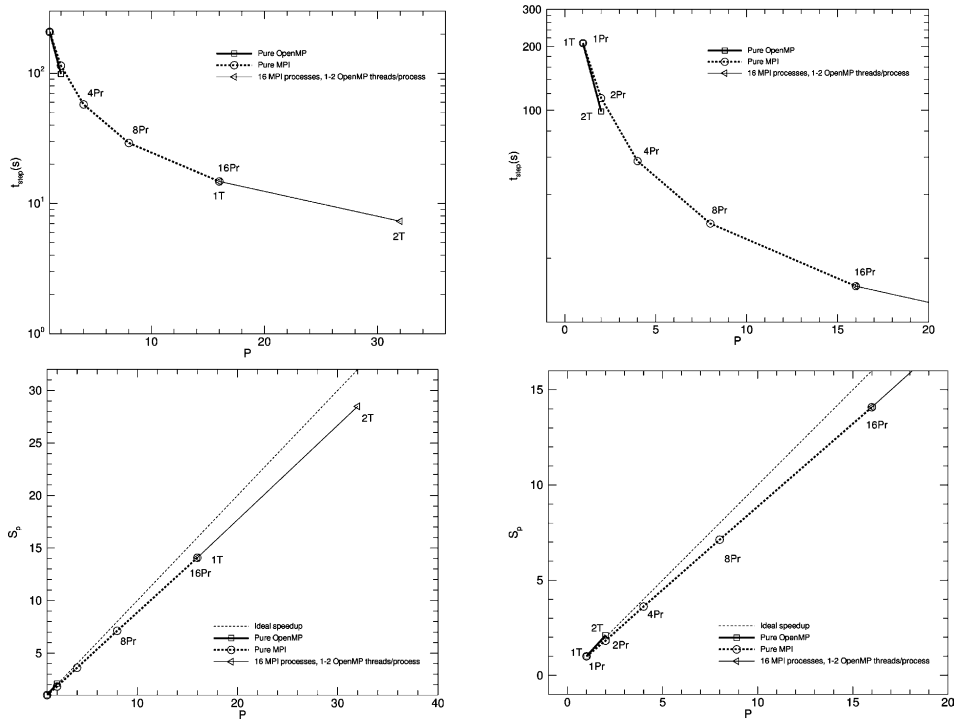

Fig. 6. Wall clock timing (top row) and parallel speedup (lower row) for the three programming models (pure MPI, pure OpenMP, hybrid) on NCSA IA64 (Titan) Cluster: top left, time per step versus total number of processors; top right, enlarged view of the figure on top left; bottom left, speedup factor versus total number of processors; bottom right, enlarged view of the figure on bottom left. See caption of Fig. 3 for the meanings of labels.

operations in all except the single-process runs. As the number of processes increases the communication cost increases drastically in the MPI runs. With 16 processes (the maximum for pure MPI) the communication accounts for about 50% of the total execution time on SDSC SP, NCSA Origin and IA64 cluster, and an even higher percentage on the PSC Compaq cluster. With multiple threads in the OpenMP and hybrid runs, the communication cost increases only slightly compared to that of the MPI case with the same number of processes. This is because only one thread handles the inter-process communications on each node. Increasing the number of threads per process does not add to the overall communication cost so much as increasing the number of processes, although an additional thread synchronization is involved in MPI communication routine in the hybrid runs.

6.2. Relative performance of three models for the same number of processors

The relative performance of the three programming models—pure MPI, OpenMP and hybrid—with the same total number of processors is also of interest. Tables 1–4

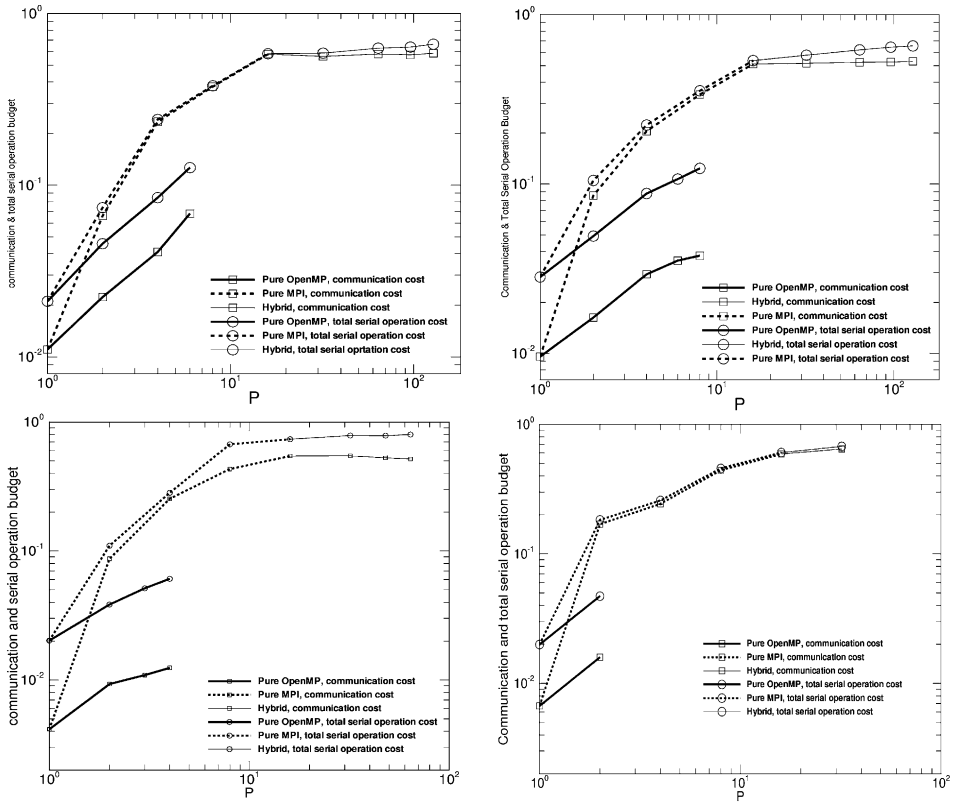

Fig. 7. Percentage of communication and total serial operation cost versus total number of processors on NCSA Origin 2000 (top left), SDSC IBM SP (top right), PSC Alpha Cluster (bottom left) and NCSA IA64 Cluster (bottom right). Total serial operations (concerning OpenMP) include inter-process communications and other serial computations such as the FFTW calls. The communication time and the total serial operation time are normalized by the total execution time. The total number of processors, P, is the product of the number of MPI processes and the number of threads per process. Pure OpenMP:1 process, 1, 2, 4, 6 and 8 threads/process; Pure MPI: 1 thread/process, 1, 2, 4, 8 and 16 processes; Hybrid: 16 processes, 1, 2, 4, 6 and 8 threads/process on Origin and SP, 1,2,3,4 threads/process on Compaq Cluster, and 1,2 threads/process on IA64 Cluster.

summarize the performance results of the three models for a total number of 4, 8 and 16 processors on the four platforms, respectively. On the Origin and IA64 cluster, the hybrid model with 2 threads per process is observed to perform better than pure MPI and OpenMP. The hybrid model with 4 threads per process performs the best in some cases on the Origin. On the other hand, the pure MPI performs better than both the hybrid model and the pure OpenMP on the IBM SP and the Compaq Alpha cluster, though some configurations of the hybrid runs show performances comparable to pure MPI. It is worth pointing out that even on these two systems the hybrid model is able to exploit a larger number of processors and thus achieve a much smaller wall clock time per step.

Table 1
Performance comparison of three programming models (pure MPI, pure OpenMP, hybrid) for a set of fixed number of processors on SGI Origin 2000 at NCSA

CPU	MPI		Hybrid						OpenMP	
	Process	Thread/process	Process	Thread/process	Process	Thread/process	Process	Thread/process	Process	Thread/process
4	4	1	2	2	N/A	N/A	N/A	N/A	1	4
	95.56 (s)		87.30 (s)		N/A		N/A		88.87 (s)	
8	8	1	4	2	2	4	N/A	N/A	1	8
	52.75 (s)		48.41 (s)		47.8 (s)		N/A		66.88 (s)	
16	16	1	8	2	4	4	2	8	N/A	N/A
	25.64 (s)		24.09 (s)		26.02 (s)		36.47 (s)		N/A	

For each number of CPU (the left column), the first row shows the configurations (the number of MPI processes and the number of threads per process); the second row shows the wall clock time per step in seconds.

Table 2
Performance comparison of three programming models (pure MPI, pure OpenMP, hybrid) for a set of fixed number of processors on the IBM SP3 at SDSC

CPU	MPI		Hybrid						OpenMP	
	Process	Thread/process	Process	Thread/process	Process	Thread/process	Process	Thread/process	Process	Thread/process
4	4	1	2	2	N/A	N/A	N/A	N/A	1	4
	29.51 (s)		34.36 (s)		N/A		N/A		41.46 (s)	
8	8	1	4	2	2	4	N/A	N/A	1	8
	14.66 (s)		17.19 (s)		21.5 (s)		N/A		31.67 (s)	
16	16	1	8	2	4	4	2	8	N/A	N/A
	7.65 (s)		8.7 (s)		10.89 (s)		16.55 (s)		N/A	

See Table 1 for the meaning of different columns.

Table 3
Performance comparison of three programming models (pure MPI, pure OpenMP, hybrid) for a set of fixed number of processors on the Compaq Alpha cluster at PSC

CPU	MPI		Hybrid						OpenMP	
	Process	Thread/process	Process	Thread/process	Process	Thread/process	Process	Thread/process	Process	Thread/process
4	4	1	2	2	N/A	N/A	N/A	N/A	1	4
	15.32 (s)		17.51 (s)		N/A		N/A		23.05 (s)	
8	8	1	4	2	2	4	N/A	N/A	N/A	N/A
	7.53 (s)		8.98 (s)		12.06 (s)		N/A		N/A	
16	16	1	8	2	4	4	N/A	N/A	N/A	N/A
	3.82 (s)		4.53 (s)		6.3 (s)		N/A		N/A	

See Table 1 for the meaning of different columns.

Table 4
Performance comparison of three programming models (pure MPI, pure OpenMP, hybrid) for a set of fixed number of processors on the IA64 Cluster at NCSA

CPU	MPI		Hybrid						OpenMP	
	Process	Thread/process	Process	Thread/process	Process	Thread/process	Process	Thread/process	Process	Thread/process
4	4	1	2	2	N/A	N/A	N/A	N/A	N/A	N/A
	57.55 (s)		51.82 (s)		N/A		N/A		N/A	
8	8	1	4	2	N/A	N/A	N/A	N/A	N/A	N/A
	29.16 (s)		27.27 (s)		N/A		N/A		N/A	
16	16	1	8	2	N/A	N/A	N/A	N/A	N/A	N/A
	14.76 (s)		14.04 (s)		N/A (s)		N/A		N/A	

See Table 1 for meanings of different columns.

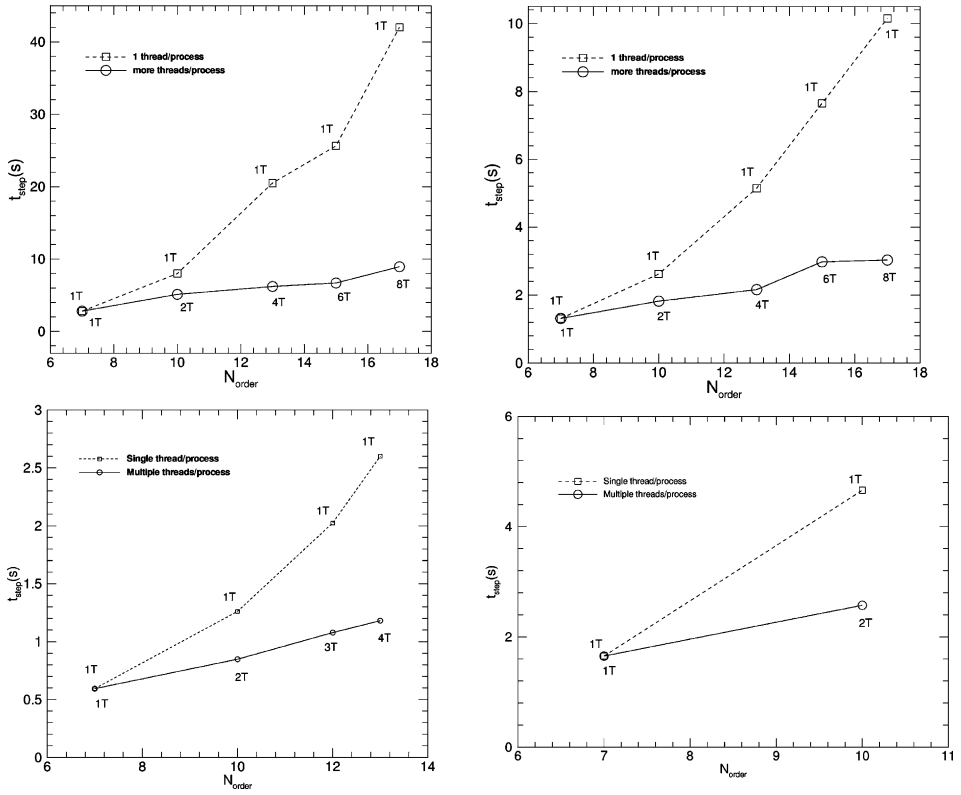

Fig. 8. Time per step in seconds versus the order of expansion polynomials on NCSA Origin 2000 (top left), SDSC IBM SP (top right), PSC Alpha Cluster (bottom left) and NCSA IA64 Cluster (bottom right). The order of polynomial varies as 7, 10, 13, 15 and 17 on Origin and SP, as 7, 10, 12, 13 on Alpha Cluster, and as 7 and 10 on IA64 Cluster. Dashed line corresponds to 1 thread/process. Solid line shows the result when the number of threads per process is increased accordingly. 16 MPI processes are used for all cases. The label near each symbol shows the number of threads per process (e.g., 4T means 4 threads per process).

6.3. Variable problem size

Next we examine the scaling of the hybrid model with respect to the problem size. The pure MPI approach has been shown to demonstrate good scalability with respect to the number of Fourier planes in the homogeneous direction on a dozen platforms [7]. However, it does not scale well as the problem size increases in the x–y plane through, e.g., *hp*-refinement. Here we concentrate on the scaling of the hybrid paradigm as the grid in the non-homogeneous x–y plane is refined. One advantage of the spectral/*hp* element method is that the grid can be refined by either (1) increasing the number of elements (called h-type refinement), which results in algebraic decay of the numerical error, or (2) increasing the order of interpolating polynomials while keeping the number of elements fixed (called *p-type refinement*), which results in exponential decay of the numerical error for smooth functions. We consider the

p-type refinement because it is typical of polynomial spectral methods. We use the same mesh as in the first group of tests, but vary the order of the interpolating polynomials. The number of Fourier planes in the homogeneous z direction is fixed at 32, and16 MPI processes are used for all the following cases. On the Origin and SP, five different problem sizes are tested corresponding to the polynomial orders of 7, 10, 13, 15 and 17. For each problem size we first test the case with one thread per process. Then we increase the number of threads per process approximately in proportion to the cost increase in the single thread/process cases, with 1, 2, 4, 6 and 8 threads per process for the five problem sizes. On the Compaq cluster, three different problem sizes are considered corresponding to the polynomial orders of 7, 10 and 13, and the number of OpenMP threads per process is varied among 1, 2 and 4 (The case of polynomial order 12, with 3 threads per process is also included in the plot). On the IA64 cluster we consider two problem sizes corresponding to polynomial orders 7 and 10, and the number the OpenMP threads per process is varied between 1 and 2. Fig. 8 shows the wall time per step versus the order of the interpolating polynomials on these four platforms. The execution time increases algebraically for the runs with a single thread per process as the order of polynomials increases. When the number of threads per process is increased in proportion, the execution time increases only slightly, indicating that the hybrid model demonstrates good scalability with respect to the problem size on these systems.

7. Concluding remarks

Based on the results on the SGI Origin 2000, the IBM SP, the Compaq Alpha Cluster and the IA64 Itanium cluster we can draw the following conclusions:

1. With the same number of processors, the hybrid model with 2 threads per process performs the best on the SGI Origin and the IA64 cluster, while pure MPI performs the best on the IBM SP and Compaq Alpha cluster.
2. For high-order methods the cost increase associated with the p-type refinement (the refinement over the interpolation order) can be effectively counter-balanced by employing multiple threads. The use of threads in conjunction with p-refinement for high-order methods is an effective way of performing adaptive discretizations. When p-refinement is performed, one can potentially retain a constant wall clock time per step by varying the number of threads per process proportionately. This is a significant new result that can potentially change the way high-order methods are implemented on parallel computers.
3. The performance of the hybrid paradigm is affected by the MPI message-passing library, the OpenMP shared-memory library and the underlying hardware. These factors also influence the performance of the pure MPI and pure OpenMP computations. The performance is also affected by the problem size. Increasing the problem size by using higher spectral order favors the hybrid model as this will localize the work in the spectral element method which will then be computed efficiently using multiple threads.

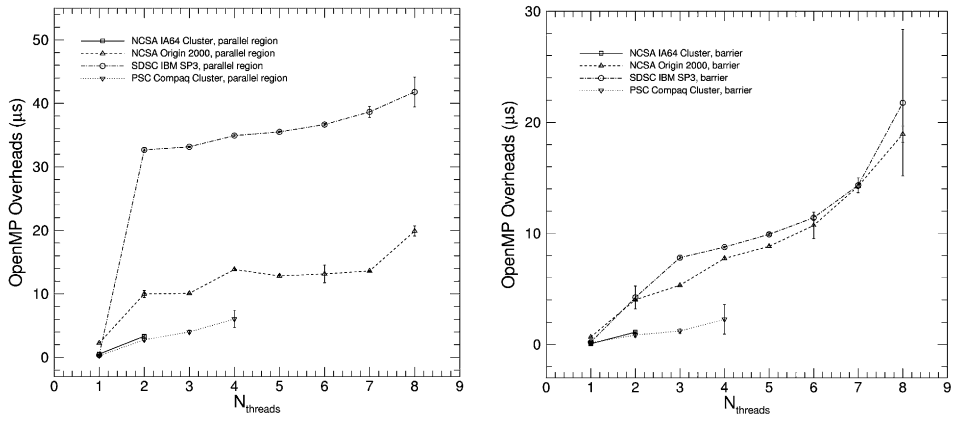

Fig. 9. Overheads of OpenMP parallel regions (left) and barriers (right) vs. the number of threads on IA64 cluster and SGI Origin 2000, IBM SP3, and Compaq cluster. Results obtained with the OpenMP micro-benchmark suite.

4. The relative performance of the hybrid paradigm and MPI is determined by the tradeoff between several factors. On the one hand, the hybrid model takes better advantage of the faster intra-node communications within the SMP nodes. Compared to pure MPI computations on the same number of processors, there are also fewer processes involved in the communications in the hybrid computations. The hybrid model reduces the number of communications and increases the message size by assembling the nodal messages into a single large one, thus reducing the network latency overhead. On the other hand, thread management overhead (creation/destruction and synchronization) can increase considerably as the number of threads increases. Fig. 9 shows the overheads of OpenMP parallel region (left) and OpenMP barrier (right) measured with the OpenMP Micro-Benchmark [15] on the four platforms we benchmarked. The OpenMP overheads vary with respect to the platform, and increase with respect to the number of threads. Another unfavorable factor is that multiple threads within a node compete for the available memory bandwidth. When a large number of threads are deployed within the node these overheads and the memory bandwidth contention can potentially overwhelm the aforementioned benefits. This is why some systems do not exhibit superior performance of the hybrid model relative to pure MPI for the same number of processors. In future systems a factor of 5 to 10 in bandwidth increase is expected. This, together with new OpenMP libraries with low overheads, would influence favorably the hybrid model developed here.

Acknowledgements

The authors would like to thank Dr. Constantinos Evangelinos at MIT for his help on the thread-safety of the Veclib and many useful discussions. The assistance

from George Loriot and Samuel Fulcomer at the Technology Center for Advanced Scientific Computing and Visualization (TCASCV) at Brown University, from Mark Straka at NCSA, and from David O'Neal and Sergiu Sanielevici at PSC is gratefully acknowledged. Computer time was provided by NCSA, NPACI and PSC via an NSF NRAC grant.

References

[1] S.W. Bova, C. Breshears, C. Cuicchi, Z. Demirbilek, H.A. Gabb, Dual-level parallel analysis of harbor wave response using MPI and OpenMP, Int. J. High Perform Comput. Appl. 14 (2000) 49–64.
[2] F. Cappello, D. Etiemble, MPI versus MPI + OpenMP on the IBM SP for the NAS Benchmarks, in: Supercomputing 2000: High Performance Networking and Computing (SC2000), 2000.
[3] C.H. Crawford, C. Evangelinos, D. Newman, G.E. Karniadakis, Parallel benchmarks of turbulence in complex geometries, Comput. Fluids 25 (1996) 677–698.
[4] C. Evangelinos, G.E. Karniadakis, Communication patterns and models in Prism: A spectral element-Fourier parallel Navier–Stokes solver, in: Supercomputing 1996: High Performance Networking and Computing (SC96), 1996.
[5] W.D. Gropp, D.K. Kaushik, D.E. Keyes, B.F. Smith, High-performance parallel implicit CFD, Parallel Comput. 27 (2001) 337–362.
[6] D.S. Henty, Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling, in: Supercomputing 2000: High Performance Networking and Computing (SC2000), 2000.
[7] G.S. Karamanos, C. Evangelinos, R.C. Boes, M. Kirby, G.E. Karniadakis, Direct numerical simulation of turbulence with a PC/Linux cluster: fact or fiction?, in: Supercomputing 1999: High Performance Networking and Computing (SC99), 1999.
[8] G.E. Karniadakis, S.J. Sherwin, Spectral/hp Element Method for CFD, Oxford University Press, 1999.
[9] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitionaing irregular graphs, SIAM J. Sci. Comput. 20 (1998) 359–392.
[10] R.D. Loft, S.J. Thomas, J.M. Dennis, Terascale spectral element dynamical core for atmospheric general circulation models, Supercomputing 2001: High Performance Networking and Computing (SC2001), 2001.
[11] P. Luong, C.P. Breshears, L.N. Ly, Coastal ocean modeling of the U.S. west coast with multiblock grid and dual-level parallelism, in: Supercomputing 2001: High Performance Networking and Computing (SC2001), 2001.
[12] D.S. Nikolopoulos, E. Ayguade, Scaling irregular parallel codes with minimal programming effort, in: Supercomputing 2001: High Performance Networking and Computing (SC2001), 2001.
[13] H. Shan, J.P. Singh, L. Oliker, R. Biswas, A comparison of three programming models for adaptive applications on the Origin2000, in: Supercomputing 2000: High Performance Networking and Computing (SC2000), 2000.
[14] D. Xiu, G.E. Karniadakis, The Wiener–Askey polynomial chaos for stochastic differential equations, SIAM J. Sci. Comput. 24 (2002) 619–644.
[15] J.M. Bull, Measuring synchronization and scheduling overheads in OpenMP, in: First European Workshop on OpenMP (EWOMP'99), Lund, Sweden, 1999.