Parallel Computing 37 (2011) 316–326

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence

Pablo D. Mininni [a,c], Duane Rosenberg [a,*], Raghu Reddy [b], Annick Pouquet [a]

[a] Institute for Mathematics Applied to Geosciences, National Center for Atmospheric Research, P.O. Box 3000, Boulder, CO 80307-3000, USA
[b] Pittsburgh Supercomputing Center, 300 S. Craig Street, Pittsburgh, PA 15213, USA
[c] Departamento de Física, Facultad de Ciencias Exactas y Naturales & IFIBA, CONICET, Ciudad Universitaria, 1428 Buenos Aires, Argentina

ARTICLE INFO

Article history:
Received 23 March 2010
Received in revised form 20 March 2011
Accepted 18 May 2011
Available online 25 May 2011

Keywords:
Computational fluids
Numerical simulation
MPI
OpenMP
Parallel scalability

ABSTRACT

A hybrid scheme that utilizes MPI for distributed memory parallelism and OpenMP for shared memory parallelism is presented. The work is motivated by the desire to achieve exceptionally high Reynolds numbers in pseudospectral computations of fluid turbulence on emerging petascale, high core-count, massively parallel processing systems. The hybrid implementation derives from and augments a well-tested scalable MPI-parallelized pseudospectral code. The hybrid paradigm leads to a new picture for the domain decomposition of the pseudospectral grids, which is helpful in understanding, among other things, the 3D transpose of the global data that is necessary for the parallel fast Fourier transforms that are the central component of the numerical discretizations. Details of the hybrid implementation are provided, and performance tests illustrate the utility of the method. It is shown that the hybrid scheme achieves good scalability up to ~20,000 compute cores with a maximum efficiency of 89%, and a mean of 79%. Data are presented that help guide the choice of the optimal number of MPI tasks and OpenMP threads in order to maximize code performance on two different platforms.

Published by Elsevier B.V.

1. Introduction

Fluid turbulence arises from interactions at all spatial and temporal scales, and is therefore the quintessential petascale application. The Reynolds number R_v, which measures the strength of the nonlinearity in turbulent fluid systems, determines the number of degrees of freedom (d.o.f.) required to resolve all spatial scales, which increases as $R_v^{9/4}$ (in the Kolmogorov framework [11,10] of homogeneous and isotropic turbulence). For geophysical flows, R_v is often greater than 10^8, suggesting the need to evolve the geo-fluid equations with greater than 10^{18} grid points, if completely accurate computations of turbulent geophysical flows are to be realized without resorting to modeling of unresolved scales. This approach to computing fluid flows in which all spatial and temporal scales are resolved is called *direct numerical simulation* (DNS). If the goal is to simulate geophysical flows accurately, such computations must be carried out at exascale resolutions, which are not currently feasible. But petascale resolutions are just now becoming available, that can accommodate resolutions of 10^{15} grid points, corresponding to $R_v \sim 10^7$, which still allows for sufficient scale separation to study physically relevant complex turbulent flows.

Pseudospectral methods provide a very useful tool to study the problem because of their computational efficiency and high order numerical convergence. Attention is often focused on a 2π-periodic box domain in order to study scale interaction as it allows the use of fast spectral transforms that have a computational complexity of $\sim N\log(N)$ instead of $\sim N^2$, where N is

* Corresponding author.
E-mail addresses: mininni@ucar.edu (P.D. Mininni), duaner@ucar.edu (D. Rosenberg), rreddy@psc.edu (R. Reddy), pouquet@ucar.edu (A. Pouquet).

0167-8191/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.parco.2011.05.004

the linear resolution. For studies of homogeneous and isotropic turbulence, this choice is entirely consistent because the domain preserves the underlying translational and rotational invariance of the physics. But the approach is useful as well for studies of anisotropic or inhomogeneous turbulence, which broadens its usefulness. On the periodic domain, the Fourier basis is optimal, and the pseudospectral discretization [1,7,8] is preeminent due to the effectiveness of the fast Fourier transform (FFT) in converting from physical to spectral space, and back again. The pseudospectral method [13] has thus been used extensively in studies of computational fluid dynamics (CFD) including turbulence, with references too numerous to cite. This method has the added advantage of capturing accurately the interaction of multiple scales with little or no numerical dissipation or dispersion. This is clearly an important property for the numerics if we wish to quantify small scale dissipative effects that arise in the context of nonlinear turbulent interactions.

Pseudospectral methods, however, require global spectral transforms, and, therefore, are hard to implement in distributed memory environments. This has been labeled a crucial limitation of the method until domain decomposition techniques arose that allowed computation of serial FFTs in different directions in space (local in memory) after performing transpositions. One of these methods is the 1D (slab) domain decomposition (see e.g., [2]), that enables multidimensional FFTs to be parallelized effectively using the Message Passing Interface (MPI). However, these methods are often limited in the number of compute cores that can be used, and generalizations to larger core counts using solely MPI are often expensive or hard to tune as transpositions require all-to-all communications. Also, multi-dimensional transforms of some non-Fourier basis, such as spherical harmonics, cannot be parallelized using this technique. In the present work, a hybrid (MPI–OpenMP) scheme is described that builds upon the existing domain decomposition scheme that has been shown to be effective for parallel scaling using MPI alone. We leverage this existing domain decomposition method in constructing a hybrid MPI–OpenMP model using loop-level OpenMP directives and multi-threaded FFTs. The implementation is intended to address several concerns. It addresses the multi-level architectures of emerging platforms, and it is also designed to be portable to a variety of systems, with the expectation that it will provide scalability and performance without detailed knowledge of network topology or cache structure.

The idea of such loop-level, or implicit, parallelization in concert with MPI is not new. To date, these have generally been attempted on small core count systems, and the pure MPI scheme is found to outperform the hybrid schemes. In the context of CFD applications, it was found that on core counts up to 256 the overall elapsed time (for a finite element solver) was better for the pure MPI scheme than for the hybrid, even though the hybrid approach showed improved communication times in some cases [17]. A hybrid approach was taken in an implementation of a parallel 3D FFT algorithm [15] that succeeded in reducing the number of cache misses in the algorithm on an SMP system. But this approach was again tested only on a small core count platform, and considered the FFT algorithm alone, without the full fluid solver. To the best of our knowledge, the scheme described herein is the first published implementation of a hybrid model in a pseudospectral CFD context that has been attempted on high core count systems, and found to scale well.

In the following sections we present a new hybrid implementation. We begin first with a description (Section 2) of the numerical method and the underlying domain decomposition scheme. In Section 3 the hybrid model is presented, and a new domain decomposition picture is offered for viewing the distribution of work on multicore nodes. We also discuss in this section the implementation of the loop-level parallelization. Benchmarks are provided in Section 4, where we also consider the overhead and performance of the OpenMP parallelization, and the scalability of the full hybrid formulation. Finally, in Section 5, we offer some concluding remarks on lessons learned and our expectations for future hybrid performance on petascale systems.

2. The pseudo-spectral method and the underlying domain decomposition

All of the work in this paper will be based on simulations of the incompressible Navier–Stokes equations:

$$\partial_t \mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u} = -\nabla p + \nu\nabla^2\mathbf{u}, \tag{1}$$

$$\nabla\cdot\mathbf{u} = 0, \tag{2}$$

where $\mathbf{u}$ is the velocity, the kinematic viscosity is ν, and the pressure p can be viewed as a Lagrange multiplier used to satisfy the incompressibility constraint (Eq. (2)). These equations are solved using a pseudo-spectral method [1,7,8,14], in which each component of $\mathbf{u}$ is represented as a truncated (Galerkin) expansion in terms of the Fourier basis, and the nonlinear term is computed in physical space and then transformed using the fast Fourier transform (FFT), to spectral space. The nonlinear term plus pressure term are computed in such a way that the velocity is projected onto a divergence-free space, in order to satisfy (Eq. (2)). Details of this projection, and of the dealiasing required by the action of the nonlinear term, are not central to the discussion and can be found elsewhere [1,14], as can additional details of the discretization and parallelization of the scheme using solely MPI [6].

The key piece of any pseudospectral method, particularly for parallel computing, is the multidimensional Fourier transform algorithm. An efficient parallel implementation of this algorithm is essential for attaining high Reynolds numbers in turbulent hydrodynamics simulations, which is of chief concern here. We focus on a 3D Fourier transform of a scalar (or vector component) field of size N^3, with N grid points in each coordinate direction of the 2π-periodic domain. The distribution of real space points can be viewed as a cubic array of N^3 real numbers. In the underlying domain decomposition each compute core receives a "slab" of size $N\times N\times M$ node points, where $M = N/N_{\mathrm{MPI}}$, and N_{MPI} is the number of MPI tasks (processes). This

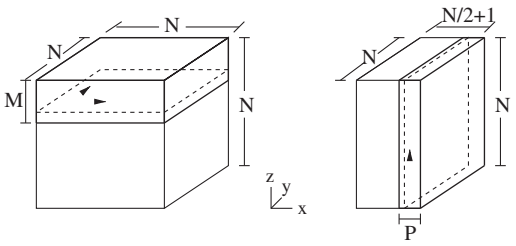

Fig. 1. Underlying 1D (slab) domain decomposition for pseudo-spectral method (*left*). Each compute core works on a slab of size $N \times N \times M$, where $M = N/N_{\mathrm{MPI}}$. The FFT is done by first doing the FFTs locally in each slab, in the directions specified by the arrows, yielding partially transformed data of size $(N/2+1) \times N \times M$. Then, an all-to-all communication is done to transpose the data globally (*right*), so that the remaining 1D FFT can be done in the direction specified by the arrow. The data for this step is stored in a cube of size $P \times N \times N$, with each core computing the FFT locally in a slab of size $P = (N/2+1)/N_{\mathrm{MPI}}$. Figure adapted from [6].

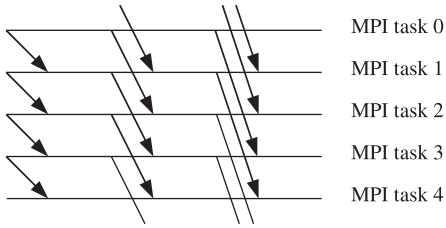

Fig. 2. Communication pattern for the all-to-all MPI communication to perform the transposition in the parallel FFT. Loops are executed in which point-to-point MPI communication (non-blocking send and receive) are performed with increasing stride between MPI tasks, until all communications are performed. In the hybrid case, each MPI task can spawn several threads, and the communication is handled by the master thread.

is referred to as a 1D domain decomposition because the distribution to MPI tasks occurs in one direction only; this decomposition is visualized in Fig. 1. Fourier transforms are performed locally in the direction of the arrows on the slab owned by an MPI task. The partially transformed (complex) data resides in a cube of size $(N/2+1) \times N \times M$. In visualizing the data cube, it appears that the array size has decreased. But this array is now complex, and its size results from the fact that a Fourier transform of real data $u(x)$ satisfies $\hat{u}(k) = -\hat{u}^*(-k)$ (where the asterisk denotes complex conjugate), and therefore only half the complex numbers need to be stored; the amount of data is still the same as in the original data cube, however. To compute the (complex) transform in the remaining direction, an all-to-all communication is carried out in order to transpose the global data cube, and decompose it into slices of size $P \times N \times N$, where $P = (N/2+1)/N_{\mathrm{MPI}}$. Non-blocking MPI communication is used for the all-to-all exchange. This communication allows the transform to be carried out in the remaining direction (seen on the right in Fig. 1) locally on each task. Besides using non-blocking calls, it is important to make the communication in an ordered way that ensures communication balance. In [2], a list of all possible pairs of MPI tasks is created to this end. Such a list may create problems for large core counts, and as a result here we implement the scheme shown in Fig. 2. Local FFTs are computed using the open source FFTW package [5,4].

The 1D domain decomposition has been shown in a production run to scale efficiently up to 2048 cores for resolutions up to 2048^3 [12], and, when properly implemented, minimizes the number of all-to-all communications that must be done to complete the transpose. However, it also limits the number of cores to the maximum number of MPI tasks that can be used, which is the linear resolution of the run, N. In practice, departures from linear scaling are observed before reaching N MPI tasks (e.g., when the thickness of a slab is one or two grid points, as in Fig. 6, **nthd** = 1 curve), as the ratio of computing to communication time decreases. We address these issues in Sections 3 and 4.

3. Implementation of the hybrid scheme

The growing tendency for petascale platforms is toward a hierarchical shared-memory node structure with each node having multiple sockets, each with increasing numbers of compute cores with shared or separate caches, and which may be encapsulated within a non-uniform memory access (NUMA) domain within the node. This hierarchical design seems especially suited to a multilevel domain decomposition scheme that can be optimized for the hierarchical hardware [9]. In order to address these emerging system designs, and to rectify the limitation in the underlying slab-only pseudo-spectral domain decomposition strategy of Section 2, which prevents scaling to core counts beyond the number of MPI tasks (with maximum equal to the linear resolution of the problem), we use OpenMP to further parallelize each MPI task. In this scheme, the MPI processes provide a coarse-grain parallelization using the slab domain decomposition described above, but OpenMP loop-level constructs and multi-threaded FFTs are applied within each MPI task to provide an inner level of parallelization.

Fig. 3 illustrates the two-level parallelization scheme. Each MPI task is parallelized by distributing work among a number of threads (*T*0...*T*3 in the figure), in possibly two different ways. This work distribution is provided by constructing parallel regions at the loop level using OpenMP directives. From the point of view of the outer level of parallelization, the multidimensional FFT discussed in Section 2 does not change. The MPI communication for the transpose takes place outside of the OpenMP parallel regions in *master only* mode of hybrid computing. To show specific inner-level parallelization and to present the origin of the two different ways to look at the decomposition, we provide here a code fragment showing the use of OpenMP directives in carrying out the transpose within a slab, crucial for computing the FFT. We focus on this particular algorithm because of its importance for the performance of the parallel FFT and also because it provides a good opportunity to highlight an important feature of the code:

```
!Multi-threaded FFTs are computed
!All-to-all MPI communication handled by the master thread
!Transposition is now done locally:
!$ omp parallel do if ((iend-ibeg)/csize.ge.nthd) private (jj,kk,i,j,k)
   DO ii = ibeg,iend,csize
!$omp parallel do if ((iend-ibeg)/csize.lt.nthd) private (kk,i,j,k)
     DO jj = 1,N,csize
       DO kk = 1,N,csize
         DO i = ii,min(iend,ii + csize-1)
         DO j = jj,min(N,jj + csize-1)
         DO k = kk,min(N,kk + csize-1)
           out (k,j,i) = cl(i,j,k)
         END DO
         END DO
         END DO
       END DO
     END DO
   END DO
```

Here, the indices **ibeg** and **iend** indicate the starting and stopping indices that define the slab for the initial domain decomposition of the data cube. The quantity **csize** refers to the cache-size, which is tunable. The outer loop is distributed among threads if the number of planes comprising the slab is greater than or equal to the number of threads, **nthd**, times the cache-size of each thread. The use of this directive suggests a decomposition scheme like that illustrated in Fig. 3(b). If the number of planes is less than **nthd**∗**csize**, then the inner loop is parallelized, which provides a domain decomposition scheme represented by Fig. 3(c). In this way, we minimize the effect of a potential load imbalance.

This example not only shows explicitly how loop-level parallelization is achieved, but also demonstrates one of the ways in which effective cache utilization is achieved in the local transposition of data by using a technique often referred to as "cache-blocking." The three outer loops ensure that the data handled by the inner loops is small enough to fit in cache. Since the cache size is tunable, this procedure for cache-optimization does not depend on whether the thread cache is shared or separate. It has been recognized [9] that the hybrid multi-level domain decomposition scheme may be especially valuable

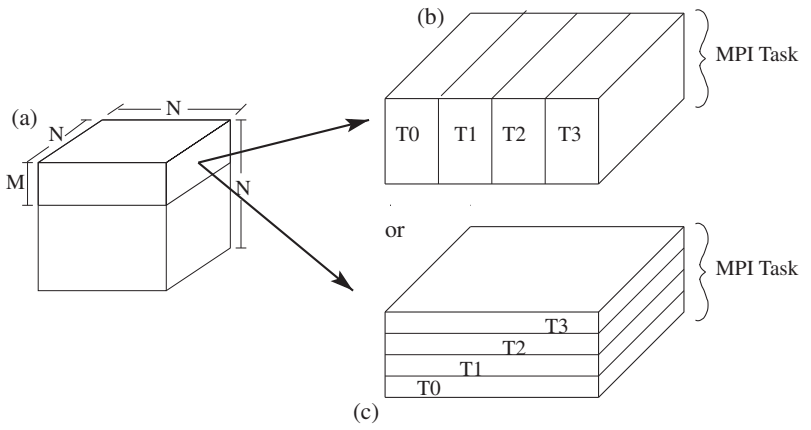

Fig. 3. Schematic of the new two-level domain decomposition strategy. (a) The 1D domain decomposition now acts as a coarse-grain MPI-based domain decomposition step. (b) and (c) A single slab (owned by a single MPI task) is further parallelized in one of two ways by loop-level OpenMP directives that distribute different "chunks" of the slab to different threads (here, labeled *T*0...*T*3) to be worked on, speeding up the MPI task. Multi-threaded FFTs are also used in each slab.

when taking cache optimization into account. All other loops in the code are modified with similar OpenMP directives, although most do not need to implement cache-blocking and the **csize** dependency. As a result, the remaining loops are parallelized as

```
!$omp parallel do if ((iend-ibeg).ge.nthd) private(j,k)
   DO i = ibeg,iend
!$omp parallel do if ((iend-ibeg).lt.nthd) private(k)
   DO j = 1,N
   DO k = 1,N
!Operations over arrays with indices ordered as A(k,j,i)
       END DO
     END DO
   END DO
```

The reason for this is that, unlike in the case of the transpose, most of the other loops load long lines of contiguous data into cache directly because they have no mixed-index dependencies; the transpose requires special treatment because of the dependence of a given block of memory on other non-contiguous blocks. Note that in all cases, the loops are ordered—like the above code fragment—so that the largest index range keeps the cache lines full. Only a few loops in the code (mostly associated with computation of global quantities or spectra which require reductions) have to be parallelized using the OpenMP ATOMIC directive.

In both examples, the choice of parallelizing the outer or middle loops based on workload per MPI task can be replaced by a COLLAPSE clause in OpenMP 3.0. This clause can be used to parallelize nested loops such as the ones shown above with only one OpenMP parallel directive. Both solutions have been benchmarked on different platforms and we observe similar timings. As a result, given the fact that the COLLAPSE clause is only available in compilers that support the new OpenMP standard, we use the approach described above to ensure portability of the code; it is also used in the following tests.

Besides the loop-level parallelism, the FFTs in each slab are also parallelized using the multi-threaded version of the FFTW libraries. MPI calls and I/O calls are only executed by the master thread in each MPI task. One of the additional benefits of the hybrid scheme presented here is that, by reducing the number of MPI tasks, we reduce not only the number of MPI calls, but we also spread out the MPI tasks more widely, improving network bandwidth, and we reduce the number of MPI buffers required to carry out communication. This also allows us to use parallel MPI I/O in environments with tens of thousands of cores, as the number of MPI tasks is a fraction of the total number of cores used. We will present cases where these considerations become significant in Section 4 where we provide performance results for the scheme.

4. Scalability and performance

A variety of tests have been performed to characterize the overhead, performance and scalability of the new hybrid domain-decomposition method. Tests were conducted primarily on two platforms: the *bluefire* system at the National Center for Atmospheric Research (NCAR), and the *kraken* system at the National Institute for Computational Sciences (NICS). The *bluefire* platform is an IBM Power 575 system, with 128 compute nodes, each of which contains 16 sockets with Power6 processors with 2 cores each. The compute nodes are interconnected with InfiniBand; each node has eight 4X InfiniBand double data rate (DDR) links. The *kraken* system is a Cray XT5 with 8256 compute nodes. Each compute node has two six-core AMD Opteron processors for a total of 99,072 cores. The compute nodes are interconnected with a 3D torus network (SeaStar). All of the tests discussed here operate in benchmark mode, for which no output other than timings are produced, and all solve Eqs. (1) and (2) for about 50 timesteps. Benchmark mode is used in order to remove the variability in times introduced by the I/O subsystem(s) operating with different throughput depending on problem size and user-defined output volume and frequency. However, we note that in production mode, we see increases in the run time of about 10% for typical output volume and frequency. Times are measured using the FORTRAN **cpu_time** routine, and the OpenMP routine **omp_get_wtime**. Timings presented below measure only the average time per timestep for the main time-advance loop; the initialization time (including the configuration of FFTW) is not included.

In the first series of tests, we consider the overhead and performance of OpenMP. The first test thus considers a single MPI task, and variable number of threads **nthd** with a fixed linear resolution of $N = 256$. The results are presented in Fig. 4. The performance for 1 and 2 threads is comparable for both platforms. After this, *bluefire* communicates out-of-socket, and its scaling decreases. As the core counts increase for this platform (e.g., as for the Power7 system) this drop-off may not be as severe. For *kraken*, there are 6 cores per socket and shared memory access becomes nonuniform over HyperTransport™ between the sockets, but we still see reasonably good scaling to about 7 threads, after which the scaling decreases markedly. Moreover, the departures from the ideal scaling observed in *kraken* while computing in-socket seem to be associated mainly with the hardware (e.g., nonuniform access, saturation of the memory bandwidth, or cache contention) and not specifically with OpenMP overhead; from Fig. 4 the curve giving the inverse timings (or unnormalized speedups) for the single thread,

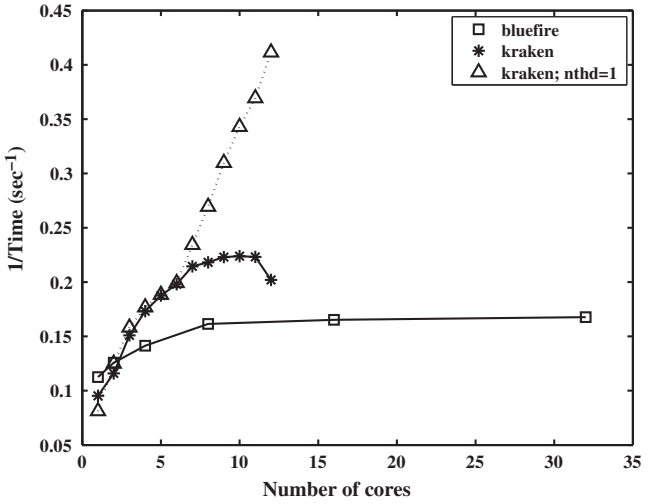

Fig. 4. Timing results with a single MPI task and multiple threads on two platforms. Both scale roughly the same for 1 and 2 threads, but afterward, the *bluefire* jobs run out-of-socket resulting in poor scaling. The *kraken* runs scale reasonably well up to 7 threads, even though there are only 6 cores per socket. The mean parallel efficiency up to 6 threads is 47%. The dotted line (with triangles) shows the speedups for a single thread, while varying the number of MPI tasks. This curve follows closely the variable thread curve (stars) in socket, with a mean parallel efficiency of 57% up to 6 tasks.

multiple MPI tasks (the triangles) follows closely the curve for the single MPI task, multiple threads (stars). For each of these curves, we can define the parallel efficiencies for a run with run time T on N_c cores as,

$$\epsilon = \frac{N_c T}{N_{c_0} T_0}, \tag{3}$$

with respect to a reference run on N_{c_0} cores with run time T_0. Indeed, taking as references the runs at 1 core for each curve, we compute mean efficiencies up to 6 cores of 57% and 47%, for the single thread multiple MPI tasks and the multiple threads single MPI task cases respectively, suggesting that, while there is an effect due to the OpenMP threads their performance is not drastically different from pure MPI in-socket.

In Fig. 5 we present a plot of the cache miss fraction for two sets of runs: those where we vary the number of MPI tasks, and those where we use a single MPI task, and vary the number of threads. The cache miss fraction is obtained by using the Integrated Performance Monitoring (IPM) infrastructure on *kraken*, and examining the Performance Application Programming Interface (PAPI) counters. As is readily seen when using pure MPI, the miss fraction is approximately constant, whereas when varying the threads, the fraction is approximately constant within a socket, after which the fraction begins a secular

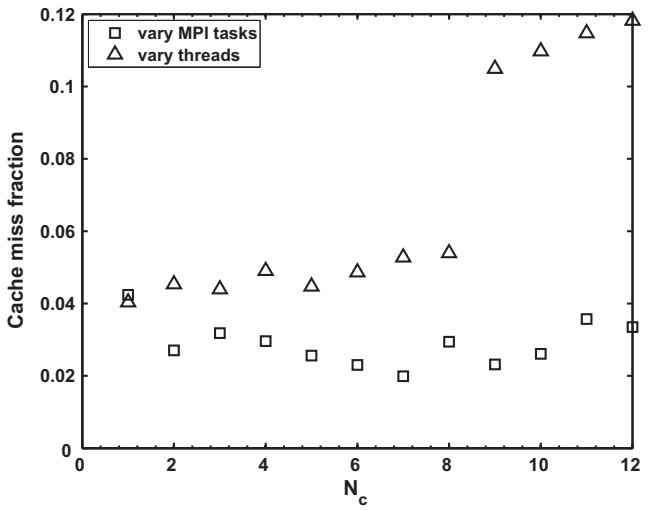

Fig. 5. Fraction of cache misses on *kraken* as core count increases. The squares represent the case where the number of MPI tasks is varied (pure MPI case), and the triangles, the case with a single MPI task (no communication), and variable number of threads.

increase. Interestingly, there is a large jump in the miss fraction most likely caused by cache contention at **nthd** = 8, which should at least be partially responsible for the turnover in the performance of the variable thread curve in Fig. 4 (see also below). We are uncertain exactly what the inherent connection is between this measured cache contention and the NUMA architecture. Moreover, it should be pointed out that PAPI and IPM were originally intented for single-core, MPI-only applications. Caution must be exercised when interpreting some counters in hybrid programs on multi-core platforms.

In order to examine effects of OpenMP overhead on *bluefire* results more closely, we compare two runs at different resolutions, one at $N = 256$, and one at $N = 512$ for a series of thread counts. These results are given in Fig. 6. In each plot the symbols refer to the same **nthd**, and the total number of compute cores, N_c, is varied by changing the number of MPI tasks. The MPI tasks were bound to cores (using "processor binding"), and symmetric multi-threading (SMT) was disabled. The first observation is that, for $N = 256$, the gains as **nthd** is increased (for any fixed number of MPI tasks) are roughly the same as the ones reported in Fig. 4 for only 1 MPI task. However, better improvements are observed for large numbers of MPI tasks; under this condition there are cases for which using **nthd** = 2 gives better timings than **nthd** = 1 using the same total number of cores (e.g., compare the triangle and the square at $N_c = 256$ on the left in Fig. 6). Increasing the number of threads further does not give substantial speed-ups. This is observed more clearly in the $N = 512$ runs. In this case, the slope of the series with **nthd** = 2 is greater than that for **nthd** = 1, indicating better gains with thread count (in-socket) as the size of the problem is increased.

Focusing on the 1- and 2-thread runs in Fig. 6, we see that the differences in speedup (or run time) in going from 1 to 2 threads *decreases* as the grid resolution increases. This suggests that there is a roughly fixed cost to the thread overhead whose relative magnitude can be reduced by increasing the work load. Then, as the thread count is increased to place them out-of-socket, extra costs, such as cache contention or memory bandwidth, appear (although for fixed number of threads, very good scaling is found with increasing number of MPI tasks). This trend can be further observed by considering runs at even larger grid resolution. Table 1 shows the efficiency, (Eq. (3)), with reference runs at $N_{c_0} = N/2$ and **nthd** = 1. The data shown refer to a case where each slab in the domain decomposition is one grid point thick, which, according the discussion at the end of Section 2, represents a worst-case scenario. Nevertheless, for a fixed number of cores $N_c = N$, the efficiency is best if two threads are used instead of one, and as resolution is increased efficiency improves. If $N_c = 2N$ and four threads are used, efficiency also increases but is at most ≈ 0.4 for $N = 1024$.

These results suggest that a hybrid approach may be most useful for large enough simulations in environments with large processor counts and when a large number of cores is available in the same socket. To verify this we consider the scaling to high core counts on *kraken*. For these runs, we set $N = 1536$, $N = 3072$, and $N = 4096$, with **nthd** = 6 or 12. At these resolutions, simulations with 1 thread cannot be executed as there is not enough memory per core in *kraken* to allocate the arrays. Several simulations at lower resolution ($N = 512$) were done to explore configuration parameters. This mainly involved NUMA options in the compiler (PGI), different binding configurations, MPI environment settings, and distribution of tasks among cores. We observed no substantial differences in the timings when changing the job distribution. Binding processors when **nthd** = 6 using the run command instead of NUMA options in the compiler was found to be best, although by a small margin ($\approx 5\%$). The implementation of MPI on *kraken* can also be configured to do fast copies in memory of the data when sending and receiving large messages. This gives a substantial speed up of the code (8–10%) but was found to require large amounts of memory that created problems at the highest resolutions. As a result, to compare on an equal footing, the runs described below were compiled with −O2 optimization, without using fast memory copy in MPI, and using the run command

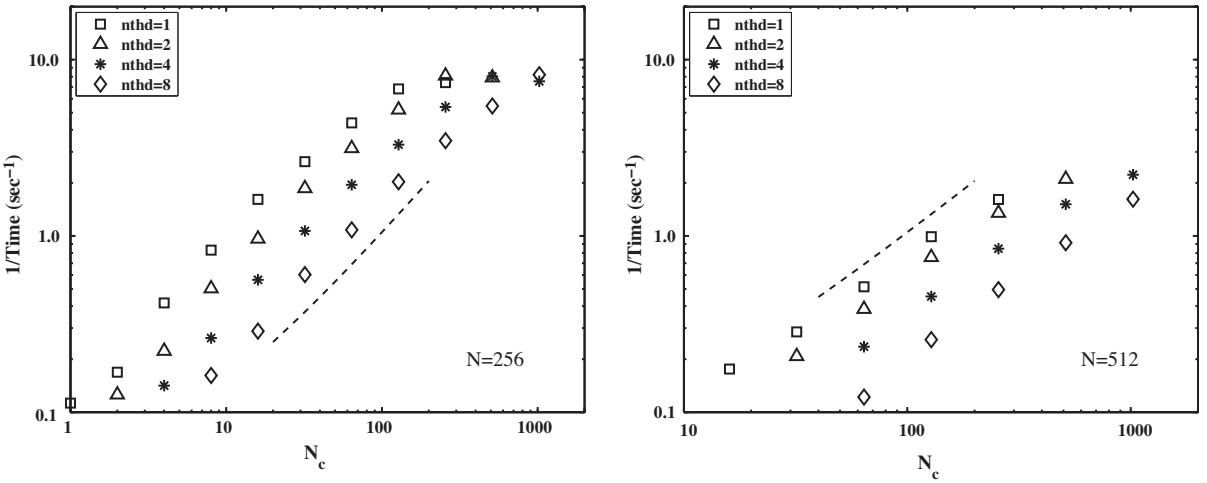

Fig. 6. Unnormalized speedup for two sets of *bluefire* runs, for $N = 256$ (*left*) and for $N = 512$ (*right*), as a function of the total number of compute cores. The curves represented by the different symbols are runs at a constant number of threads, as given in the legend. Note, in particular, that the difference in run time between the 1- and 2-thread surveys are smaller for the runs with $N = 512$, than for the cases where $N = 256$, which suggests that the thread overhead will manifest itself with smaller work load on this platform. Note also the almost linear speedup up to 1000 cores.

Table 1
Efficiency of runs with N linear resolution in *bluefire*, taking as reference runs with $N_{c_0} = N_c/2$ cores and **nthd** = 1 threads.

N	$N_{MPI} = N$, **nthd** = 1	$N_{MPI} = N$, **nthd** = 2
256	0.54	0.59
512	0.58	0.65
1024	0.63	0.66

```
aprun −n $NMPI −S 1 −d $OMP_NUM_THREADS executable
```

for the **nthd** = 6 runs, and the run command

```
aprun −n $NMPI −d $OMP_NUM_THREADS executable
```

for **nthd** = 12. In the former case, the −S 1 option tells aprun to bind one MPI process per socket, and NMPI refers to the total number of MPI processes. All cores in a given node in these runs are fully populated when **nthd** > 1. It should be noted that in the runs for which the most aggressive optimization options can be used (e.g., if enough memory is available), improvements in the times of up to 20% were found.

The results are shown graphically in Fig. 7 where we provide both a lin–lin (top) and a log–log (bottom) plot of unnormalized speedup vs. total core count, N_c. We see that good speedup is achieved up to ∼20,000 cores, but there are differences between the 6- and 12-thread cases. First note that the **nthd** = 12 in both problem sizes appears to maintain linear scaling, whereas the **nthd** = 6 does not. However, 6-thread cases have higher speedups than the 12-thread cases at lower core counts, and then saturate, becoming smaller than or equal to the **nthd** = 12 runs. The cross-over point occurs when the number of MPI tasks approaches the linear grid resolution at the same total core count, as noted at the end of Section 2 in the reference to Fig. 6. The higher speedup of the 6-thread runs at lower N_c is not surprising considering Fig. 4, since we see that when using a single MPI task for 12-threads, the threads beyond about 7 do not contribute significantly to speedup, probably due to cache contention (Figs. 4 and 5, and following discussion).

In Table 2 we provide the parallel efficiencies (Eq. (3)) for all runs at both problem resolutions. We see in general that for both problems sizes, the peak efficiencies can be reasonably high. The **nthd** = 12 runs have nearly the same mean efficiency, but from our data we cannot make this claim for the **nthd** = 6 runs. We compute that the speedup for the N = 1536 runs are higher for the **nthd** = 6 runs than for the **nthd** = 12 runs, until the number of MPI tasks reaches the linear grid resolution (fourth row in Table 2), at which time the former saturates.

We have examined the performance plateau of the **nthd** = 6 runs in somewhat more detail by instrumenting the code for use with IPM on *kraken*, which enables us to distinguish between computation and communication costs. We find that the largest component of communication costs, by far, are the MPI_Wait states, used for asynchronous send and receive (synchronization or barriers are a negligible contribution) during the all-to-all communication required for the data transposition in the parallel FFT. Hence, we provide in Table 3 the percent of total time accounted for by communication, and that accounted for, in particular, by the MPI_Wait state for a variety of the runs represented in Fig. 7. As we see from this data, the communication cost for all grid resolutions increases with the number of MPI tasks, N_{MPI}, and most of this cost is due to cores waiting for the asynchronous all-to-all communication to complete. The percentage of total communication at the largest N_{MPI} is insensitive to both the thread count and whether there are one or two MPI tasks per compute node (seen in the last three rows of the N = 1536 runs in Table 3), and is nearly constant. Comparison of rows 4 and 5 with row 1 also shows that the placement of the MPI tasks does not seem to be the determining factor in the communication cost. However, it is clear that the MPI_Wait time increases with the number of MPI tasks, as can be seen by comparing rows 2 and 4 for N = 1536 in Table 3, which show a remarkable jump in communication cost due to MPI_Wait conditions at the same total core count. A similar jump is seen in the N = 3072 runs in comparing rows 8 and 10 or rows 9 and 11, respectively, confirming at larger workload the increase in communication cost as the number of MPI tasks increases. Thus, the hybrid approach can decrease the communication time, and improve performance, by decreasing the number of MPI tasks that must communicate all-to-all, and, hence, increasing the effective network bandwidth for each MPI task. The increase in the MPI_Wait times in rows 1, 4, and 5 of Table 3 also makes explicit the likely reason for the departures from linear scaling in the pure MPI case, as stated at the end of Section 2, when $N_{MPI} = N$. We note, however, that there is no dramatic plateau in the N = 3072 curve of Fig. 7 for the six thread runs. This is likely due to the increased work load, such that the relative communication time is smaller, as can be seen, for example, by comparing rows 6 and 10 or rows 7 and 9 in the same table. But the increased communication cost with the number of MPI tasks can still be seen in the higher grid resolution case by comparing the slope of the unnormalized speedup curves (Fig. 7 bottom), for which the slope of the 6-thread curve is smaller than that for the 12-thread curve.

The amount of data transferred during any one send and receive in the all-to-all communication step decreases as N_{MPI} increases, as does the work load per MPI task (and a given **nthd**). But the plateau for the six thread, N = 1536 curve cannot be due entirely to network latency because the twelve thread curve continues to scale, and it has the same amount of work per MPI task. Thus, the reason for the increase in communication cost with number of MPI tasks per node may also be due to MPI resource contention between the two NUMA nodes (hex-core sockets) within that node.

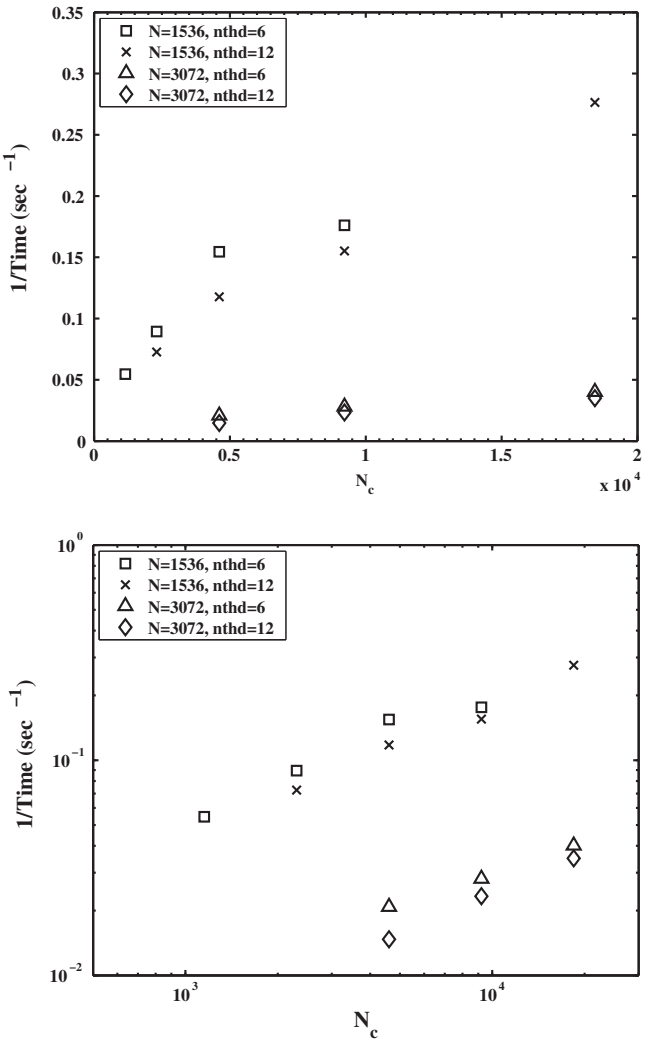

Fig. 7. Inverse run times for two sets of runs on *kraken*. The squares and crosses represent the timings for a $N = 1536$ run using 6 and 12 threads, respectively. The triangles and diamonds represent the same for a run size of $N = 3072$. We plot the data in two ways so as to be as explicit as possible regarding the results. *Top:* Lin–lin plot of inverse time vs. number total core count. *Bottom:* Log–log plot of inverse time vs. total core count. Note the tendency toward a cross-over in performance at about $N_c = 10{,}000$, where the **nthd** = 12 configuration outperforms the **nthd** = 6 configuration, similar to the behavior of the **nthd** = 2 for $N = 256$ in Fig. 6.

Table 2
Efficiency, ϵ, of *kraken* runs with $N = 1536$ and 3072 linear grid resolution, taking as reference runs with $N_{c_0} = N_c/2$ cores and same number of threads. Note that we cannot compute ϵ for the first run in each series. Mean efficiency is computed for a series using all computed ϵ for that series.

N	**nthd** = 6		**nthd** = 12	
	N_c	ϵ	N_c	ϵ
1536	1152	–	2304	–
	2304	0.82	4608	0.81
	4608	0.86	9216	0.66
	9216	0.57	18,432	0.89
Mean		0.75		0.79
3072	4608	–	4608	–
	9216	0.58	9216	0.80
	18,432	0.71	18,432	0.75
Mean		0.69		0.78

Table 3
Breakdown of the total communication cost of several runs for a problem size of N on *kraken* using IPM. "# nodes" is the number of 2-hex-core socket nodes used; "% comm" refers to the percentage of total run time comprising communication; and "% MPI_Wait" is the percentage of total run time comprising the MPI_Wait state. Note that the total core count is just the product of N_{MPI} and **nthd**.

N	N_{MPI}	**nthd**	# nodes	% comm	% MPI_Wait
1536	256	1	128	45	38
	256	6	128	57	49
	128	12	128	44	33
	1536	1	128	81	77
	1536	1	768	79	72
	1536	6	768	82	75
	1536	12	1536	76	64
3072	768	12	768	53	43
	1536	12	1536	64	52
	1536	6	768	70	60
	3072	6	1536	76	65

These results are consistent with the findings in *bluefire*, but the larger core count and cores-per-socket in *kraken* allow us to obtain significant additional gains using the hybrid approach on the latter system. We conclude that if the workload per MPI process becomes too small—in particular, if the N_{MPI} approaches the linear grid resolution—it is better to use more threads even if this places threads out-of-socket.

5. Discussion and conclusion

We have presented a hybrid MPI–OpenMP model for a pseudo-spectral CFD code. Beginning with an underlying "slab" domain decomposition adequate for parallelization by MPI, we have shown how the basic method is modified by loop-level parallelization to create a two-level parallelization scheme. The new level of parallelization can be thought of as modifying the underlying domain decomposition scheme, and we have pointed out precisely how this has been done depending on the size of the problem, number of threads, and number of MPI tasks.

The hybrid code has been tested primarily on two systems: the IBM Power6 system *bluefire* at NCAR, and the Cray XT5 system *kraken* at NICS. We have tested the thread overhead and performance, and found limitations of small socket core counts in *bluefire*, and observed cache contention in-node at the largest core counts on *kraken*. For low core counts, we have also discovered that there is a resolution threshold, N, below which the thread overhead manifests itself more clearly on *bluefire* and reduces scalability. In terms of large core counts, our results show good scalability up to about 20,000 cores on the *kraken* system. For large enough problems, we find the best scalability when the number of threads is 12 (one MPI process per compute node). On the other hand, we find that the performance time and speedup is better when **nthd** = 6, until the workload per MPI process is large enough, roughly that obtained by approaching the linear resolution of the grid, at which point, the performance time is better for the case where **nthd** = 12. This plateauing of the six-thread performance may be a result ultimately of resource contention by MPI. Nevertheless, for a given MPI/OpenMP configuration, and a given resolution, we find that the results are consistent from run-to-run, with little fluctuation in terms of scalability or run time.

The hybrid scheme presented allows us to scale to reasonably high core counts, and, perhaps most importantly, it allows the method to overcome the major limitation of using a slab-only decomposition pure-MPI scheme, namely, the inability to utilize a number of compute cores greater than the linear grid resolution of the problem. Furthermore, we have shown that the relative cost of communication increases with the number of MPI tasks, and that the hybrid scheme enables us to reduce this number. We would, therefore expect that for a given work load per MPI task, the hybrid scheme would clearly win out over the pure MPI version in terms of compute time. Unfortunately, while our results show that the hybrid scheme can be competitive, and cost nothing more than the pure MPI slab domain decomposition, we do not see a clear "win" for this scheme on the systems we have tested. We have seen that there is significant cache contention in the multithreaded case, particularly when the threads are placed out-of-socket. It may be that cache contention or memory bandwidth will nearly always prevent the hybrid scheme from outperforming the pure-MPI method as the number of cores per socket continues to grow.

Our experimentation has suggested a number of ways in which to improve the compute time of *kraken* runs. Perhaps the most important of these involve configuring the MPI environment. For the large message sizes we are using, setting the MPI environment variable MPICH_FAST_MEMCPY yields an 8–10% speedup over runs that do not use it, but it requires significantly more buffer memory. This increase in buffer requirements can prevent the code at large resolutions from fitting into memory, and must be considered carefully before attempting a production run. As an example, for $N = 4096$ using this configuration, we could only execute the code using 24,576 cores and 6 threads, and any other distribution using the same or smaller number of cores and changing the number of threads failed because of insufficient memory. We have not attempted larger resolution runs yet, but we note that the memory issue addressed here will become more of a concern the larger N becomes.

The hybrid scheme introduced here is not the only way in which to decompose the pseudo-spectral grid. An alternative is to retain a pure MPI model as in [16]. In this model, the domain decomposition takes the form of "pencils" which yield a 2D (N^2) distribution among MPI processes, and OpenMP is not required. This technique is also found to scale well [3] to large core counts, although severe fluctuations in performance are observed even within a given compute core-domain mapping. The pure MPI model does not suffer from effects of thread overhead (thread re-starts and synchronization) that we observe in smaller resolution runs, nor from potential problems with compiler optimizations that may arise when OpenMP is used [9]. Nevertheless, the hybrid method described here can be applied to non-Fourier basis spectral methods which may be impossible to parallelize with the 2D distribution (e.g., spherical harmonics). As pointed out in Section 3, our hybrid method offers a two-level parallelization method that may be more effective in mapping the domain to the hierarchical architectures that are now emerging, and better suited for environments with multiple cores per socket. The hybrid scheme may also aid in the MPI memory problems mentioned above, in that fewer MPI processes require less buffer memory. We intend in the future to continue testing this method to higher resolution as accessibility to a larger number of compute cores becomes more readily available. Since the code described here integrates the Navier–Stokes or magnetohydrodynamics equations when coupling to a magnetic field, including rotation, the hybrid scheme we have developed will prove useful in a variety of geophysical and astrophysical phenomena. And finally, we note that this approach works well even if the aspect ratio of the computational domain is not equal to unity.

Acknowledgements

Computer time was provided by NSF under sponsorship of the National Center for Atmospheric Research, and under TeraGrid (Project No. ASC090050) and is gratefully acknowledged. We would also like to acknowledge the TeraGrid ASTA support for this work.

References

[1] C. Canuto, M.Y. Hussaini, A. Quateroni, T.A. Zang, Spectral Methods in Fluid Dynamics, Springer, New York, 1988.
[2] P. Dmitruk, L.-P. Wang, W.H. Matthaeus, R. Zhang, D. Seckel, Scalable parallel FFT for simulations on a Beowulf cluster, Parallel Comput. 27 (2001) 1921–1936.
[3] D.A. Donzis, P.K. Yeung, D. Pekurovksy, Turbulence simulations at o(10^4) core counts, in: TeraGrid G08 Conference, Las Vegas, NV, 2008 (Science track paper).
[4] M. Frigo, S.G. Johnson, The design and implementation of FFTW3, in: Proceedings of the IEEE International Conference Acoustics Speech and Signal Processing, vol. 3, p. 1381, 1998.
[5] M. Frigo, S.G. Johnson, The design and implementation of FFTW3, Proceedings of the IEEE, vol. 93, Academic Press Inc., New York, 2005, pp. 216–231.
[6] D.O. Go'mez, P.D. Mininni, P. Dmitruk, Parallel simulations in turbulent MHD, Phys. Scripta T116 (2005) 123–127.
[7] D. Gottlieb, M.Y. Hussaini, S.A. Orszag, Spectral Methods for Partial Differential Equations, SIAM, Philadelphia, 1984.
[8] D. Gottlieb, S.A. Orszag, Numerical Analysis of Spectral Methods: Theory and Application, SIAM, Philadelphia, 1977.
[9] G. Hager, G. Jost, R. Rabenseifner, Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: Cray User Group Proceedings, 2009. <http://www.cug.org/5-publications/proceedings_attendee_lists/CUG09CD/CUG2009/pages/1-program/final_program/20.tuesday.html>.
[10] A.N. Kolmogorov, Dissipation of energy in locally isotropic turbulence, Dokl. Akad. Nauk SSSR 32 (1941) 16–18.
[11] A.N. Kolmogorov, The local structure of turbulence in incompressible viscous fluid for very large Reynolds number, Dokl. Akad. Nauk SSSR 30 (1941) 9–13.
[12] P.D. Mininni, A. Alexakis, A. Pouquet, Nonlocal interactions in hydrodynamic turbulence at high Reynolds numbers: the slow emergence of scaling laws, Phys. Rev. E 77 (2008) 036306.
[13] S.A. Orszag, Comparison of pseudospectral and spectral approximation, Stud. Appl. Math. 51 (1972) 253–259.
[14] G. Patterson, S.A. Orszag, Spectral calculations of isotropic turbulence: efficient removal of aliasing interactions, Phys. Fluids 14 (1971) 2538–2541.
[15] D. Takahashi, A hybrid MPI/OpenMP implementation of a parallel 3-D FFT on SMP clusters, in: R. Wyrzykowski et al. (Eds.), Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science, vol. 3911, Springer, Berlin/Heidelberg, 2006, pp. 970–977.
[16] P.K. Yeung, D.A. Donzis, K.R. Sreenivasan, High Reynolds number simulation of turbulent mixing, Phys. Fluids 17 (2005) 081703.
[17] E. Yilmaz, R.U. Payli, H.U. Akay, A. Ecer, Hybrid parallelism for CFD simulations: combining MPI with OpenMP, in: I.H. Tuncer, U. Glcat, D.R. Emerson, K. Matsuno (Eds.), Parallel Computational Fluid Dynamics, Lecture Notes in Computational Science and Engineering, vol. 67, Springer, Berlin/Heidelberg, 2007, pp. 401–408.