Hybrid Parallelization for BATS-R-US: Adding Efficient OpenMP to MPI Code

Hongyang Zhou[1]*

Abstract
We reviewed the edge-cutting finite volume MHD code Block Adaptive Tree Solarwind Roe Upwind Scheme (BATS-R-US) and managed to add coarse-grain OpenMP support to the previous pure MPI implementation, making BATS-R-US an efficient hybrid parallelization model. This paved the road for future OpenACC/Cuda development of a GPU support MHD model.

Keywords
OpenMP — MPI — Hybrid — MHD

[1]*Department of Climate and Space Sciences and Engineering, University of Michigan, Ann Arbor, MI*
***Corresponding author**: Hongyang Zhou, hyzhou@umich.edu

Contents

Introduction

The *Block Adaptive Tree Roe Upwind Scheme* (**BATS-R-US**) is an MHD code written in Fortran 90+ that has been developed at the University of Michigan for over 20 years and is still under active development. It is at the core of the Space Weather Modeling Framework (SWMF) and has been applied to simulate multi-scale space environment phenomena including, but not limited to, solar events (coronal mass ejections, solarwind transportation), planetary global magnetosphere, comets and outer heliosphere. For the purpose of *adaptive mesh refinement* (**AMR**) and running efficiency, the code was designed from the very beginning to use *adaptive block* structure in 3D for parallelization. In 2012, a new *Block Adaptive Tree Library* (**BATL**) is introduced for generalizing and separating the grid structure from the original source code.

However, BATS-R-US was orginally designed for pure MPI application and missed the rapid development of multi-threading technique starting from late 1990s. Even though MPI are generally observed to give better scalings than OpenMP, one obvious shortcoming of the current pure MPI implementation is memory usage: the tree structure (which will be explained in detail in the following review section) for storing the grid information as well as other precalculated variables are replicated on every MPI process, so memory leakage would happen if the number of MPI tasks used exceeds about 30000 or the number of blocks exceeds a few million on a 2 GB/core computing node. In order to scale to more cores in the current high performance clusters, we need the aid of multi-threading and shared memory. Adding OpenMP to BATS-R-US makes it a hybrid parallel code without lossing much efficiency, while in the meantime a maximum grid size of more than an order of magnitude larger than the orginal code is achieved.

1. Current Code Review

BATS-R-US was originally designed around a 3D block-adaptive grid, but it has been greatly extended in recent years. Each block has the same number of cells, but the blocks can have different sizes in physical space. The blocks can be split into eight children (refinement), or eight blocks can be merged into one (coarsening). Block adaptivity has a lot of advantages: each block is a simple structured grid allowing various numerical schemes, the fixed-size blocks are easy to load balance, the data corresponding to a block can easily fit into the cache, and the fixed length loops can be efficiently optimized (unrolled) by the compiler [**?**].

The block-adaptive grid algorithm has been improved and extended to handle arbitrary number of variables and generalized coordinates. Message passing over the poles of spherical and cylindrical grids, between cell faces edges and corners, etc., together with various algorithms, including block-to-block as well as single buffer communication, have been added to the code.

For application of block-adaptive grids with generalized coordinates in 1D, 2D and 3D, the new AMR library, i.e. *Block Adaptive Tree Library* (**BATL**) is introduced as an independent part of BATS-R-US [**?**]. It is written in Fortran 90 using the MPI library for communication, where pointers and derived types are intentionally avoided because MPI libraries available on supercomputers (at that time) lacked the capability of passing them between cores. Simple integer indexes are used instead of pointers, and arrays with named indexes instead of derived types.

The topology of the adaptive grid can be described by a set of *treenodes*. The nodes are identified by a global node index. All nodes have a bunch of information stored in an integer array `Tree_IA` for tracking and organizing grid block structure. The node information (18 integers per node in 3D) is replicated on all processors, which simplifies the algorithms substantially, and it allows a lot of functionality without inter-processor communication. As long as the total number of nodes does not exceed a few million, storage should not be an issue on most supercomputers with a gigabyte or more memory per core (note that "node" used here is equivalent to block in BATS-R-US; it is different from a computing node on clusters). However, this drawback triggers me into thinking about using shared memory for this grid library and also the whole BATS-R-US.

The grid blocks are surrounded by n_G layers of ghost cells to facilitate information exchange between the blocks. The value of the n_G constant is set before compilation for the sake of efficient optimization. Numerical experiments on supercomputers are needed for the optimal true/ghost cell ratio used in simulations.

BATL is intended to support arbitrary grid geometries, including Cartesian, spherical, cylindrical, toroidal, or even arbitrarily stretched grids. The whole computational domain is a brick in the generalized coordinates, but this can correspond to a spherical shell, a cylinder, or a distorted torus in real space. Similarly, every grid block is uniform in the generalized coordinates, but not necessarily in the Cartesian space. For the sake of efficiency useful geometrical information are precomputed : cell sizes (in generalized coordinates), face areas, face normals, cell volumes as well as the cartesian coordinates of the cell centers. This information is distributed over the processors, and it is indexed by the local block index.

For purposes of load balancing the active nodes, i.e. blocks, are ordered by a space filling curve. The root nodes are simply ordered in the usual array order by looping over them dimension by dimension. The trees starting from the root nodes are ordered recursively by looping over the children again dimension by dimension. This ordering corresponds to the Morton space filling curve. In the simplest case, load balancing means that we cut the space filling curve into N_p equal pieces, where N_p is the number of processors. BATL also supports a more complicated load balancing procedure, when there are different types of blocks, and each type is distributed evenly among the processors. For example the blocks on different AMR levels, or explicitly and implicitly advanced blocks can be load balanced separately. To do this the space filling curve is colored according to the block type, and then the curve of a given color is cut into N_p equal pieces. In general the number of blocks per type is not an integer multiple of N_p, but it is ensured that the number of blocks of a given type as well as the total number of blocks per processor varies by at most 1.

With all the above features included, the high level main code structure is shown in Listing.**??**. There are 94 modules with complicated dependencies and one main file. After `MPI_Init`, each process sets its own parameters , advances the solution using blocks and does the message pass every timestep through ghost cells. Being aware that most of the computation time is spent in `BATS_advance` (just like all other finite volume solvers for PDEs), we focus specifically on this part of the code for modification.

From the perspective of code design, BATS-R-US follows the principle of functional programming and avoids the usage of object oriented programming. Fortran modules are used heavily for storing global variables and categorising functions and subroutines. Currently there are 99 modules in the kernel source directory, 23 in the BATL directory, and 35 in the share library. Most key variables are declared as module public variables and are called/modified upon using the specific modules.

2. Hybrid Parallelization

One of the key features of BATS-R-US is scalability on supercomputers. Previous benchmark of pure MPI results have shown good strong scaling up to 8,192 cores and weak scaling up to 16,384 cores. However, pure MPI parallel structure and grid-related precalculated information for efficiency generate an unavoidable memory redundancy on computational nodes. To increase the scalability to even larger sizes, we need to re-organize the code and come up with a more advanced solution beyond MPI.

The idea of hybrid programming with OpenMP and MPI arises naturally with the architecture of modern supercomputers, where each node containing several multi-core chips are connected through the network (e.g. InfiniBand). Since MPI is originally designed for distributed memory (DM) while OpenMP is designed for shared memory (SM), the first idea of mixing them together is to use one MPI per node and one thread per node core.

There are several ways of communication supported by the current version of MPI library:

- Single: MPI only outside PARALLEL regions
- Funneled: MPI only on master thread
- Serialized: MPI calls on multiple threads serially
- Multiple: cocurrent MPI calls on multiple threads

Each of these has a specific MPI initialization indicator because of thread-safe support. For multiple-thread communication, the tags have to be used for distinguishing messages from different threads. In this project, I decided to set the support level to “funneled” to gain the benefits of OpenMP while minimizing the complexity of rewriting the source code. In hybrid programming, `MPI_Init` should be replaced by `MPI_Init_Thread` with the required and obtained support level as new additional arguments.

Basically two types of hybrid programming styles have been widely used: the fine-grained one, which uses `omp parallel do` on the innermost intensive loops, and the coarse-grained one, which uses OpenMP threads on a relatively higher level to replace MPI tasks. For in our implementation of adaptive mesh refinement, this is closely related to the choice of parallelization level on either blocks or cells? As explained in detail above, BATS-R-US spreads the blocks among MPI processes and then loop through cells inside each block for calculating the face values, face fluxes and source terms for update in each timestep. More specifically, there are a list of choices of numerical schemes (far beyond the original Roe Riemann solver) with up to fifth order accuracy, both in explicit and implicit timestepping. Simply speaking, explicit schemes loop over all the grid cells and update the solution one step forward from previous timestep, while implicit schemes solves linear equations iteratively until the system reach the tolerance level. We followed the coarse-grain parallelization idea and added OpenMP support to most if not all block-loops in both explicit and implicit schemes available. The key structure of the code is shown in List.**??**.

Note that:

1. we are unable to add multi-thread to the stage loop because of data backward dependency;
2. most notable input argument to the subroutines called inside block loop is the loop index `iBlock` (many other arguments are neglected for demonstration);
3. there are a lot of `if` statements in real research code with plenty of options;
4. MPI message passing is barried deep inside subroutine `exchange_messages` called from module `ModMessagePass`.

Finding this sweet spot for adding multi-threading parallelization is the key in hybrid code with simple yet efficient implentation.

In OpenMP, the most important characteristics of a variable is the range of scope. In such a large multi-module Fortran 90+ code as BATS-R-US, there are all kinds of arrays and scalars defined either globally or locally across all modules, making it nearly impossible to explicitly state each variable‘s scope attribute property at the beginning of OMP parallelized loops. Therefore, I have to rely on the default treatment of variable scopes. In principle, all the variables defined in modules are shared; all the variables defined in functions and subroutines called from modules are private. However, naively doing parallelization following the OpenMP defaults will definitely cause errors since there are actually temporary and local variables defined in modules that shouldn‘t be shared within different blocks. After some effort of searching and thinking, I came up with the solution of the “notorious” `threadprivate` option: any array with a block index must be `shared`, while those looping over cells and faces only must be `threadprivate`; scalars are generally `shared`, with a few exceptions if they are changing values inside the multi-threading parallelized region.

As a result of `threadprivate` usage, I need to reorganize some of the array initialization subroutines. Threadprivate variables need to be initialized for each single thread used, while global variables must only be initialized once.

I went through all the source codes (95 .f90 files in total) and applied the `threadprivate` modification to those variables related to the explicit schemes. Unfortunately there is nothing like a threadprivate region in OpenMP, so I had to explictly list all the variables with the `OMP threadprivate` constructs.

Further ideas of implementing OpenMP include adding multi-thread to message passing done in the BATL library and using nested OMP parallelized loops. The current version of message passing between blocks can be simply demonstrated in two categories:

1. if two spatial adjacent blocks are located on the same MPI process, make a direct copy from the physical cell values in one block into the ghost cells of the other block;
2. if two spatial adjacent blocks are located on different MPI process, unroll the quantities of physical cells, pass into a 1D array, do a MPI send/recv, and fill the ghost cell values on the receiver block in a known sequence.

A possible improvement to the current algorithm is to add multi-thread to asynchronous MPI isend/irecv: we can parallelize the local copy in category 1 and inter-MPI copy in category 2 with one multi-thread loop, and then use multiple threads for MPI send/recv, respectively. This may require the highest level of thread safety support “multiple”, with thread indexes used as tags. While still under progress, the idea is illustrated in pseudo code listed in List.**??**.

Algorithm 1 Multi-threading in MPI message pass

```
!$OMP parallel do
for iBlockSend=1:nBlock do
    if iProc == iProcRecv then
        local copy of physical values State_VGB from iBlockSend to iBlockRecv
    else
        copy to send buffer bufferS_I
!$OMP end parallel do
!$OMP parallel do
for iProcSend=0:nProc-1 do
    MPI_Irecv(bufferS_I, iProcSend, iThread)
!$OMP parallel do
for iProcRecv=0:nProc-1 do
    MPI_Irecv(bufferR_I, iProcRecv, iThread)
```

Besides, the nested loop idea will only be considered if the above ideas do not give satisfied scaling results. I can already imagine a mess in thread index when doing nested parallelization. Also up to this point, I intentionally skip the idea of dynamically allocated threads. More complicated things are considered only if simple ideas don‘t work. The motto is always ‘keep it simple and stupid’.

One possible shortcoming in terms of performance is the thread generation overhead everytime we go into the `advance_explicit` subroutine at each timestep. It may be better to move the start of parallelization region outside the subroutine and use `!$OMP single` for the serial parts. This idea will be tested in the future.

References can be found in [**?**], [**?**] and [**?**]. Many video tutorials can also be found on Youtube.

3. Implementation

There are several high level rules that we follow in improving the code:

1. For backward compatibility, the code should still work correctly and efficiently without OpenMP compilation flag.
2. Modification should be done as less as possible.

Because of the complex module structure of the code and the incomplete explicit control from

OpenMP derivatives, we have to rely on the default rules and settings about the variable scopes, especially for the function and subroutine calls within the multi-threaded region.

Current standard of Fortran variable scope default rules:

1. Variables explicitly showing up in the `omp` parallel section follow the common rules.
2. Module variables are shared.
3. Variables with save attribute are shared.
4. Variables being initialized are shared.
5. Otherwise variables are by default private.

There are several issues that I encountered during the process.

1. There are many places in the code, especially in the implicit scheme, where the structure of interdependences within loops exists and prevent direct OpenMP parallelization. However, in most cases the indexes of starting blocks can be calculated directly from the grid structure, which then allows a multi-threaded parallelization on the block level.
2. The OpenMP reduction operations are needed for multi-threading in global timesteps calculation and update checking. In the time accurate simulation, a minimum timestep under stability criterion is needed for advance in time, where finding a global minimum can be speed up with multi-thread reduction; in the update check session, the maximum relative change of pressure or density may be needed for reducing timestep, where again OpenMP reduction can be applied.
3. Using Fortran modules without the `ONLY` keyword inherits all the public variables and subroutines from the module, which may cause unexpected bugs in the program because of duplicate names. We avoid the unwanted behavior by always listing the public attributes and using `ONLY` keyword to import variables and functions as needed.
4. The timing library we used in BATS-R-US only works for single-threaded MPI runs. This new OpenMP extension of the code requires a modification of the current library. Now we have a new subroutine which only record the timings from the master thread for each MPI process.
5. Compilers play important roles in high performance computing. In practice, finding the optimal combination of platform, machine, compiler and compiler flags requires careful testing. I used gfortran (which is actually is wrapper of gcc) on my local machine, ftn (which is actually a wrapper of ifort) on Bluewaters, and ifort on Pleiades.
6. Thread affinity is critical in achieving good performance on large clusters, but the setup may not be trivial and requires special attention to different machines.

   For MPI/OpenMP hybrid codes built with SGI‘s MPT library, all the OpenMP threads for the same MPI process have the same process ID. In this case, setting the `MPI_DSM_DISTRIBUTE` environment variable to 1 (which is the default) causes all OpenMP threads to be pinned on the same core and the performance suffers. This default setting looks ridiculous to me, and I still don‘t understand the purpose of it.

   Following the recommended fix listed on NASA‘s website, I set `MPI_DSM_DISTRIBUTE` to 0 and use omplace for pinning instead in my job script. I also set `KMP_AFFINITY` to disabled so that Intel‘s thread affinity interface would not interfere with `omplace`. The essence part of the batch script is listed in Listing.**??**.

Listing 1. Sample batch script on Pleiades for hybrid runs

```
#PBS -lselect=2:ncpus=16:mpiprocs=8:
   ompthreads=2:model=sandy
#PBS -lwalltime=1:00:00

module load comp-intel mpi-sgi/mpt

setenv MPI_DSM_DISTRIBUTE 0
setenv KMP_AFFINITY disabled

cd $PBS_O_WORKDIR
```

```
mpiexec -np 16 omplace ./BATSRUS.exe
    > runlog
```

On Blue Waters, thread affinity for a certain version of Intel compiler was known to bind threads created by one MPI process accidentally if they are not using Intel CPUs. (It‘s actually AMD on Bluewaters). Intel seemed to fix this with the latest version of ifort.

Thread affinity is still kind of a mystery for me, and it takes time to figure out how to set it efficiently.

7. Load imbalance is always a concern when doing parallelization. OpenMP provides the programmer with several different tactics of scheduling. Due to the adaptive mesh used in BATS-R-US, the total number of allocated blocks may be larger than what is actually needed, and the sequence of blocks residing on one MPI process may be changed. For an overall good performance and avoid the small chunk size overhead, we use the default static schedule.

8. User module provides great extensibility to BATS-R-US's existing capability. It allows users to control almost every part of the code and add new features to the kernel. However, the hybrid parallelization structure poses additional requirements to the users when designing their own module: they need to know when and how to do MPI communication as well as avoid false sharing and race conditions for OpenMP. The rules listed in ??? serves as an guideline for future code extensions.

9. In many scenarios, using OpenMP may not be a good idea for performance. For instance, message pass between faces is used together with grid resolution changes to conserve face fluxes in our finite volume scheme. This usually includes too few blocks so it makes less sense to parallelize this part with OpenMP.

3.1 Improvements

3.1.1 Single fluid performance

The multi-fluid module is introduced into BATS-R-US several years ago. The extension from single fluid MHD to multi-fluid MHD causes performance penalty for single fluid runs because of unncessary string copies and function calls for setting up the state for different fluid species. This part is now being optimized to give a 40% improvement on single fluid simulations.

3.1.2 Message passing

The orginal message passing is done inside the BATL library for both cellwise and facewise message exchange between layers of ghost cells and neighboring physical cells. Message passing on the faces are implemented for the adaptive grid where the numerical fluxes need to be conserved between the coarse and fine cell faces on different resolution level. For all the current real applications, this part usually takes insignificant amount of time, so OpenMP derivatives are not added as the overhead may even be detrimental to performance. In the case of cellwise message passing, the old scheme combines the local exchange (message passing between blocks on the same processor) and the remote exchange (message passing between blocks on the remote processors). This turns out to be a tough case for adding multi-threading parallelization, which is now rewritten to fully separate the local and remote part. The local copies are multi-threaded while the remote exchange are done using 1D buffers and MPI iSend/iRecv. Currently this implementation only requires the thread safety level up to `MPI_THREAD_FUNNELED`, which means only the main thread can make MPI calls. If we observe a significant slowdown for the MPI message passing in any future application, higher thread levels with more complex communication scheme can be tested.

3.1.3 Preventing temporary arrays

Temporary arrays are created when we try to pass slices of a large array which are strided in memory into another function or subroutine. This potentially slows down the code. We avoid temporary arrays in most cases by allocating local arrays and pass as assumed shape array.

3.1.4 Parameters setup

Because of the integrity of all kinds of numerical methods for solving the face flux for the Riemann problem, each parameter is set individually without interferences with each other for the flexibility of adding new schemes. However, the old FaceFlux module sets all the parameters during each timestep, which is modified to be done only once in the initial timestep to avoid redundant computation.

4. Test and Verification

4.1 Platform Information

All the testing platforms are listed in Table ???.

The verification tests are done on MacbookPro 2015 with 2.2GHz qual-core Intel Core i7 using Nagfor and gfortran compilers.

Stampede2 SKX nodes with 192 GB of memory per node. Each node contains 48 Intel Xeon Platinum 8160 Skylake cores.

The large scale tests are done on Blue Waters Cray XE6 nodes with 64 GB of memory per node. Each node contains 16 Bulldozer Cores with 2 AMD 6276 Interlagos Processors. There are 32 integer scheduling units per node with 2 GB memory per unit.

4.2 Verification

The first series of tests were done locally with 2.2GHz qual-core Intel Core i7. These tests are the standard tests created for BATS-R-US for verification. Without any surprise, among all 46 tests designed for BATS-R-US (Table.**??**), when I compiled with OpenMP flags and used 2 threads for each MPI process, only 10 passed successfully (with a checkmark ✓), 6 showed differences in the outputs (with a question mark ?), and the rest caused runtime errors when running with more than 2 threads (shown with a cross X). This shows that:

1. The modified code works for purely explicit scheme with no AMR. Since I made no changes in the semi-implicit, point-implicit and fully implicit scheme modules, nearly all the related tests caused runtime error.
2. The pure explicit scheme only covers a small range of applications.

Nevertheless, this is already impressive to me considering the minor modification ratio of the code.

Then I proceeded to do performance tests locally. The standard shocktube test and two another simple density wave propagation tests are selected to estimate the timing on my local machine with 4-core 2.2 GHz Intel i7 processor. The results are shown in Fig.**??**. Note that this is total runtime, including the `set_parameter` section

Table 1. BATSRUS nightly tests

Test Name	Status
2bodyplot	✓
amr	?
anisotropic	✓
awsomfluids	X
chromo	✓
comet	✓
cometCGfluids	✓
cometfluids_restart	X
cometCGhd	✓
cometfluids	X
corona	✓
coronasph	✓
earthsph	✓
eosgodunov	X
fivemoment_alfven	✓
fivemoment_langmuir	✓
fivemoment_shock	✓
fluxemergence	X
func	?
ganymede	✓
graydiffusion	✓
hallmhd	✓
laserpackage	X
magnetometer	X
mars	✓
mars_restart	✓
marsfluids	✓
marsfluids_restart	✓
mercurysph	✓
mhdions	✓
mhdnoncons	✓
multifluid	✓
multiion	✓
outerhelio	✓
partsteady	✓
region2d	✓
saturn	✓
shockramp	✓
shocktube	✓
spectrum	X
titan	✓
titan_restart	✓
twofluidmhd	✓
venus	✓
venus_restart	✓
viscosity	✓

(pure MPI), `advance` section (MPI + OpenMP), and `save_output` section (pure MPI), so we would expect a slightly decrease in overall performance for cases where $nProc*nThread = constant$ (because within one MPI process, the reading and writing parts are "serial" in hybrid programming sense). In addition, the `threadprivate` declaration generates non-neglibile overhead to the multi-threaded region.

The third series of tests were done on Pleiades Sandy Bridge nodes (Fig.**??**). Each node has a dual-socket 16 core structure. In this test I use 512 blocks (8 in each dimension), with a cell size of $8*8*8$ and 2 levels of ghost cells in Cartesian coordinates. Theoretically, load balance of blocks among processors won't be an issue in this test. Ideally, for the series of tests with mixed set of number of processors and number of threads under a fixed product ($nProc*nThread =$ const.), we would expect a horizontal colored line if MPI and OpenMP provide perfect parallelizations. In real tests shown in Fig.**??**, I observed slightly decreasing efficiency as the number of threads generated per node increases from 1 to 8, followed with a surprising drop in performance when running with 16 threads (which is equal to the number of cores on Sandy Bridge) per node. To further diagnose the problem, I went to Ivy Bridge with 20 cores per node and found the similar trend: the performance vastly decreased when the number of threads goes beyond half number of cores per node. My hypothesis is that since each node on Sandy Bridge or Ivy Bridge actually consists of 2 NUMA nodes, the thread affinity setup may limit multi-threading to one NUMA node, such that if I allocate more threads, they just fight for the physical cores and memory which results in huge decrease in timing. I believe this can be solved with more detailed explicit thread affinity setup, but for now I just leave it for the future development plan.

The results of simple strong scaling multi-threaded test runs are shown in Fig.**??** (1 MPI + multi-threads). I can get pretty good scaling up to 8 threads with an efficieny $s = 0.86$.

Another important question is whether or not I observe improved memory usage. The real memory used in hybrid simulations with $nProc*nThread = 8$ is plotted in Fig.**??**. I do see a significant drop in memory usage from over 600 MB to 300 MB from pure MPI to pure OpenMP. This illustrates the importance of shared memory usage for simulations, especially when pure MPI implementation encounters memory issue. This encouraging result gives me confidence in the upcoming large scale tests. (Based on the discussion with Gabor, this information is in fact misleading. I reduction in memory usage is mostly from the decrease in MPI processes: there's a fixed number of blocks on each process, whether it is used or not. As long as the number of blocks is small, you cannot actually see the benefits of OpenMP!)

5. Conclusion and Future Work

We have successfully extended our finite volume MHD code BATS-R-US with adaptive mesh refinement in general coordinates from pure MPI to MPI+OpenMP implementation. While maintaining the nice scaling performance up to $\sim 10^6$ cores, we are now able to solve problems more than an order of magnitude larger than before thanks to the usage of shared memory. This also paves the way for porting the complex solver to accelerators and GPUs with minimum code modifications possible.

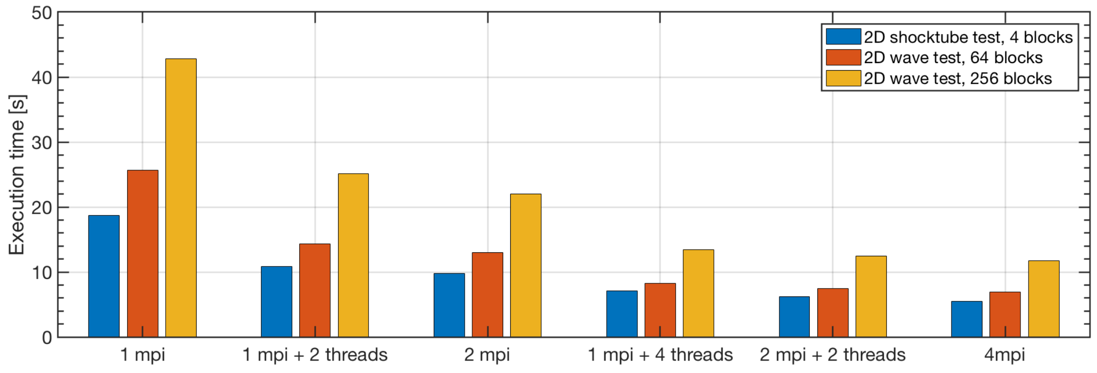

Figure 1. Timing of 3 simple tests on Macbook Pro with 4-core 2.2 GHz Intel i7 processor.

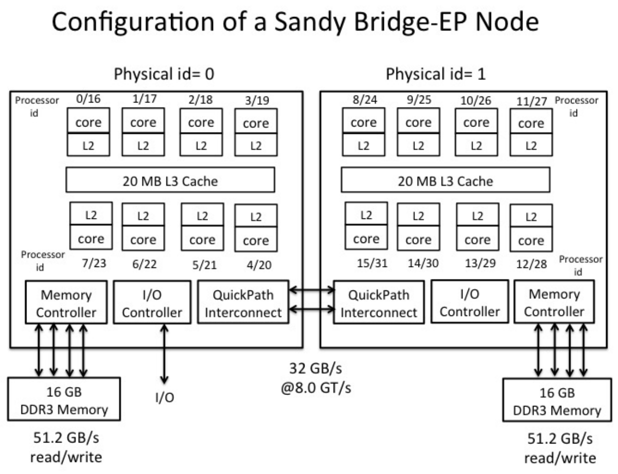

Figure 2. Configuration of testing node.

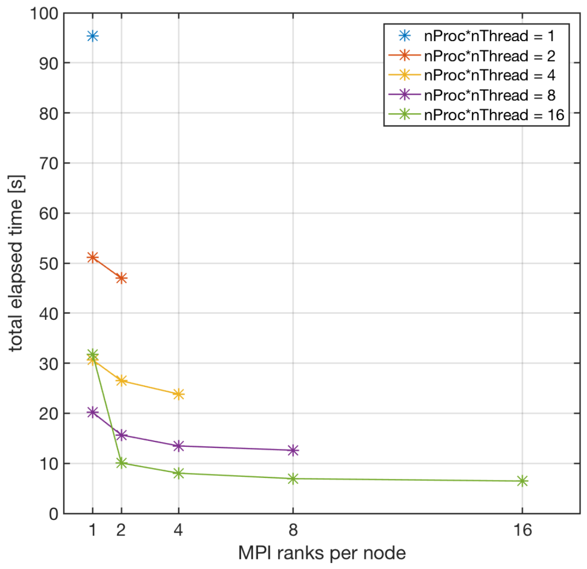

Figure 3. Performance on a single node on Pleiades Sandy Bridge tested with 3D wave propagation. The cell size is $8*8*8$ in Cartesian coordinates, with 2 levels of ghost cells in each dimension. The x axis represents the number of MPI processes per node, and y axis is the total elasped time. The same color represents series of runs with a fixed number of the product of *nProc* and *nThread*.

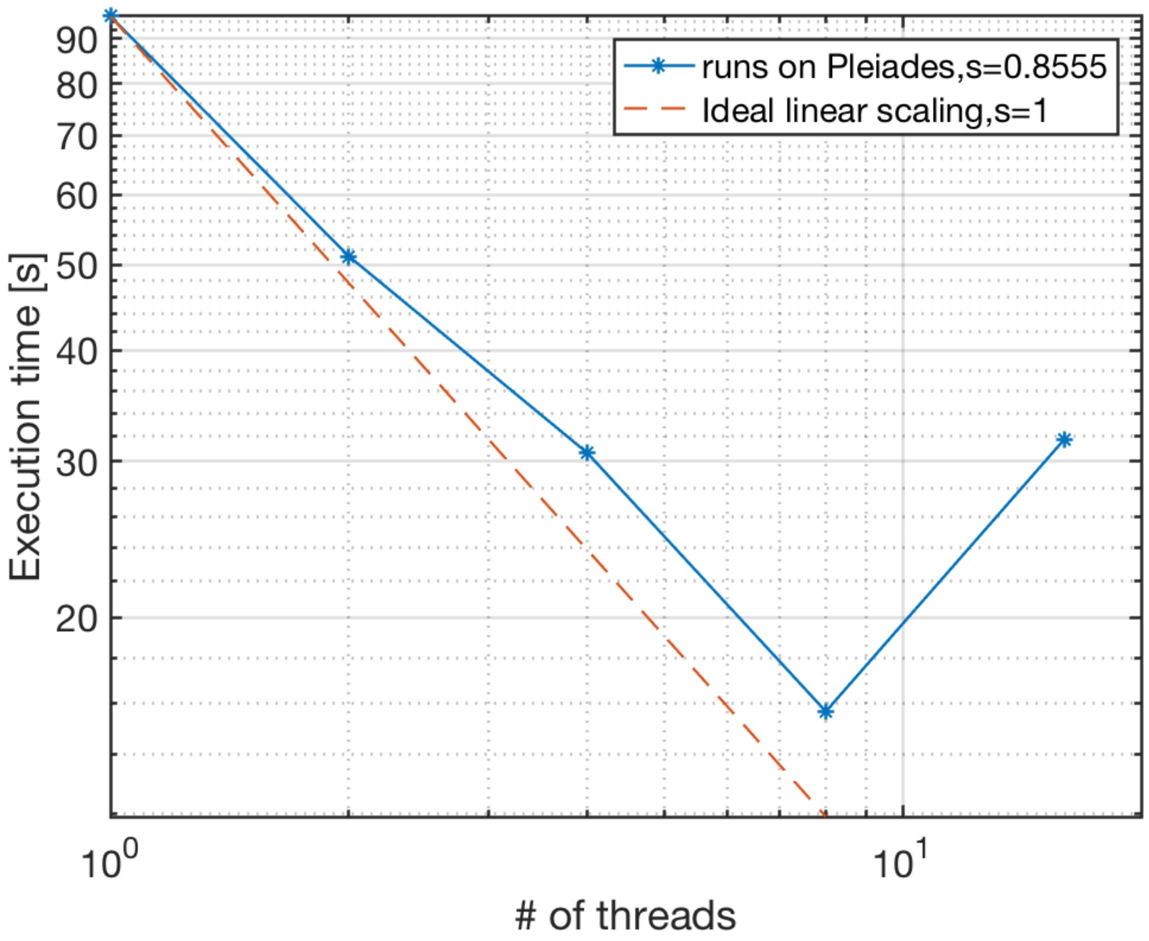

Figure 4. Scaling with the number of threads.

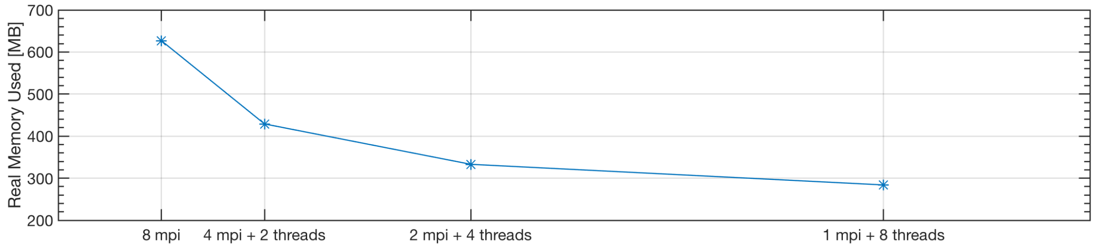

Figure 5. Memory usage for 3D wave tests monitored by system log file.