Scaling the MPT Software on the Cray XT5 System and Other New Features

Mark Pagel,
Kim McMahon, David Knaak
Cray Inc.

ABSTRACT: *The MPT 3.1.0 release allowed MPI and SHMEM codes to run on over 150,000 cores and was necessary to help the Cray XT5 at ORNL to achieve over a petaflop in performance on HPL. MPT 3.1.0 was also used in quickly getting numerous other applications to scale to the full size of the machine. This paper will walk through the latest MPT features including both performance enhancements and functional improvements added over the last year including improvements for MPI_Allgather, MPI_Bcast as well as for MPI-IO collective buffering. New heuristics for better default values for a number of MPI environment variables resulting in less application reruns will also be discussed.*

KEYWORDS: MPI, SHMEM, MPI-IO, Petaflop

1. Introduction

The Cray XT5 systems have been purchased by customers with larger and larger total core counts. This is especially true for the Oak Ridge National Laboratory machine called Jaguar. The previous Jaguar machine was an XT4 with around 30,000 cores. In order to support the XT5 machine MPI and SHMEM needed to scale to over 150,000 cores. In addition to the scaling changes needed to support the Cray XT5 at ORNL the MPT 3.1.0 and MPT 3.2.0 releases also contain a number of new functionality and performance features. This paper describes the experience and changes needed to scale the MPT software as well as some of the new features recently added.

2. Scaling to Over 150,000 cores

MPI and SHMEM Hard Limit Modifications

Prior to version 3.1.0, the Cray Message Passing Toolkit (MPT) had support for a maximum of 65,536 ranks for a single MPI job. Several internal data structures and various internal limits needed to be modified to allow support for a larger number of MPI ranks.

One of the basic problems was the use of a 16-bit field to store the MPI rank number in some key data structures associated with internal packet headers and matchbits entries. Originally these MPI rank fields were stored as 16-bit fields to help conserve memory in latency-sensitive data structures. It is also probable that the original authors did not foresee the need for MPI programs running with more than 65,000 MPI ranks. However, in order to support the extreme scaling of the ORNL XT5 system and beyond, these rank values were changed to be full 32-bit (unsigned integer) fields. This change caused some internal data structures to increase in size. However, increasing the size of other internal data structures was not feasible. Because of this, additional structure-specific modifications were required to keep the size of the structure constant.

To allow larger MPI rank values in the MPI matchbits, reducing the number of bits used for the MPI tag field was necessary. This reduced the maximum MPI TAG value from 32 bits to 24 bits. Likewise the MPI_TAG_UB attribute was set to correctly reflect our reduced tag space size.

Modifications were also necessary for a central structure used by the MPI portals device driver. One of the internal portals header structures is a 64-bit structure,

and cannot be easily expanded. The MPI rank value needs to be stored in this header structure. Keeping the MPI rank value in this structure is required for checkpoint/restart capability, since the physical location of a rank may change during a restart operation.

In order to accommodate a larger MPI rank value in this limited space, we chose to support an 18-bit rank field which allows support for a maximum of 256K (262,144) MPI ranks. However, adding the 2 bits onto the rank field required taking away 2 bits from an allocation_ID field. The allocation_ID field, which is used as part of the Portals driver messages matching ability, was now reduced to 14 bits. The reduction in size of this field required an implementation of a new algorithm to manage this resource more precisely. The existing increment/wraparound technique for the allocation_ID field was converted to a managed push-down stack algorithm. In addition, code was added to handle the case when this resource becomes exhausted during a job, by integrating this resource into our internal state engine flow-control feature. This allows an MPI job to gracefully survive an allocation_ID out-of-resource situation should that condition ever arise.

The changes to SHMEM also modified some hard limits for how high SHMEM jobs could scale on XT systems. The new limit is 256,000 SHMEM PEs.

In order to support higher scaling, changes were made to the SHMEM header files that require a recompile when using this new version. The new library will detect this incompatibility and issue a FATAL error message telling you to recompile with the new headers.

Testing at Scale and Additional Modifications

In order to test some of the new scaling modifications made to the MPT software, prior to having access to the actual XT5 hardware, our plan was to use the existing ORNL jaguar XT4 system, and over-subscribe those 30,000 cpus with multiple MPI processes. This plan required several changes to the existing Application Level Placement Scheduler(ALPS) and MPI software.

A custom version of the ALPS software was enhanced to support an 'emulation' mode. This mode allowed an over-subscription of processes to cpus, with a maximum of 32 processes on a single node. In addition, the MPT software was modified to support up to 32 processes per node, and an additional feature was added to allow each MPI process to call sched_yield() in the appropriate internal progress engines to allow forward progress and fairer scheduling when the cpus are over-subscribed.

During dedicated jaguar XT4 time, we ran with the custom ALPS emulation mode to over-subscribe the jaguar nodes by a factor of 5-6x (running 20-24 processes on each node). Since the nodes were heavily over-subscribed, this exercise was about functionality, not performance.

We were able to run a multi_pingpong MPI test to verify functionality. In this test, the desired number of MPI processes are launched, with each process using MPI_Send and MPI_Recv functions to ping pong data messages back and forth between a pair of processes. The data values were modified and verified for each pass. This test was run successfully at several configurations, up to 180,000 processes, each sending one thousand 16384-byte messages between the pairs. Several MPI collective functions were also tested, including MPI_Allreduce, MPI_Reduce, MPI_Bcast and MPI_Barrier. These tests all ran successfully at 150,000 processes, and the resulting data was verified to be correct.

One potential issue was identified during the over-subscription testing. The MPT collective optimization startup routine was taking a very long time to complete during the launch of the over-subscribed MPI jobs. This startup routine needed to be disabled in order to complete the tests. This performance problem turned out to be an issue when we tested on real hardware as well and was then resolved as will be described later.

In addition to identifying and fixing some of the hard limits for maximum MPI rank value in our software, several other existing features were enhanced or expanded to aid in application scaling to these extreme process counts.

The first feature we looked at was the default values for various MPI environment variables. In particular, we looked closely at the environment variables used to tweak the MPI portals device driver, since that is the driver that takes the brunt of the scaling work on an XT5.

In MPT 3.0.0 and prior versions, we used static constants for these default values. However, when applications scale up to 150,000 processes, we found that the underlying network communication patterns are quite different from those network patterns at small process counts. This required us to adjust our MPI portals driver assumptions as well, since it was clear that static values weren't the right choice.

To address this, we created a set of auto-scaling MPI environment variables. These are environment variables whose default values change as a function of the total number of ranks of the given MPI job. Some of these default values increase as the number of ranks increases, others decrease. This feature is designed to allow higher scaling of MPT jobs with fewer tweaks to environment variables and to help reduce the number of required times to run the MPI job. If necessary, the user is still able to override any of these auto-scaling defaults, simply by setting these variables to the desired value prior to the job launch.

The performance of the MPT collective optimization startup routine was also investigated at scale. This was identified in the over-subscription work done on the XT4 system as a potential performance bottleneck. Analysis of the optimization startup routine showed the use of two small message MPI_Allgather calls consuming the large majority of the time. Further analysis of the small message MPI_Allgather function confirmed that the current algorithm was not scaling well on the XT5. To resolve this, the MPI_Allgather algorithm used for small messages (2048 bytes or less) was re-written to better suit the Portals/Seastar interconnect properties. For some message sizes, this new algorithm resulted in a 12X performance boost. In addition, the new algorithm significantly reduced the MPI_Init startup time (including the collective optimization initialization) on very large jobs. For example, the start-up time for a 86,000 MPI rank job went from 280 seconds down to 128 seconds.

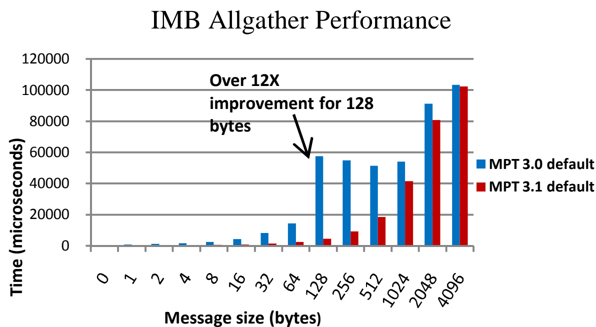

Chart 1. MPT 3.0 default compared with MPT 3.1 with optimized MPI_Allgather default for 4096pes on an XT5 (lower is better)

Another MPT feature that was enhanced to allow for better scaling was the allocation of internal MPI message headers. In prior versions, if MPI ran out of headers, the program would abort and request the user increase the limit via the MPICH_MSGS_PER_PROC environment variable. With the MPT 3.1.0 release, if additional message headers are required during program execution, MPI dynamically allocates more message headers in quantities of MPICH_MSGS_PER_PROC. The user is able to specify the value of MPICH_MSGS_PER_PROC via an environment setting if desired.

3. MPI-IO Collective Buffering

In addition to these scaling features several new features have been recently added. One of these features is MPI-IO collective buffering improvements. The MPT 3.0.0 release (and earlier) supported the MPI-IO optimization called collective buffering. The algorithm used in that release was the original algorithm that is in the ROMIO implementation of MPI-IO, which is the base for the Cray implementation of MPI-IO. The collective buffering is controlled by several MPI-IO hints, which are documented in the "intro_mpi" man page. Using the hints, collective buffering can be enabled, disabled, or left in the default automatic mode.

After analysis of the I/O performance of some benchmarks with MPT 3.0.0, we added a new collective buffering algorithm in the MPT 3.1.0 release while maintaining the old algorithm. The environment variable MPICH_MPIIO_CB_ALIGN was introduced (by Cray for our implementation) to control which algorithm to use, with values of 0 or 1. The new algorithm reduced Lustre extent lock contention by dividing the I/O workload along Lustre stripe boundaries rather than a simple equal division of the workload. This resulted in significant improvement in some applications but not all.

The MPT 3.2.0 release contains a third algorithm which does a better job of dividing the workload such that each file stripe is accessed by one and only one aggregator across all the collective I/O calls for that file. It is controlled by the same environment variable, with a value of 2.

A white paper written by David Knaak and Dick Oswald, "Optimizing MPI-IO for Cray XT Applications" is available that gives more details.

Here are 4 sets of benchmark results comparing the I/O bandwith in MB/sec for the 4 modes: without collective buffering (that is, collective buffering disabled), and with collective buffering with algorithms 0, 1, and 2. For IOR, there is also a comparison with POSIX shared-file.

MPI-IO Benchmark descriptions and results

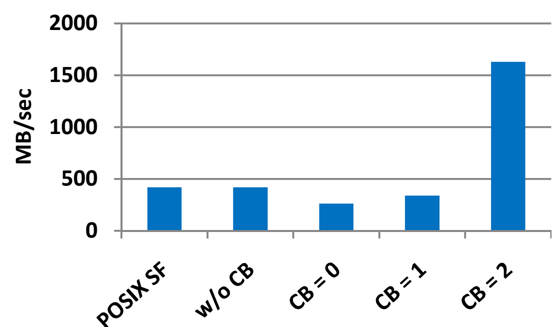

Chart 2. IOR benchmark 1,000,000 bytes , MPI-IO API , non-power-of-2 blocks and transfers, in this case blocks and transfers both of 1M bytes and a strided access pattern. Tested on an XT5 with 32 PEs, 8 cores/node, 16 stripes, 16 aggregators, 3220 segments, 96 GB file

As you can see from Chart 2 collective buffering in MPT 3.2.0 (CB=2) is significantly the best performer.

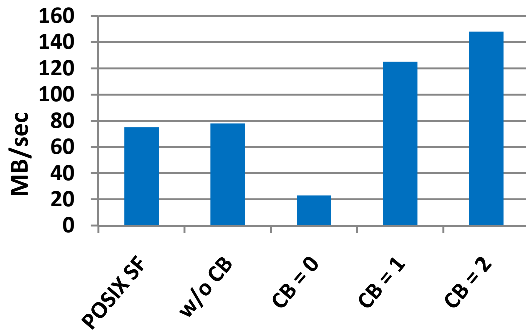

Chart 3. IOR benchmark 10,000 bytes , MPI-IO API , non-power-of-2 blocks and transfers, in this case blocks and transfers both of 1M bytes and a strided access pattern. Tested on an XT5 with 32 PEs, 8 cores/node, 16 stripes, 16 aggregators, 3220 segments, 96 GB file

In the IOR benchmark test in Chart 3 the MPT 3.2 version performs about 2 times better than without collective buffering and significantly better than the default collective buffering.

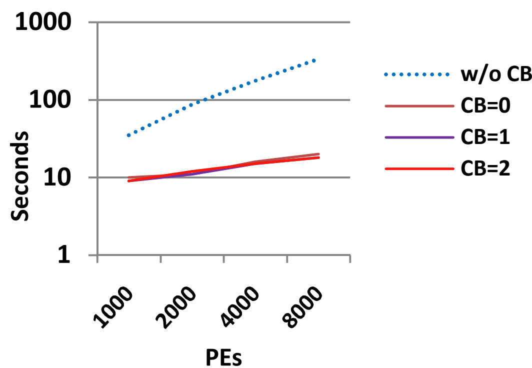

Chart 4. HDF5 format dump file from all PEs. Total file size 6.4 GB. Mesh of 64M bytes 32M elements, with work divided amongst all PEs. Note that disabling data sieving was necessary. Tested on an XT5, 8 stripes, 8 cb_nodes

In Chart 4 the original test did not use collective buffering and had very poor scaling. For example, without collective buffering, 8000 PEs take over 5 minutes to dump. Also very little difference in the different implementations of collective buffering for this case.

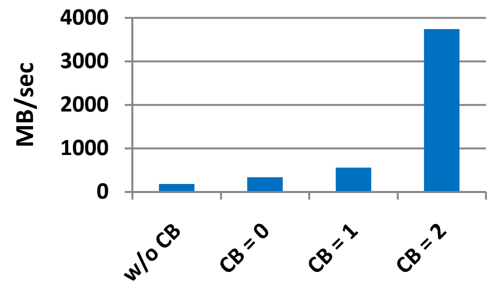

Chart 5. HYCOM MPI-2 I/O On 5107 PEs, and by application design, a subset of the Pes(88), do the writes. With collective buffering, this is further reduced to 22 aggregators (cb_nodes) writing to 22 stripes. Tested on an XT5 with 5107 Pes, 8 cores/node

In Chart 5 the HYCOM MPI-2 I/O application shows the most dramatic improvement from the most recent implementation for collective buffering in MPT 3.2.0. The current plan is to make the CB=2 implementation the user default in the MPT 3.3.0 release planned for June 2009 but this will be depend on user experience gained over the next several weeks.

4. Improvements to MPI_Bcast and MPI_Reduce

Other new features include performance improvements for several MPI collectives. Both the MPI_Bcast and MPI_Reduce algorithms were significantly enhanced in MPT 3.1.1 and MPT 3.1.2 respectively. Prior to these enhancements, basic tree algorithms were used for these collectives. In order to provide better overall performance, and performance that scales with large systems, we've introduced new SMP-aware algorithms for these commonly used MPI collectives.

The new SMP-aware algorithms are the default algorithms in MPT 3.2.0. The key to these SMP-aware algorithms is they take into consideration and exploit the physical location of the ranks involved in the collective operation. Emphasis is placed on reducing network communication as much as possible. The focus of this algorithm is to first manipulate and/or reduce the amount of data locally on each node. Since this step only involves local operations, all of the nodes participating in the collective can do this in parallel. The amount and type of local operations are dependent on the algorithm semantics. When global network communication is required, both the amount of processes involved in the network communication as well as the amount of data required to traverse the network is significantly reduced. This technique can result in a sizeable performance gain

over a simple tree-based algorithm, especially when scaling to high process counts.

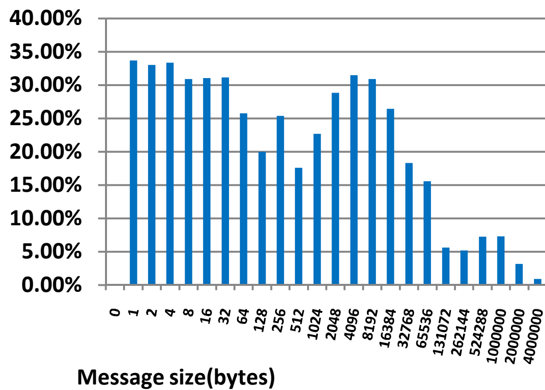

Chart 6. Percent Improvement of SMP-aware Bcast comparing default MPT 3.2 against default MPT 3.0 on XT5 with 256 PEs

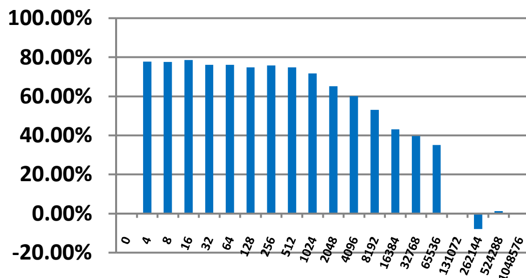

Chart 7. Percent Improvement of SMP-aware Reduce comparing default MPT 3.2 against default MPT 3.0 on XT5 with 256 PEs . For this chart we show what would happen if we didn't have the cutoff at 128K to switch back to the original algorithm. See mpi man page for more info on the MPICH_REDUCE_LARGE_MSG env variable.

In Charts 6 and 7 the performance improvements for MPI_Bcast and MPI_Reduce can be seen especially for message sizes of 65K bytes and smaller where it is at least 15 percent better and nearly 80 percent better for MPI_Reduce.

5. Other Features

In addition to the above features several other new features were added. These include:

- Moving from ANL MPICH2 1.0.4p1 to MPICH2 1.0.6p1
- Cpu affinity support
- MPI Thread Safety
- Wildcard matching for filenames when using MPICH_MPIIO_HINTS environment variable
- Support for the Cray Compiling Environment (CCE) 7.0 compiler
- An MPI Barrier before collectives feature
- Improved performance for on-node very large discontiguous messages

These will be described briefly below.

The move to MPICH2 1.0.6 fixed a number of problems, some of which we had already applied to the MPT 3.0 release on an as needed basis. Many of the new features added in MPICH2 1.0.6 don't affect XT users but it is important that we stay close to the MPICH2 latest releases to take advantage of improvements they are making. That being said, we have already found and fixed several regressions introduced by MPICH2 1.0.5 or MPICH2 1.0.6. Here are just some of the features in MPICH2 1.0.5 and MPICH2 1.0.6 listed from the ANL MPICH2 changes document that may affect XT users:

- Performance improvements for derived datatypes (including packing and communication) through loop-unrolling and buffer alignment.
- Performance improvements for MPI_Gather when non-power-of-two processes are used, and when a non-zero ranked root is performing the gather.
- MPI_Comm_create now works for intercommunicators.
- Many other bug fixes, memory leak fixes and code cleanup.

Support for CPU affinity has been added to this release. This allows MPI processes to be pinned to a specific CPU or set of CPUs, as directed by the user via the new aprun affinity and placement options. Affinity support is provided for both MPI and MPI/OpenMP hybrid applications.

Support has been added for wildcard pattern matching for filenames in the MPICH_MPIIO_HINTS environment variable. This allows easier specification of hints for multiple files that are opened with MPI_File_open in the program. The filename pattern matching follows standard shell pattern matching rules for meta-characters ?, \\, [], and *.

Support was added that allows the x86 ABI compatible mode of the Cray Compiling Environment (CCE) 7.0 to be compatible with the Fortran MPI bindings for that compiler. The CCE 7.0 compiler was released Q408 and has a dependency on the MPT 3.1 release or newer.

In some situations a Barrier inserted before a collective may improve performance due to load imbalance. This feature adds support for a new environment variable MPICH_COLL_SYNC which will cause a Barrier call to be inserted before all collectives or

only certain collectives. See the “mpi” man page for more information.

This MPI Thread Safety feature is enabled by setting the MPICH_MAX_THREAD_SAFETY env variable. Setting this new env variable specifies which thread-safety level should be returned by MPI_Init_thread() in the "provided" argument. The MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, as well as the MPI_THREAD_SERIALIZED cases are high-performance implementations in the main MPI library. The MPI_THREAD_MULTIPLE case is not a high-performance implementation and for performance reasons is in a separate library that replaces the main MPI library, therefore the library reference “-lmpich_threadm” must be included when linking. See the “mpi” man page for more information.

Finally a new algorithm for the on-node SMP device to process large discontiguous messages was also added. The new algorithm allows the use of our on-node Portals-assisted call that is used in our MPT 3.0.0 single-copy feature rather than buffering the data into very small chunks as was previously being done. Some applications have seen as much as a 3X speedup with discontiguous messages in excess of 4M bytes.

6. Conclusion

The Cray MPT software stack continues to improve as can be seen the number of functional and performance features added in the MPT 3.1 and MPT 3.2 releases. The challenges in scaling software with limited access to large machines will continue to be a challenge in the future but the technique of over-subscribing to test functionality can be beneficial in discovering and resolving some issues prior to the build up of petascale machines. Improving performance of the MPI-IO collective buffering should provide noticeable speedups on many applications and continues to be one of the key areas where we believe further improvements in the MPT software will be realized.

About the Authors

Mark Pagel is the manager of the MPT group at Cray Inc. and can be reached by email at pags@cray.com. Kim McMahon and David Knaak are key members of the MPT development team and continue to make many improvements in the XT MPT software.