A Need for Better Management of Heterogenous HPC Resources

Kenneth Moreland
Sandia National Laboratories

Chuck Atkins
Kitware, Inc.

1 Introduction

Achieving computational rates beyond the petascale requires an increasing amount of heterogeneity in our high-performance computing (HPC) hardware. This heterogeneity, in turn, means that a node of an HPC system can no longer be considered a monolithic resource. Rather, a node has many individual components such as processors, cores, SMT threads, accelerators, and tierd memories that must be further allocated and managed by... something.

Currently, that something is an ad-hoc mix of arguments and environments when launching jobs. We follow the same process used on prior HPC systems with nodes of uniform components; unfortunately, the shims introduced to provide the additional specification of node-level components are inconsistent and unwieldy. As we shall see, even at our current moderate level of heterogeneity our effective utilization of HPC software is hampered by poor resource management.

Future systems will continue to grow in heterogeneity both in number and type of resources. Our current approach to resource management cannot scale. We need a more cohesive approach to managing heterogeneous resources in HPC systems.

2 A Case Study

The motivation for this position paper comes from the issues our team has encountered while supporting the ParaView application [4] on modern HPC systems such as Trinity [3]. ParaView is a large and complex visualization application that has been developed for many years which, in light of industry trends toward multi-core processing, has been aggressively introducing intra-node parallel techniques.

Initially, installations of the latest version of ParaView performed poorly in large part because the previous launching technique spawned many processes per-node, which, along with the new multi-core code, in turn, spawned overlapping threads starving one another for cycles. The solution proved to be quite involved as the optimal number of processes was somewhere between one process per-node and one per-core. In addition to manually specifying on-node MPI decomposition via scheduler arguments, the allocation and affinity placement of threads used within each process is left to a variety of configurations. For example, code that relies on OpenMP can be configured through environment variables, while other libraries, such as Intel Thread Building Blocks (TBB), or just Unix pthreads, have a different set of independent environment variables. Tracking down the documentation for each, if it exists at all, can be difficult at best.

Further complications occurred because of the different threading solutions used by ParaView. The surface rendering, ray casting, and computational geometry libraries each have mechanisms for configuring their threads. Even though the libraries run mutually exclusively, the configuration of one can, and often does, interfere with the other. For example, configuring one to bind threads to a desired set of cores can inadvertently cause another to disregard said cores and serialize itself. We continue to struggle with these issues.

Finally, all these configurations are presupposed on the notion that the job launching system can control these resources in the first place. To be fair, job scheduling systems are getting better about providing controls for heterogeneous resources. SLURM and Torque, for example, both have "generic resource" features [1, 2] allowing users to specify site-specific resources like GPUs and configure the environment accordingly. Still, this requires end users to know everything about both the site-specific configuration options and the needs of the application being run.

3 Problems with Current Resources Management

As demonstrated in the previous case study, our current approach for managing the allocation of heterogeneous resources (or lack thereof) through job scheduling and execution imposes several serious problems.

1. The problem of resource allocation is pushed to end users. Those with the most expertise in how the software should run are (hopefully) those writing the HPC software. They are the ones that should be tasked with defining the optimal resource allocation. However, the tools to managed resource allocation are outside the scope of HPC software layers like MPI and are instead dropped to the environment customized to the HPC installation. This consequently causes the burden to drop to end users attempting to run the software, which is sometimes the group least equipped to do so. End users are also often the largest and least cohesive group involved with HPC software, so the problem is likely re-solved numerous times.
2. Resources are specified in a "backward" manner. After being assigned some collection of nodes, current job schedulers assign processes to cores by taking the total number of processes to create and the order in which cores are assigned. This, however, is backward from how developers usually think about core/resource allocation. Modern programming practice for parallel HPC generally dictates that problems are decomposed into tasks or blocks, which means a developer is focused on what collection of cores and other resources are most effective for a particular process. Getting each process to have the desired cores requires the user to take the desired number of cores per process, convert that to the number of processes per node, convert that to the number of total processes for the job, and then figure out what order of process-to-core assignment results in the desired grouping of cores.
3. There is no consistent way to specify all resources. Nodes are assigned by the scheduler. Cores are assigned to processes partially by the execute command, but also require configuration of the threading libraries. OpenMP is configured through environment variables, but most other threading libraries are configured through their API. CUDA devices also have to be manged through a library API, but the management also has to coordinate with MPI so that resources are assigned evenly (even though MPI provides little help

in this regard). Newer resource types like burst buffers have no conventional interface at all, and so the means to allocate it changes from system to system.

4. The management of one library can conflict with the management of another. For example, in our experience we have found that configuring one multithreaded library to the correct number of threads on the correct cores can unintentionally restrict another library from running on those cores even though the two run mutually exclusively.

4 Proposed Solution

This position paper is a call to the HPC software stack community to provide better solutions for runtime resource management. Our vision is to have a central management library that is aware of the resources available on the system and the respective libraries that interface with them. For lack of a better term, we will use *Resource Manager Library* to refer to this hypothetical library.

The *Resource Manager Library* coordinates with the job scheduler and other resource libraries to give a consistent interface to resource management. The *Resource Manager Library* is not meant to dictate or replace the interface to these resources in the way that MPI specifies the interface to pass messages on an interconnect. Rather, the scope of the *Resource Manager Library* is to configure and coordinate the resources and then let the HPC software interface with the resource directly. The coordination of these resources must occur regardless of the existence of the *Resource Manager Library*. Having the *Resource Manager Library* means that the challenges of coordinating these resources are solved once and applied across our HPC software.

We envision execution with the *Resource Manager Library* to work roughly as follows.

1. The job schedule command (e.g. qsub or salloc) specifies the group of resources to allocate (in terms of nodes) along with optional details for accounts, queues, constraints, etc. This is roughly the same as today.
2. After the job is scheduled, the launch command (e.g. jobrun or srun) does not need any arguments. Not even the number of processes to launch. By default, the implementation launches one process per node (which is not necessarily the amount the application will eventually see).
3. When the software launches, it does not directly call MPI_Init. Instead, it interfaces with the *Resource Manager Library*. The application provides directives to the *Resource Manager Library* about how resources should be allocated such as provide one process per NUMA domain or attache one process to each CUDA device. The application then calls an initialize function in the *Resource Manager Library*.
4. The *Resource Manager Library*'s initialize function forks (or kills) processes to match the desired resource allocations. It then calls MPI_Init to initialize the MPI layer. That way when the *Resource Manager Library*'s initialize function returns, the application's state is equivalent to if the user specified the resource management through the job scheduler's arguments. The *Resource Manager Library* also sets the state of the other resource libraries to use the allocated resources. For example, it would set up threading interfaces like OpenMP, TBB, and pthreads to use the prescribed number of threads on some set of cores. It could also tell CUDA to default to a particular device.

5 Research Challenges

The work discussed in this paper has a heavy development focus to it. Nevertheless, there are several research challenges to be addressed including the following.

- The operation of the hypothetical *Resource Manager Library* is predicated on the application's ability to specify the resources it needs. What are the most effective means of specifying resource requirements? For example, is it best for a multithreaded application to specify preferred thread group size or preferred localization?
- What is the best way to ensure that resources are shared as expected? Is it possible to enforce resource allotments through protected modes or other means?
- Is it possible (or desirable) to change resource allocations within an application. For example, if an application is using two libraries where one desires many threads per process and the other does not use threads and desires the maximum amount of MPI processes, is it possible to switch back and forth between the two?
- Is this resource management best configured as its own layer with MPI, or is it more efficient to manage at a higher level as part of an AMT system like Legion [5] or DHARMA [6]. How does an AMT system manage the added complexity of heterogeneous systems? And could such a system still support applications using a message passing approach?

6 Conclusion

There is no consistent management of heterogeneous resources in HPC systems. Rather, the management of different types of resources have grown independently by the hardware and software developers of these resources. As the amount of heterogeneity in our HPC system grows, they quickly become unmanageable. We identify a need for a centralized system that can work with both job scheduler and application software.

7 Acknowledgments

Thanks to W. Alan Scott and Vitus Leung from Sandia National Laboratories and to Berk Geveci from Kitware, Inc.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] Slurm workload manager: Generic resource (GRES) scheduling. `https://slurm.schedmd.com/gres.html`, Nov. 2017.

[2] Torque resource manager: Admistrator guide 6.1.1. `http://docs.adaptivecomputing.com/torque/6-1-1/adminGuide/torqueAdminGuide-6.1.1.pdf`, Mar. 2017.

[3] Trinity: Advanced technology system. `http://www.lanl.gov/projects/trinity/`, Nov. 2017.

[4] J. Ahrens, B. Geveci, and C. Law. ParaView: An end-user tool for large data visualization. In *Visualization Handbook*. Elesvier, 2005. ISBN 978-0123875822.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Supercomputing*, November 2012.

[6] J. Bennett, R. Clay, et al. ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms. Technical Report SAND2015-8312, Sandia Nat'l Labs, 2015.