Running Large Scale Jobs on a Cray XE6 System

Yun (Helen) He and Katie Antypas
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, USA
e-mail: yhe@lbl.gov; kanytpas@lbl.gov

***Abstract*— Users face various challenges with running and scaling large scale jobs on peta-scale production systems. For example, certain applications may not have enough memory per core, the default environment variables may need to be adjusted, or I/O dominates run time. Using real application and benchmark examples, this paper will discuss some of the run time tuning options for running large scale pure MPI and hybrid MPI/OpenMP jobs successfully and efficiently on Hopper, the NERSC production XE6 system. These tuning options include MPI environment settings, OpenMP threads, process and memory affinity choices, and IO file striping settings.**

Keywords- running large jobs; XE6; MPI rank reordering; hybrid MPI/OpenMP; process and memory affinity; huge pages; IO file striping

I. INTRODUCTION

The peta-scale Cray XE6 system, Hopper [1], is the flagship production machine at the National Energy Research Scientific Computing (NERSC) that serves over 4,500 users in about 650 different projects across a broad range of science disciplines supported by the Department of Energy (DOE) Office of Science. Usually there are hundreds of users logged into the system at any time of the day.

Hopper has 6,384 nodes, and 153,126 cores. It has the peak performance of 1.28 PFlops, and sustained performance of ~140 Tflops. Total memory is 212 TB. Each compute node has 2 twelve-core AMD MagnyCours 2.1 GHz processors (2 sockets). On each socket, there are 2 dies (also called NUMA domains). There are 4 NUMA nodes per node, and 6 cores per NUMA node. Memory access to a remote NUMA domain is slower than within the local NUMA domain (Fig 1). Memory bandwidth ranges from 6.4 GB/sec to 21.3 GB/sec depending on the origin and destination of the NUMA nodes. Most of the compute nodes have 32 GB per node (1.33 GB/core) while some large memory nodes have 64 GB per node (2.67 GB/core). Hopper uses Lustre as its scratch parallel file system.

NERSC measures the percentage of hours that are devoted to jobs of different sizes to understand the workload on the system. Fig 2 shows the raw compute hours (in millions) by number of cores used on Hopper from January 2011 to March 2012. Applications on Hopper run at all different concurrencies. About 40% of total compute time is used by jobs over 8,192 cores. The percentage was even higher before charging started on May 1, 2011, since users are more ambitious in scaling when charging and allocation are not of concern. At the same time, many users take advantage of Hopper's large size to run thousands of moderately sized jobs.

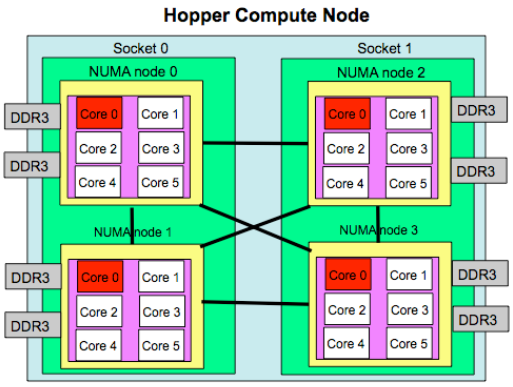


Figure 1. An illustration of a Hopper compute node. There are 4 NUMA nodes per node, and 6 cores per NUMA node. Memory access to a remote NUMA domain is slower than within the local NUMA domain.

When users attempt to scale their scientific applications to larger concurrencies, they may run into some challenges. For example, certain applications may not have enough memory per core, the default environment variables may need to be adjusted, performance is subject to NUMA effects, or I/O dominates run time.

We have conducted a survey of NERSC users who run large scale jobs routinely. The questions in the survey include: what are their initial obstacles and challenges, and what are the strategies they have used to ensure the successful running of the codes. The areas to explore include using fewer cores per node, trying different MPI environment settings, testing different MPI rank reordering methods, using hybrid MPI/OpenMP or different OpenMP scheduling, measuring NUMA effects, adjusting process and memory affinity options, huge pages, core specialization, and IO tunings, etc.

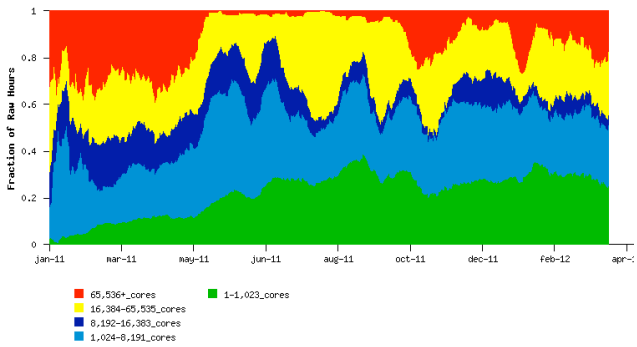


Figure 2. Raw compute hours (in millions) by number of cores used on Hopper from January 2011 to March 2012. About 40% of total compute time is used by jobs over 8k cores. The percentage was even higher before charging started on May 1, 2011.

The goal of this study is to learn about application tuning successes with real application examples, and to come up with a few practical tips and run time tuning options for running large jobs on Hopper [2,3], that can benefit general users to achieve optimal performance and scaling of their applications.

The rest of the paper is organized as follows: Section II contains general feedback from our user survey on running large jobs on Hopper. Section III gives tuning examples of real user applications and NERSC benchmarks. Section IV discusses two general system issues that have an impact on running jobs, especially large jobs. Section V discusses IO tunings. And Section VI provides a summary of practical tips for running large jobs on Hopper.

II. USER SURVEY RESULTS

We conducted a "Running Large Scale Jobs on Hopper" user survey from users who routinely run large jobs using 682+ nodes (16,368+ cores). Jobs over this size receive charging discount on Hopper. The survey was sent to about 50 users, and over 1/3 responded. Questions included in the survey and general feedback are:

1. What is the range of large job sizes you typically run? (number of cores, number of nodes)

The range is from single node to full machine. Some users conduct scaling studies, and some users debug their codes in small scale, and run production codes in large scale. Some codes are mature enough to run in capability mode using the whole machine.

2. What are the code names and the science areas?
Science areas cover compute science, engineering, chemistry, material sciences, astrophysics, nuclear physics, and more.

3. What are the challenges you face of running large jobs on Hopper? You can talk about any aspect including job scheduling, job failures, job scaling, run time variation, IO performance, and many more.

The biggest challenge users mentioned is actually the difficulty of getting through the queue. The recent retirement of the NERSC's other Cray system (Franklin, XT4) contributes to the crowdedness on Hopper because of user workload migration.

Other challenges mentioned include:

- No mechanism to control the mapping between logical topology into physical nodes that the submitted job was allocated. For example, if the MPI tasks have a 3D torus, there is no guarantee that these MPI tasks are perfectly mapped to a true and compact physical 3D torus.
- The increase of communication overhead and job variation caused by the above issue.
- Jobs sometimes hung.
- Jobs failed due to not enough memory.
- Parallel IO performance. MPI-IO does not scale well

4. What kind of scaling results do you see?

Some applications got very good scaling even with 100k+ cores. Some had good scaling until a certain size, such as 32k. Several users provided scaling results and tuning strategies, which we will include in Section III as user tuning stories.

5. What kind of job run time variation do you see?

About 20% is normal for most applications. Users do report seeing stable peak performance. Much larger variations in IO.

6. What tuning options have you used to improve job performance/scaling?

6a) Have you tried with different compilers and various compiler options? If yes, what are the results?

Many users have tried various compilers and flags. Surprisingly, many chose GNU compiler after comparison. Some users choose PGI, which is the default compiler on Hopper. Some users choose Cray compiler, to take advantage of the UPC and CAF support. Some Pathscale

compiler users migrated to use Cray compiler due to the lack of the ongoing Pathscale compiler support.

Some of the compiler flags users adopted include:

- PGI: -fast -Mipa=fast,inline -mp=nonuma
- GCC: -O3 -ffast-math -funroll-loops
- CRAY: -O3

6b) Have you tried changing default MPI environment variables? If yes, what are they and what are the results?

No users have tried to adjust various MPI buffers. We got lots of insufficient MPICH_UNEX_BUFFER_SIZE or MPICH_PTL_UNEX_EVENTS error messages for Portals on XT, but no more Gemini related such errors on XE6.

6c) Have you tried changing default MPI rank ordering? If yes, what are the results?

Most used MPI environment variable for tuning by users is MPI rank reordering. This is a simple yet effective run time tuning option that no source code modification, re-compilation or re-linking is required.

A couple of user application examples that benefited from MPI rank reordering will be described in Section III.

6d) Do you use hybrid MPI/OpenMP? If yes, how many threads per node? How do you choose the number of threads per MPI task? Do you use first touch memory for each local thread? Do you have some performance data with hybrid scaling results with various options?

Yes, some users have tried. They normally use 4 MPI tasks per node (one MPI task on each NUMA node), and 6 OpenMP threads per MPI task, as we suggested on our NERSC web site. Users have heard of "first touch", but none has actually tried it. One user tried to use other NUMA control mechanisms with 24 threads, but it proves to be hard, and performance was not better than with 6 threads.

6e) Have you tried with various aprun options, such as -cc, -S, -ss, for process/memory affinity control? If yes, what are the results?

Yes, users have tried with these options. The default option of "-cc cpu" seems to be very good for process affinity, but "-S", "-ss" options along with "-N" and "-d" have great impact on code performance.

The NUMA effect will be illustrated with a NERSC benchmark in Section III.

6f) Do you have to use fewer cores (than fully packed) per node in order to run or to get better performance? If yes, what are the results?

Yes from some users. It is helpful for being able to use more memory per process and for having fewer processes to share the memory and interconnect bandwidth. This is another simple yet effective run time tuning option. A user example will be described in Section III.

6g) Have you used huge pages? If yes, what are the results?

None of the users responded have used huge pages explicitly, although MPI uses huge pages explicitly.

The effects of using huge pages explicitly for two NERSC benchmarks and the impact from system issues related to huge pages will be described in Section IV.

6h) Have you used core specialization ("-r" option for aprun)? If yes, what are the results?

No users responded have tried core specialization. We are aware of the effect of core specialization on constraining OS jitter on the designated core, and using just the rest of the cores for application helped to improve the performance of an ocean code "POP" [4]. We are also aware of this feature being helpful for MILC and S3D performance [5], using two phases of MPI library (one in released version and one in development branch).

However, we tried this with a couple of applications (one user application, one NERSC benchmark code). The tests were done and also independent of any other optimizations (such as huge pages or malloc settings), and with the two suggested MPI environment variables (MPICH_NEMESIS_ASYNC_PROGRESS and MPICH_INIT_THREAD_SAFETY) being set in the released xt-mpich2/5.4.4 version, however, we do not see obvious improvement during production environment.

6i) What IO performance tunings have you tried? Non-default file striping? Parallel IO? What are the results?

Some users specify one IO write/reader per node. Some users use MPI-IO. One user achieved 4 times speedup by using max file striping instead of the default striping (see details in Section V).

During the user survey, we were able to help some users to debug their code scaling problems. One such example is: A user mentioned that his MPI/OpenMP hybrid never scales better than 1 thread per node. It turned out that he thought he was using the Cray compiler (with OpenMP enabled automatically), but without switching the programming environment module from PrgEnv-pgi to PrgEnv-cray, he is actually using the PGI compiler with the compiler wrapper "ftn", and without adding "-mp" explicitly, OpenMP was not turned on.

Many users benefited from the questions we asked them, since they have never heard of some of the tuning options we mentioned. Many plan to explore these options for their applications. We provided documentations and tuning suggestions to them via explicit user communications.

Another tuning option we experimented with but did not ask in the user survey is to set the system malloc settings:

export MALLOC_MMAP_MAX_=0
export MALLOC_TRIM_THRESHOLD_=536870912

We tried with the above settings on selected user applications and NERSC benchmarks, but no specific improvement was seen under production environment.

III. Applications Tuning Stories

Some runtime tuning stories of several user applications and NERSC benchmarks are described in this section. Since all the runs were performed under production environment, to minimize the runtime variation effect caused by the application placement in the topology, multiple apruns with different run time options were used in the same batch job, so that the same set of nodes can be used. Fastest run times from multiple batch jobs are used in the performance comparison to mimic dedicated environment.

A. S3D: MPI Rank Reordering

This tuning story is from NERSC users Hemanth Kolla and Evatt Hawkes [6]. S3D is a code for numerical simulations of turbulent combustion. In S3D, there is very little global communication and almost all communication is among nearest neighbors in the physical space. Reordering MPI ranks to place ranks that are contiguous in physical space proved to be beneficial (Fig 3), and the scaling of S3D is very good all the way up to 130k cores. Using 3,072 cores, the S3D run time per grid per time step is 176.2 sec using the default MPI rank ordering, and 165.7 sec with the custom rank reordering, a difference of 4%. The difference is bigger with larger jobs using more cores.

The ordering of the ranks in the physical domain is trivial (X direction first, then Y and finally Z direction). So the users know which ranks are contiguous to each other in the physical space (and hence communicate a lot with each other). A simple perl script is used to generate the custom MPICH_RANK_REORDER file where each line lays out the list of ranks that form a physically contiguous 3D block.

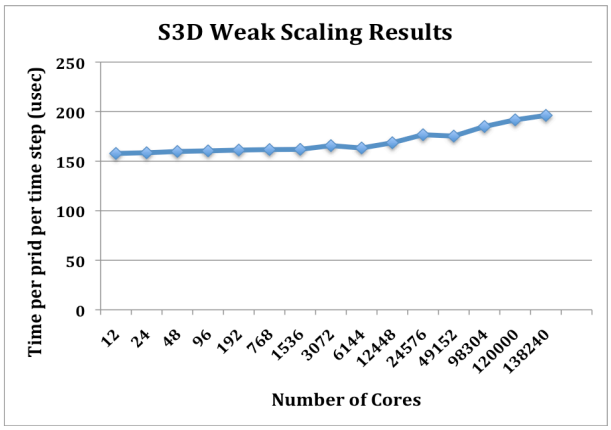


Figure 3. S3D run time per grid per time step using different number of cores on Hopper. This is weak scaling, so the ideal scaling is flat, and lower is better for scaling. 30x30x30 cube with c2h2 combustion chemistry. 50 time steps, no IO. (Courtesy of Hemanth Kolla and Evatt Hawkes)

B. PSOCI: Using Fewer Cores per Node

This tuning story is from NERSC user Jeffrey Tilson (personal communication). PSOCI (Parallel Spin Orbit Configuration Interaction) is a Chemistry code developed under the US DOE SciDAC-e award "Enhancing Productivity of Materials Discovery Computations for Solar Fuels and Next Generation Photovoltaics". Global Arrays version 5.0.3 (built with ANL's armci) with ga++ binding is used in PSOCI. The user left four cores free per node for local I/O etc.

Fig 4 shows the comparison of PSOCI run time with or without using fewer cores per node. Hamilton construction is phase 1, and Diag is phase 2. Using fewer cores per node, with 3,000 cores, speedup in phase 1 is 14%, speedup in phase 2 is 11%; and with 6,000 cores (with a different problem size), speedup in phase 1 is 6%, speedup in phase 2 is 15%.

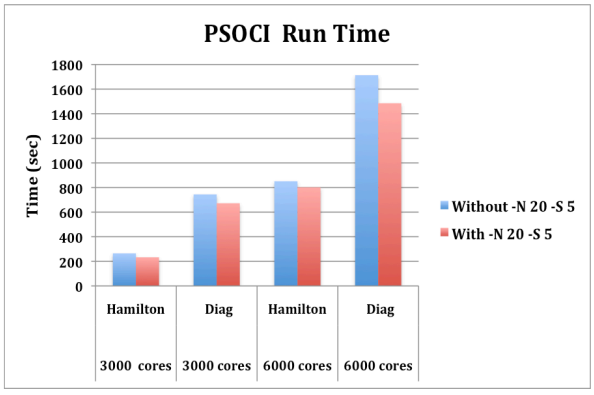


Figure 4. PSOCI run time with and without using fewer cores per node. 3,000 core and 6,000 core runs used different problem sizes. (Courtesy of Jeffrey Tilson)

C. *Hybrid MPI/OpenMP with Process and Memory Affinity*

Memory affinity on the compute nodes is not decided by the memory allocation, but by the initialization. Memory will be mapped to the NUMA domain that first touches the data and so it is important to initialize data carefully. This is referred to as the "first touch" principle. It is essential in hybrid MPI/OpenMP applications that each thread only accesses local memory from its local NUMA node to avoid the NUMA penalty of accessing memory from the remote NUMA nodes.

The following code example shows how to implement the "first touch" in an OpenMP parallel region. Local variables are initialized (not only allocated) in the first parallel region. Using same number of threads in the following parallel regions ensures local memory access for these variables.

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}

#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j]=b[j]+d*c[j];}
```

Fig 5 illustrates of the NUMA effects on Hopper with the STREAM benchmark [7]. The result shown here is from a single node, with pure OpenMP, using 1 to 24 threads. With "first touch" (TouchByAll), the memory bandwidth increases all the way up to 24 threads. While without "first touch" (TouchByOne), the memory bandwidth saturates at 6 threads.

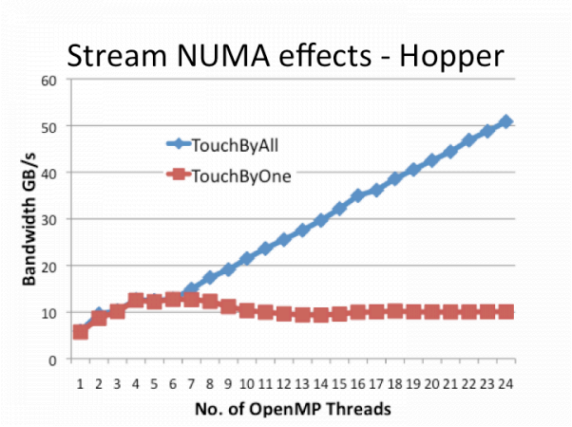


Figure 5. NUMA effects on Hopper with the STREAM benchmark. The result shown here is from a single node, with pure OpenMP, using 1 to 24 threads. (Courtesy of Hongzhang Shan)

It is very hard to do "perfect touch" for real applications. No real user applications reported from the user survey implemented "first touch". NERSC recommends not using more than 6 threads per node on Hopper to avoid NUMA effects. Using 4 MPI tasks per node, with 6 OpenMP threads per MPI task, and initializing local variables from each thread follows the "first touch" policy naturally. Memory for the local variables will be contained within the local NUMA node. In the example of STREAMS benchmark shown in Fig 5, using 4 MPI tasks with 6 threads per MPI task will reach the same bandwidth as using 1 MPI task with 24 threads with perfect first touch (TouchByAll), but avoid the difficulty of implementing first touch. (The only benefit of using 24 threads may be to allow more memory access per MPI task and to have larger and fewer MPI messages).

The "-S" option is especially important for hybrid MPI/OpenMP applications, since it is needed to spread the MPI tasks onto different NUMA nodes. Fig 6 demonstrates the number of MPI tasks per NUMA node with or without the "-S" option. The upper scheme uses "aprun –n 4 –d 6 …". Without "-S 1", all 4 MPI tasks are allocated on the same NUMA node. The bottom scheme uses "aprun –n 4 –S 1 –d 6 …". With "-S 1", there is only 1 MPI task per NUMA node. 6 threads will be forked for each MPI task on each local NUMA node.

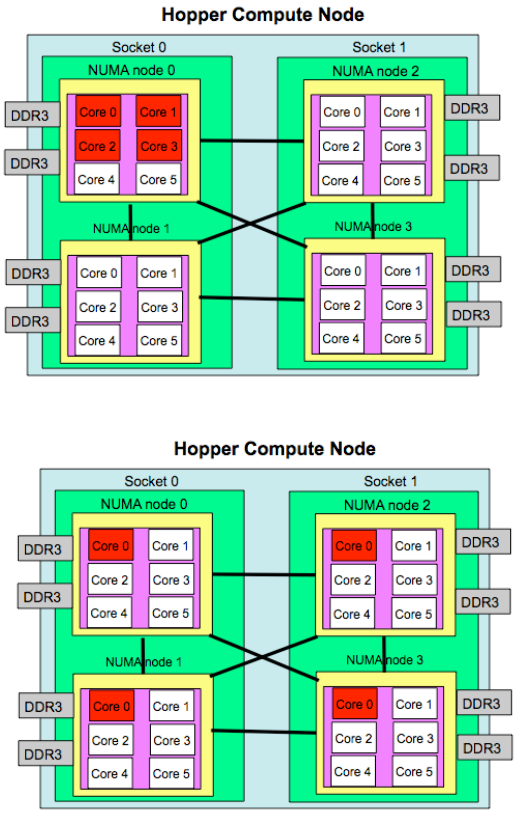


Figure 6. Illustration of the "-S" option in aprun. Upper: aprun –n 4 –d 6 …. Without "-S", all 4 MPI tasks are allocated on the same NUMA node. Bottom: aprun –n 4 –S 1 –d 6 …. With "-S 1", there is only 1 MPI task per NUMA node.

Fig 7 shows the results of a NERSC benchmark GTC [8] code with hybrid MPI/OpenMP implementation, GTC is a 3-dimensional code used to study microturbulence in magnetically confined toroidal fusion plasmas via the Particle-In-Cell (PIC) method.

A total of 24,576 cores are used. The sweet spot is to use 8 MPI tasks per compute node, with 3 threads per MPI task. NUMA penalty is seen when 12 threads per compute node are used. There is no result for 24 threads per node as this run did not complete correctly. The improvement of using "-S" to spread out MPI tasks onto different NUMA nodes is also shown in Fig 7 with 2, 3 and 6 threads (i.e., 12, 8, and 4 MPI tasks) per compute node.

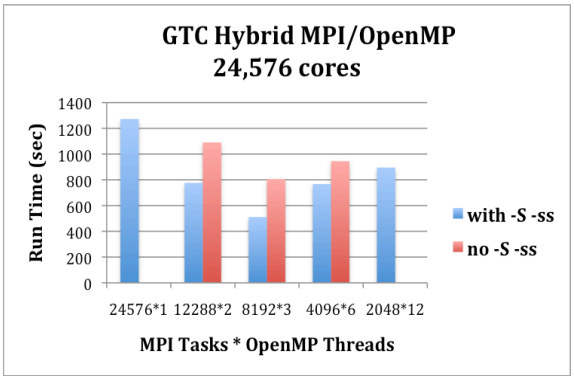


Figure 7. GTC hybrid MPI/OpenMP run time with various nuber of threads per node, also with and without "-S –ss" option in aprun. 3,000 core and 6,000 core runs used different problem sizes.

D. QLA: Code Tuning

This tuning story is form NERSC user Min Soe [9]. QLA (Quantum Lattice Gas Algorithm) is a mesoscopic unitary algorithm code to study quantum turbulence. It reaches super-linear scaling to 150,000 cores due to low MPI communication overhead (Fig 8). The strong scaling and weak scaling tests used different problem sizes. Only 2% run time difference is seen between using 276 cores and 110k+ cores with the weak scaling test.

There are 3 major steps in the code: unitary collide, stream, and rotate. Combining first two steps to minimize memory traffic resulted in 1.6x speedup. Hand tuning by simplifying expressions to eliminate redundant operations not recognized by the compilers gained additional 1.4x speedup. The tuning strategies also included using a mixture of non-blocking and blocking sends and receives for communication and computation overlap.

E. MFDn: Code Tunings

This tuning story is from NERSC users Hasan Metin Aktulga *et al.* [10,11]. MFDn is a nuclear physics code used for calculating the structures of light atomic nuclei based on first principles quantum mechanical model.

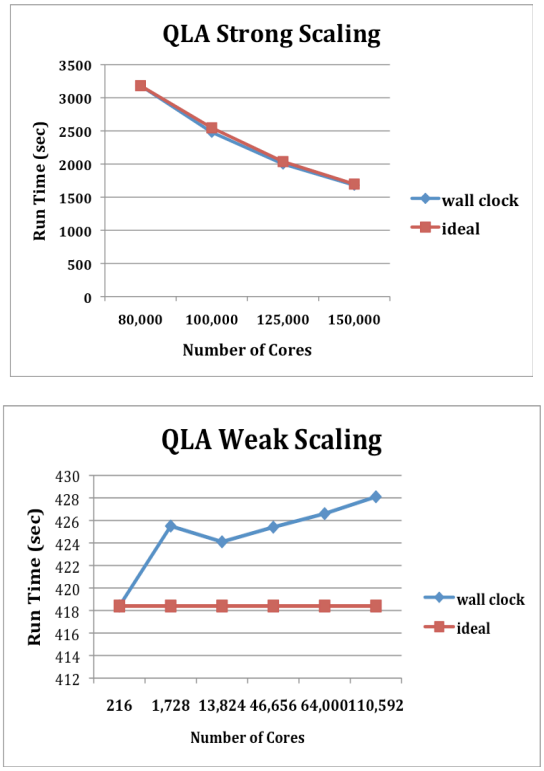


Figure 8. Scaling of Qutatum Lattice Gas Algorithm. Top: strong scaling. Bottom: weak scaling. The problem sizes used for the strong scaling and weak scaling are different. (Courtesy of Min Soe).

1) MPI Rank Reordering

This application involves computing the lowest eigenvalues and associated eigenvectors of a large many-body nuclear Hamiltonian, Hˆ. The Hamiltonian matrix is an extremely sparse symmetric matrix, but still has many non-zeros so that parallel processing is needed.

Fig 9 illustrates the different MPI process orderings for the 2D decomposition of the Hamiltonian H using a total of 15 processors, with 5 processors on the diagonal. A total of 4 MPI rank orderings are used: Diagonal Major (DM) ordering, Column Major (CM) ordering, Balanced Diagonal Major (BDM) ordering, and Balances Column Major (BCM) ordering. In order to balance the cardinality of communication groups, the processors are assigned to the upper right corners in BDM and BCM orderings.

It shows that topology-aware mapping of tasks and data to physical processors helps to avoid hot spot and congested links. For a typical large scale eigenvalues calculation, up to a factor of 2.5 improvement can be obtained in overall performance by using a topology-aware mapping (Fig 10).

BCM ordering is the best among all the orderings tested with various numbers of cores.

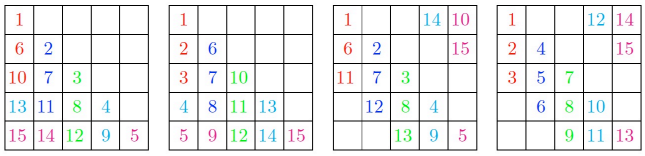


Figure 9. Illustration of 2D decompositin of the Hamiltonian H over processors in various rank orderings. Total of 15 processors (5 diagonal processors) in this case. From left to right: Diagonal Major (DM) ordering, Column Major (CM) ordering, Balanced Diagonal Major (BDM) ordering, and Balanced Column Major (BCM) ordering. (Courtesy of H. Metin Aktulga *et al.*)

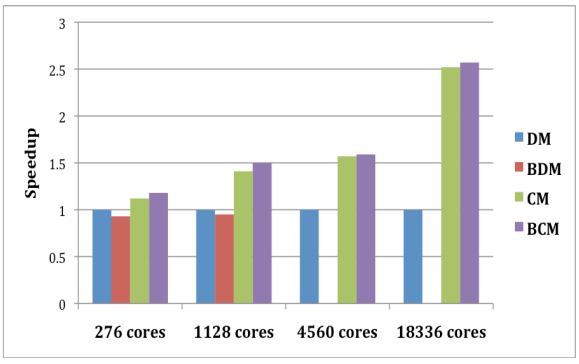


Figure 10. Speedup of MFDn with different numbers of total processors using different MPI rank orderings. BCM ordering is the best. (Courtesy of H. Metin Aktulga *et al.*)

2) Overlap Communication and Computation

Overlapping communication and computation is an effective way to reduce communication overheads in an application. But this technique could not be applied to applications with communication dominant MPI collective calls since non-blocking MPI collective primitives are not available.

Communication and computation overlap can be achieved in hybrid MPI/OpenMP implementations by designating one (or more) thread(s) for communication, and the other threads for computation.

The symmetric SpMV computations in MFDn can be divided into two subtasks, so certain data dependencies can be broken, which allows hiding communication during the SpMV phase. Fig 11 shows the speedup from various hybrid MPI/OpenMP implementations with overlapping communication and SpMV computations compared to pure MPI for the MFDn code.

BCM ordering is used for all tests, and the dynamic scheduling is used for OpenMP parallel regions. Total of 12,096 MPI processes are used for pure MPI implementation; and total of 2,016 MPI processes, with 6 threads per MPI task are used for hybrid MPI/OpenMP implementations.

- Hybrid A: hybrid MPI/OpenMP
- Hybrid B: hybrid A, plus: merge MPI_Reduce and MPI_Scatter into MPI_Reduce_Scatter, and merge MPI_Gather and MPI_Bcast into MPI_Allgatherv.
- Hybrid C: Hybrid B, plus: overlap row-group communications with computation.
- Hybrid D: Hybrid C, plus: overlap (most) column-group communications with computation.

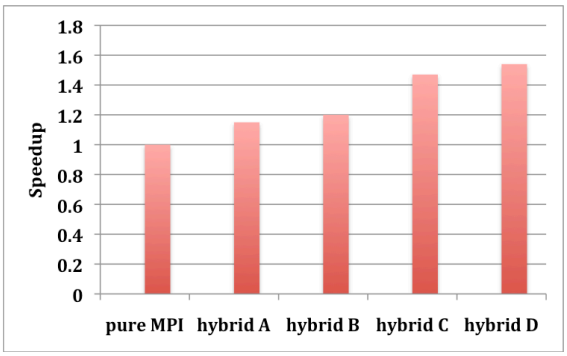


Figure 11. Speedup of MFDn with hybrid MPI/OpenMP implementations with overlaping communication and SpMV computations compared to pure MPI implementation. (Courtesy of H. Metin Aktulga *et al.*)

IV. Issues having Large Impact on Large Jobs

Large jobs are more sensitive to overall system issues simply because more nodes are needed. "Bad" nodes in the system tend to affect large jobs more due to the increased possibility of any "bad" node being assigned to a large job. In this section, two system issues that have affected a large number of large jobs on Hopper are described.

A. Huge Pages Issue

Huge pages can improve memory performance for common access patterns on large data sets.

Fig 12 shows NERSC benchmarks MILC [12] and GTC run time with and without huge pages. The benchmark code MILC represents part of a set of codes written by the MIMD Lattice Computation (MILC) collaboration used to study Quantum Chromodynamics (QCD), the theory of the strong interactions of subatomic physics. GTC 8,192 core and 24,576 core runs used different problem sizes. However, huge pages effect for these tests is within production environment variations.

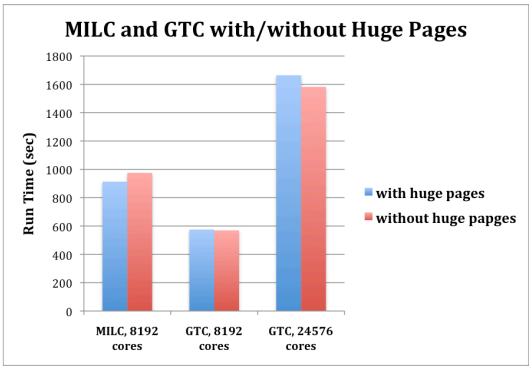


Figure 12. Run time of NERSC benchmarks MILC and GTC with and without huge pages. GTC 8,192 core and 24,576 core runs used different problem sizes.

Jobs explicitly use huge pages or large jobs which implicitly use huge pages by MPI are sometimes affected by not enough memory for huge pages error on the compute nodes. From June to Oct 2011, we reported that two NERSC benchmark applications (MILC and GTC) that use huge pages only had ~35% success rate.

Right after a system boot, about 30 GB of memory can be used for 2MB huge pages. But the available huge pages memory decreases over time due to memory fragmentation and memory leaks. Cray is actively pursing several bugs related to this problem. Meanwhile, we modified the local node health check script to identify these low huge pages memory nodes, and mark down and warm-boot the nodes manually. A bug in PMI library was also fixed. Jobs using huge pages are having much higher success rate now.

B. *Hung Jobs Issue*

Shortly after CLE4.0UP02 upgrade in Jan 2012, we received hung jobs report. Many users (50+) were affected, mostly running large jobs out of wall clock time. Huge amount of compute hours were wasted, 13.5M core hours had to be refunded to the users. Cray and NERSC teams worked intensively and held daily progress meetings for about a month. 8 "bad" nodes in a state that datagram packets cannot be received were identified. Rebooting these nodes helped the situation tremendously. Since the problem was intermittent, it was not conclusive that a successful job with one setting meant the problem was resolved. We worked with many users to test various MPI libraries and environment settings before Cray provided the two patches for the kGNI bugs that attributed to the "bad" nodes. No more hung jobs have been reported since mid March.

V. IO TUNINGS

I/O can also be a potential issue with large scale applications performance. If IO is not scaling well, then its time can dominate the run time for a large scale job.

A. *MPI-IO Block Size*

Good MPI-IO performance continues to be a challenge for real applications to achieve. This is because scientific applications often do not write and read data in the method that is most beneficial for the parallel file system. For example, a parallel file system like Lustre or GPFS performs well when large contiguous blocks of data are written. Applications that do not exhibit this I/O pattern can suffer in performance. Fig 13 shows the effect of small block size can have on MPI-IO performance on the Lustre file system. The results are obtained using the IOR benchmark [13].

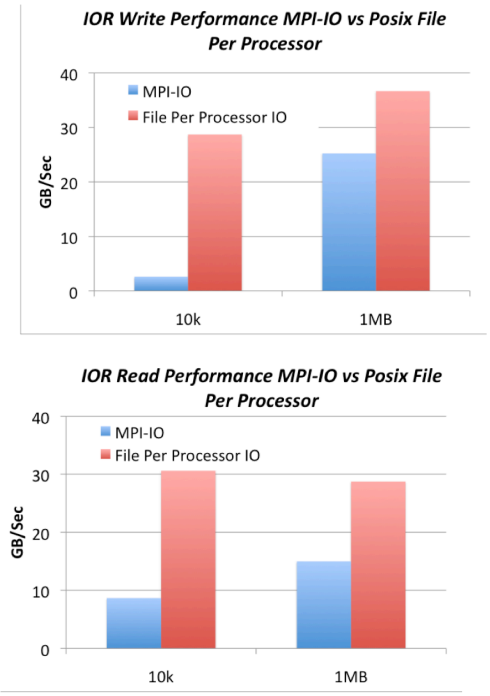


Figure 13. IOR write (top) and read (bottom) performance MPI-IO *vs.* Posix file per processor using different block sizes on Hopper scratch Lustre file system. (Courtesy of Yushu Yao)

In addition to a local Lustre file system on Hopper, NERSC also mounts a global GPFS based file system. Because Hopper does not have native GPFS clients on the compute nodes, it uses an I/O forwarding layer developed by Cray, called the Data Virtualization Service (DVS). Our initial testing showed MPI-IO performance to the GPFS based file system was about the same as on with Lustre. A

collaboration has been formed between Cray and NERSC to address MPI-IO performance through DVS to the GPFS file systems.

B. MFDn J-scheme: Lustre File Striping

This IO tuning story is from NERSC users Hasan Metin Aktulga *et al.* [14]

The application used here is a flavor of MFDn (called J-Scheme MFDn) from an out-of-core implementation to an in-core algorithm (using MPI one-sided primitives) plus Lustre file striping tunings. File striping is a way to increase IO performance since writing or reading from multiple Object Storage Targets (OSTs) simultaneously increases the available IO bandwidth.

Fig 14 shows J-Scheme MFDn overall run time (computation plus IO) improved from an out-of-core implementation to an in-core implementation along with default and max Lustre file stripings.

- Problem size: ^{6}Li, Nmax = 12, J = 0 to 4
- Total size of all data-blocks: 2.7~10 GB
- Number of blocks: 2.5×10^{5}
- Total number of block accesses: $1.1 \sim 1.6 \times 10^{8}$
- Default file striping on Hopper = 2
- Max file striping on Hopper = 156
  % lfs setstripe -c -1

The overall speedup with J = 2 is 30% from out-of-core default striping to out-of-core max striping.

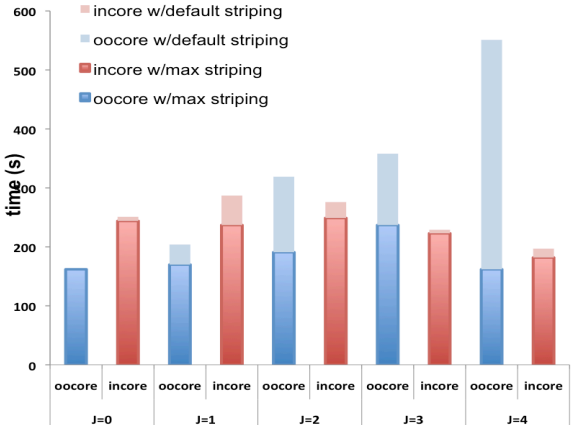


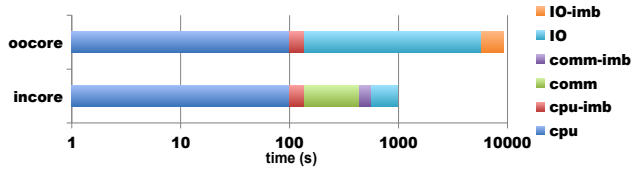
Figure 14. J-Scheme MFDn overall run time (computation plus IO) with out-of-core and in-core implementations using default and max file stripings. (Courtesy of H. Metin Aktulga *et al.*)

With a much larger problem, the gain with Lustre file striping is bigger. Fig 15 shows the overall run time for a larger problem size with default and max file stripings using out-of-core and in-core implementations.

- Problem size: ^{6}Li, Nmax = 14, J = 3
- Total size of all data-blocks: 67.1 GB
- Number of blocks: 7.4×10^{5}
- Total number of block accesses: 7×10^{8}

Using max striping instead of default striping results in 4x speedup in out-of-core's total running time. It also gains 4x speed up in in-core's IO time. With the time needed to load data from disk to distributed memory of processors in in-core implementation, the overall speedup in total run time is 30%.

Default Striping:



Default and Max Striping:

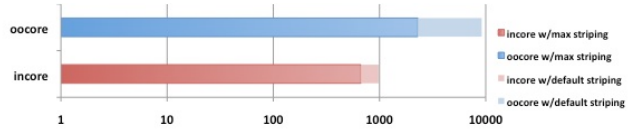


Figure 15. J-Scheme MFDn run time with out-of-core and in-core implementations using default and max file stripings for a larger problem size. Top: Computation and IO time with default striping. Bottom: Total run time with default and max stripings. (Courtesy of H. Metin Aktulga *et al.*)

VI. SUMMARY

Running large scale jobs on a large system is very challenging. The goal of this study is to come up with some practical tips for our general users by learning from our selected user successful stories and staff benchmarking experiences. Effective run time tuning options are most welcome since they do not need to recompile or re-link the application codes.

In summary, some of the tips are:

- Experiment with different compilers and compiler flags.
- MPI rank reordering is a simple and effective run time tuning method if you know your application's communication pattern well.
- Try to use fewer cores per node to allow more memory access and more memory and interconnect bandwidth per process.

- Hybrid MPI/OpenMP is encouraged on Hopper since it also reduces the memory footprint. NERSC suggests not to use more than 6 threads on one node. Aprun options process (-S) and memory affinity options (-ss) are essential to ensure the MPI tasks being spreaded out across different NUMA domains.

- Consider overlapping communication and computation in hybrid MPI/OpenMP.

- Advanced tuning options such as using huge pages are worth trying.

- Tune block sizes for MPI-IO; Use different stripe sizes for large Lustre files.

ACKNOWLEDGMENT

We would like to thank many NERSC users who participated in Running Large Jobs on Hopper user survey and provided valued feedback on various aspects. Special thanks to Hemanth Kolla, Evatt Hawkes, Jeffrey Tilson, Min Soe, Hasan Metin Aktulga, and Pieter Maris for providing performance tuning stories and test cases. We would like to thank Hongzhang Shan for the STREAM NUMA effect plot and Yushu Yao for the MPI-IO performance plot.

The authors are supported by the Director, Office of Science, Advanced Scientific Computing Research, U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

REFERENCES

[1] NERSC Web pages for Hopper: http://www.nersc.gov/users/computational-systems/hopper/

[2] Hopper Performance and Optimizations web page: http://www.nersc.gov/users/computational-systems/hopper/performance-and-optimization/

[3] Hopper Run Time Tuning Options web page: http://www.nersc.gov/users/computational-systems/hopper/running-jobs/runtime-tuning-options

[4] C. Carroll. Cray Operating Systems Road Map. Proceedings of Cray User Group 2010, Edinburgh, UK.

[5] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella. Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems. Proceedings of Cray User Group 2012, Stuttgart, Germany.

[6] J.H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E.R. Hawkes, S. Klasky, W.K. Liao, K.L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende and C.S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D, 2009, Computational Science & Discovery, 2:015001.

[7] STREAM benchmark: http://www.cs.virginia.edu/stream

[8] GTC: http://w3.pppl.gov/theory/proj_gksim.html

[9] G. Vahala, M. Soe, B. Zhang, L. Vahala, J. Carter, and S. Ziegeler. Unitary Qubit Lattice Simulations of Multiscale Phenomena in Quantum Turbulence. Proceesings of SuperComputing 2011, Seattle, WA.

[10] H.M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J.P. Vary. Topology-aware Mappings for Large-Scale Eigenvalue Problems. Submitted to Euro-Par 2012. International European Conference on Parallel and Distributed Computing.

[11] H.M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J.P. Vary. Scalable Eigensolver for Multi-core Platforms. Submitted to SC2012. The International Conference for High Performance Computing, Networking, Storage and Analysis.

[12] MILC: http://www.physics.indiana.edu/~sg/milc.html

[13] IOR code: http://computing.llnl.gov. Scalable I/O Benchmark Project. Download: http://sourceforge.net/projects/ior-sio/

[14] H. M. Aktulga. On Reducing I/O Overheads in Large-Scale Invariant Subspace Projections. Workshop on Algorithms and Programming Tools for Next-Generation High-Performance Scientific Software., HPSS 2011, in the context of Euro-Par 2011, August 29, 2011, Bordeaux, France.