Hybrid OpenMP-MPI Turbulent Boundary Layer Code Over 32k Cores

Juan Sillero[1,*], Guillem Borrell[1], Javier Jiménez[1], and Robert D. Moser[2]

[1] School of Aeronautics, Universidad Politécnica de Madrid, 280040 Madrid, Spain
[2] Department of Mechanical Engineering and Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, TX 78735, USA
sillero@torroja.dmt.upm.es

Abstract. A hybrid OpenMP-MPI code has been developed and optimized for Blue Gene/P in order to perform a direct numerical simulation of a zero-pressure-gradient turbulent boundary layer at high Reynolds numbers. OpenMP is becoming the standard application programming interface for shared memory platforms, offering simplicity and portability. For architectures with limiting memory as Blue Gene/P, the use of OpenMP is especially well suited. MPI communications overhead are also improved due to the decreasing number of processes involved. Two boundary layers are simultaneously run due to physical considerations, represented by two different MPI groups. Different node mappings layouts have been investigated reducing communication times in a factor of two. The present hybrid code shows approximately linear weak scaling up to 32k cores.

Keywords: OpenMP, MPI, data locality, blocking, node topology.

1 Introduction

Modern parallel programming paradigms are now often used in clusters, combining Message Passing Interface (MPI) paradigm [2] for across the nodes with Open Multi-Processing (OpenMP) [1] within the nodes, known as hybrid OpenMP-MPI. The use of a hybrid methodology has some important advantages with respect to the traditional use of MPI: it is easy to implement through the use of directives, has low latency, high bandwidth, fine granularity, implicit communications versus explicit communications at node level, etc.

Previous TBL codes by our group were developed using MPI [6]. This choice was justified because of the computer architecture and the relatively low number of cores used. Nevertheless, using tens of thousands of cores with only MPI may degrade the code scalability and thus, its performance. This is one of the reasons to modify the original TBL code to a new hybrid OpenMP-MPI. Despite that, the main reason to port the code is the available memory per core. In order to simulate smooth $Re_\theta \approx 6650$ and rough $Re_\theta \approx 4200$ TBLs, allocation

[*] Corresponding author.

time in Intrepid at Argonne National Laboratory (USA) and Jugene at Jülich Forschungszentrum (Germany) have been granted through an INCITE award and a PRACE project respectively. Both codes are similar, and, from now on, we will just describe the smooth-wall one. The available memory per core is in both cases 512 Mb, instead of 2 GB as is the case of Mare Nostrum (MN, Barcelona). The previous $Re_\theta \approx 2000$ TBL was run on MN facility under the RES (Red Española de Supercomputación) project. With this available memory and the current TBL problem size, the use of OpenMP has naturally arisen as the simpler solution to overcome this issue. With the usage of OpenMP, some of the extra communication overhead associated with the use of MPI within the node is avoided as well. Nevertheless, other problems such as locality, false sharing, data placement [4] can arise from its usage.

2 The Numerical Code

The boundary layer is simulated in a parallelepiped over a smooth wall, spatially periodic spanwise, but with nonperiodic inflow and outflow in the streamwise direction. The code uses a relatively classical fractional-step method [7,8] to solve the incompressible Navier-Stokes equations expressed in primitive variables, using spectral expansions in the spanwise direction, and compact finite differences in the other two. A three sub-step, semi-implicit low storage Runge-Kutta scheme is used to evolve the equations in time.

For the problem here considered, both spectral methods and compact finite differences are tightly coupled operations. Our code is constructed in such way that only single data lines, along one of the coordinate directions, have to be accessed globally. However, the three directions have to be treated in every substep.

The code uses single precision in the I/O operations and communications and double precision in the differentiation and interpolation operations where the implicit part of the compact finite differences can cause loss of significance.

Compared to other highly scalable DNS/LES codes like FrontTier, Nek5000 or PHASTA, this code is specifically designed an tuned for a single purpose: to solve a zero-pressure-gradient turbulent boundary layer over a flat plate.

2.1 Computational Setup

The simulation is split in two concatenated domains with different boundary conditions as showed in figure 1. The planes π_i and π'_i are given inflow boundary conditions, and outflow boundary conditions are assigned to π_e and π'_e. The boundary conditions in π_t and π'_t impose a zero pressure gradient on the domain. Finally, the spanwise direction is considered periodic. The mission of the first boundary layer (BL_1) is to provide accurate inflow boundary conditions to the second one (BL_2). The inflow of BL_1 is obtained from its own plane π_1 that is rescaled using a method based on the one proposed by Lund, Wu and Squires[5]. The physical length of BL_1 is chosen to be long enough to let the large scales recover from an unrealistic initial condition and, once this asymptotic state has been reached, the plane π_2 is used to give BL_2 its inflow boundary condition. As a consequence, a small portion of the BL_1 simulation is thrown away.

Given that the goal of BL_1 is to allow the large scales to reach their asymptotic state and, given that the smaller scales take much shorter to reach a similar condition, BL_1 is run at a coarser resolution than BL_2. This setup permits computing a single boundary layer with significantly less computational work.

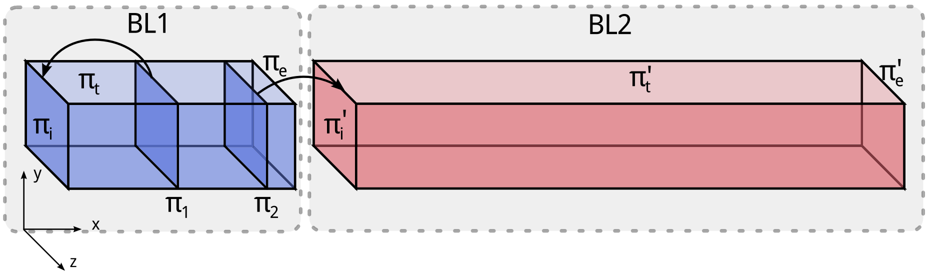

Fig. 1. Scheme of the computational domain and boundary conditions

Each of these two boundary layers is mapped to an MPI group. The first group runs the auxiliary simulation at coarse resolution and it consists of 512 nodes while the second MPI group comprises 7680 nodes and runs the main one in high resolution. The first MPI group is only about 8.5% of the total computational cost. This information is shown in table 1.

Table 1. Computational setup for the auxiliary BL_1 and main BL_2 boundary layers: N_t is the total number of degree of freedoms in giga points; Time/DoF is the amount of total CPU (core) time spent to compute a degree of freedom for every step

Case	Re_θ	Nodes	$N_x \times N_y \times N_z$	N_t (Gp)	Time/DoF
BL_1	1100-3000	512	$3585 \times 315 \times 2560$	2.89	13.98 μs
BL_2	2800-6650	7680	$15361 \times 535 \times 4096$	33.66	18.01 μs

MPI groups communicate each other only twice per sub-step by means of the MPI_COMM_WORLD communicator, while communications within each group occur via a local communicator defined at the beginning of the program. The first global operation is a SEND/RECEIVE of the π_2 plane, from BL_1 to BL_2. The second global operation is an MPI_ALL_REDUCE to set the time step for the temporal Runge-Kutta integrator, thus synchronizing both groups. The work done by each group must be balanced since each MPI group must wait for the other one in global operations, otherwise one group will slow down the second one that must remain idle during that time. The worst case scenario is when the auxiliary simulation slows down the main one. The time employed in communications for the auxiliary simulation has been improved using a customized node topology described in section 3.

2.2 Domain Decomposition

The parallelization distributes the simulation space over the different nodes, and to avoid global operations across nodes, it does a global transpose of the whole flow field twice every time sub-step (back and forth). The domain decomposition is sketched in figure 3 and can be classified as a *plane to pencil* domain decomposition. This strategy is motivated by the limited amount of memory in the Blue Gene/P nodes. Only transverse planes Π_{ZY} can fit in a node, and longitudinal planes Π_{XY} must be decomposed in X lines, i.e, pencils $\mathcal{P}_X$. According with the values presented in table 1, transverse planes are 25 Mb, longitudinal planes 94 Mb and pencils 120 Kbytes. Sixteen double precision buffers are need, and $3\Pi_{ZY}$ planes per node are used in the main simulation. Hence, the memory node occupation is close to 60%.

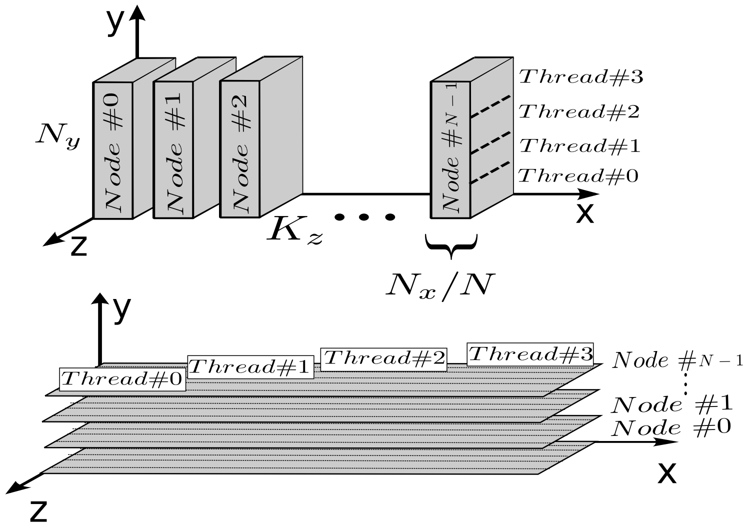

Fig. 2. Partition of the computational domain for OpenMP-MPI for $\mathcal{N}$ nodes and four threads. Top, Π_{ZY} planes; bottom, $\mathcal{P}_X$ pencil.

Each node contains $N_x/\mathcal{N}$ cross-flow planes, where N_x is the number of grid points in streamwise direction and $\mathcal{N}$ the total number of nodes. Each node is an MPI process, and OpenMP is applied within the node, splitting the sub-domain in a number of pieces equal to the available number of threads, four in Blue Gene/P.

The variables are allocated in memory as $\psi(K_z, N_y, N_x/\mathcal{N})$, where K_z is the number of modes in spanwise direction $(2/3N_z)$ and N_y the number of Y grid points. Each thread works in the same memory region of the shared variables using a first-touch data placement policy [3], maximizing data locality and diminishing cache missed [4], thus improving performance. The most common configuration that a team of threads can find is presented in figure 3, in which each thread works over a portion of N_y with static scheduling. This scheduling is defined manually through *thread_private* indexes, which maximizes memory reuse. In that way, each thread always works in the same portion of the array. Nevertheless, when loop dependencies in Y direction are found (i.e, LU decomposition) threads work over portions of K_z. For such loops, blocking techniques are used, putting the innermost loop index to the outermost part, thus maximizing data locality since strips of the arrays fit into the cache at the same time that threads can efficiently share the work load. The block size has been tuned for Blue Gene/P architecture comparing the performance of several runs.

In this configuration, operations in Y and Z are then performed. For operations in X direction global transposes are used to change variables memory layout to $\psi(N_x, K_z N_y/\mathcal{N})$. Now, each node contains a number of $K_z N_y/\mathcal{N}$ pencils. Each OpenMP thread works over a packet of $(K_z N_y/\mathcal{N})/N_{thread}$, where N_{thread} is the total number of threads. As in the previous configuration, workload is statically distributed among threads using *thread_private* indexes.

2.3 Global Transposes and Collective Communications

Roughly 45% of the overall execution time is spent transposing the variables from planes to pencils and back, therefore it was mandatory to optimize the global transpose as much as possible. Preliminary tests revealed that the most suitable communication strategy was to use the alltoallv routine and the BG/P torus network, twice as fast than our previous custom transpose routine based on point to point communication over the same network.

The global transpose is split into three sub-steps. The first one changes the alignment of the buffer containing a variable and casts the data from double to single precision to reduce the amount of data to be communicated. If more than one Π_{ZY} plane is stored in every node then, the buffer comprises the portion of contiguous data belonging to that node in order to keep message sizes as big as possible.

The second sub-step is a call to the MPI_ALLTOALLV routine. It was decided not to use MPI derived types because the transpose operations that change the data alignment and the double to float casting are parallelized with OpenMP.

The third and last sub-step transpose the resulting buffer aligning the data $\mathcal{P}_X$-wise. This last transpose has been optimized using a blocking strategy because the array to be transposed has many times more rows than columns. The whole array is split into smaller and squarer arrays that are transposed separately. The aspect ratio of those smaller arrays is optimized for cache performance using collected data from a series of tests. Finally the data is cast to double precision again.

The procedure to transpose from $\mathcal{P}_X$ pencils to Π_{ZY} planes is similar and is split in three sub-steps too.

3 Blue Gene/P Mapping

Mapping virtual processes onto physical processors is one of the essential issues in parallel computing, being a field of intense study in the last decade. Proper mapping is critical to achieve sustainable and scalable performance in modern supercomputing systems.

Blue Gene/P has a torus network topology, except for allocations smaller than 512 nodes, in which the torus degenerates to a mesh. Therefore, each node is connected to six nodes by a direct link. The location of a node within the torus can be described by three coordinates $[X, Y, Z]$.

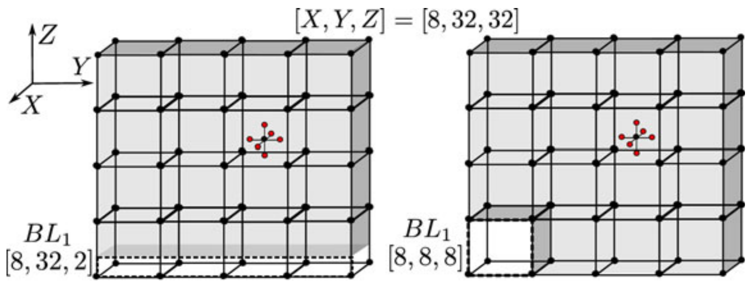

Fig. 3. Predefined (left) and custom (right) node mapping for a 8192 node partition in a $[8, 32, 32]$ topology. The predefined mapping assigns to BL_1 the nodes in a $[8, 32, 2]$ sub-domain. Custom mapping assigns the nodes to a $[8, 8, 8]$ sub-domain. BL_2 is mapped to the rest of the domain till complete the partition.

Different physical layouts of MPI tasks onto physical processors are predefined depending of the number of nodes to be allocated. The predefined mapping for a 512 node partition is a $[8, 8, 8]$ topology, while for 8192 nodes it is $[8, 32, 32]$ as it is shown in figure 3. Users can specify their desired node topology by using the environment variable BG_MAPPING and specifying the topology in a text file.

Changing the node topology completely changes the graph embedding problem and the path in which the MPI message travels. This can increase or decrease the number of hops needed to connect one node to another, and as a result, alter the communication time to send a message. Fine tuning for specific problems can considerably improve the time spent in communications. Table 2 shows different mappings that have been evaluated for our specific problem size. The custom mapping reduces the communication time for BL_1 by a factor of two. The work load for BL_1 is projected using this new communication time while the load for BL_2 is fixed. Balance is achieved minimising the time in which BL_1 or BL_2 are idle in the global communications.

The choice of a user-defined mapping is motivated due to the particular distribution of nodes and MPI groups. The first boundary layer BL_1 runs in 512 MPI processes mapped onto the first 512 nodes, while BL_2 runs in 7680 MPI processes mapped on the nodes ranging form 513 to 8192. Note that at

Table 2. Time spent in communication during global transposes. Different node topologies are presented for 10 time steps and for each boundary layer. Times are given in seconds.

Topology	Nodes	Comm BL_1	Comm BL_2
Predefined $[8, 8, 8]$	512	27.77	—
Custom $[32, 32, 8]$	8192	79.59	86.09
Predefined $[32, 32, 8]$	8192	160.22	85.44

the moment the communicator is split such that $Comm_{BL_1} \cup Comm_{BL_2} = MPI\_COMM\_WORLD$, neither $Comm_{BL_1}$ nor $Comm_{BL_2}$ can be on a 3D torus network. The communications will drop down to a 2D mesh with sub-optimal performance. Therefore, the optimum topology for our particular problem would be the one in which the number of hops for each MPI group is minimum, since collective communications occur locally for each group. For a single 512 node partitions the optimum is the use of a $[8, 8, 8]$ topology, in which messages travel within a single communication switch. We have found the optimum mapping for BL_1 to be a $[8, 8, 8]$ sub-domain within the predefined $[8, 32, 32]$, as shown in the right side of figure 3. BL_2 is mapped to the remaining nodes using the predefined topology and no other mappings have been further investigated. Although a $[8, 8, 8]$ topology is used for BL_1 by analogy with the single 512 node partition, communication time is nevertheless greater. This is due to the sub-optimal performance of using a 2D mesh instead of a 3D torus network, as already discussed. Ultimately, the reason can be found in the new hardware connection, since the 512 nodes and 8192 nodes of the 3-Dimensional torus network are physically connected in a different way. This leads to the increase in the number of hops for BL_1 collective communications, since messages cannot travel within a single communication switch anymore.

The methodology to optimize communications for another size partitions would be similar to the one just described: mapping virtual processes to nodes that are physically as close as possible so the number of hops is minimized.

4 Scalability Results in Blue Gene/P

4.1 OpenMP Scalability

It is important to state that the reason to mix concurrency and parallelism was not driven by the need for more performance but because the small memory capacity of the Blue Gene/P node, which does not allow a physically-significant block of data to be allocated to each core.

Some tests were run in a 512 node configuration after porting the code to OpenMP. The results are shown in table 3. These samples suggest that almost no penalty is paid when the computations are parallelized with OpenMP. In addition, the problem size per node and the MPI message size can be increased by a factor of four while using all the node's resources.

Table 3. OpenMP scalability test performed on 512 nodes. Two efficiencies (η) given: one based on the computation time (*Comp. T*) and one based on the total time (*Total T.*). Times are given in seconds.

$N_{threads}$	*Comp. T*	η	*Total T.*	η
1	60.820	1	70.528	1
2	30.895	0.984	38.951	0.905
4	16.470	0.923	24.438	0.721

4.2 MPI Scalability

Extensive data about MPI scalability was collected during the test runs in a BG/P system. The most relevant cases are listed in the table 4.

Table 4. Data collected from the profiled test cases. Time/DoF is the amount of total CPU (core) time spent to compute a degree of freedom for every step; N_t is the size in GiB of a buffer of size $N_x \times N_y \times N_z$; Comm, Transp and Comp are the percentage of the communication, transpose and computation time respect to the total.

Nodes	$N_x \times N_y \times N_z$	N_t	Time/DoF	Comm.	Transp.	Comp.	Symbol
512	$1297 \times 331 \times 768$	0.33	10.6 μs	17.9%	8.29%	73.8%	▶
1024	$3457 \times 646 \times 1536$	3.43	17.6 μs	44.7%	7.52%	47.8%	◀
2048	$6145 \times 646 \times 1536$	6.10	17.4 μs	46.0%	5.31%	48.8%	▲
4096	$8193 \times 711 \times 1536$	8.94	17.6 μs	44.6%	5.23%	53.2%	▼
8192	$8193 \times 711 \times 2048$	11.93	19.4 μs	37.4%	8.30%	57.6%	◆
8192	$16385 \times 801 \times 4608$	60.47	19.3 μs	39.7%	8.41%	51.9%	■

All the simulations run show a linear weak scaling up to 8192 nodes (32768 cores). The same code is expected to scale further without modifications, although at this time, higher node partitions have been not tested.

Figure 4(b) shows that the communications time is typically 40% of the total run time, and that both computation and communications are scaling as expected. The global transpose implementation shows an excellent scalability in all the test cases as shown in figure 4(a). It is important to mention that in the BG/P supercomputer architecture, the linear scaling is kept even when the estimated message size is about 1 kB in size. All our previous implementations of the global transpose broke the scalability near the 3 kB estimated message size limit.

5 Parallel I/O

Intermediate stages of the simulation in the form of flow fields (velocities and pressure) are an important result and are saved even more often than what checkpointing would require. Another mandatory feature to maintain the scalability with a high node count is the support for parallel collective I/O operations

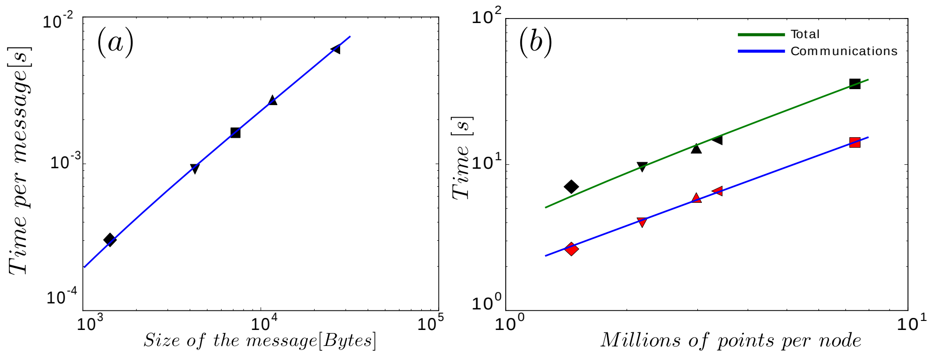

Fig. 4. Latency analysis (a) and scalability of the total and communication time for different test cases (b). Solid lines are linear regressions computed before taking logarithms of both axis.

when a parallel file system is available. A handful of alternatives have been tested, mainly upon GPFS, like raw posix calls enforcing the file system block size, sionlib (developed at JFZ) and parallel hdf5.

Hdf5 is a more convenient choice for storing and distributing scientific data than the alternatives tested because, despite having better performance, they require to translate the resulting files to a more useful format. Unfortunately sufficient performance could not be acheived without tuning the I/O process. Hdf5 performance depends on the availability of a cache in the file system. The observed behaviour in the BG/P systems was that writing was one, and sometimes two, orders of magnitude slower than reading because in the GPFS used the write cache was turned off. To overcome this issue, when the MPI I/O driver for hdf5 is used, the sieve buffer size parameter of hdf5 can be set to the file system block size. As a result, the write bandwidth for 8192 nodes was increased up to 16GiB/s, similar to the read bandwidth 22GiB/s and closer to the estimated maximum.

6 Conclusions

A hybrid OpenMP-MPI code has been developed from its original MPI version to perform direct numerical simulations of smooth and rough turbulent boundary layers at high Reynolds numbers. The code has been tested in a Blue Gene/P computer using up to 8192 nodes for MPI processes, and four threads per process for OpenMP, showing good scalability for both MPI and OpenMP. Two different domain decompositions are used to perform global operations in each of the 3-dimensional directions, employing collective communications to perform global transposes. Customized mappings of processes onto physical processors has been used for each of the two MPI groups, representing the auxiliary low resolution and the main high resolution simulation, speeding communications up by a factor of two.

Acknowledgments. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. Also funded by the Prace initiative. The work at the UPM was funded by CICYT under grant TRA2009-11498 and Consolider CSD2007-00050. J.A. Sillero was supported by an FPU fellowship from the UPM.

References

1. OpenMP Architecture Review Board OpenMP Specifications, `http://www.openmp.org`
2. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface, 2nd edn. MIT Press, Cambridge (1999)
3. Marchetti, M., Kontothanassis, L., Bianchini, R., Scott, M.: Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In: Proceeding of the 9th International Parallel Processing Symposium, Santa Barbara, CA, pp. 480–485 (April 1995)
4. Nikolopoulos, D.S., Papatheodorou, t.S., Polychronopoulos, c.D., Labarta, J., Ayguade, E.: Is Data Distribution Necessary in OpenMP? IEEE (2000)
5. Lund, T.S., Wu, X., Squires, K.D.: Generation or turbulent inflow data for spatially-developing boundary layer simulations. J. Comput. Phys. 140, 233–258 (1998)
6. Simens, M.P., Jiménez, J., Hoyas, S., Mizuno, Y.: A high-resolution code for turbulent boundary layers. J. Comput. Phys. 228, 4218–4231 (2009)
7. Kim, J., Moin, P.: Application of a fractional-step method to incompressible Navier-Stokes equations. J. Computat. Phys. 59, 308–323 (1985)
8. Perot, J.B.: An analysis of the fractional step method. J. Computat. Phys. 108, 51–58 (1993)