Hybrid Model Parallel Programs

Henry Neeman and Charlie Peck
Blue Waters/UPEP Institute @ NCSA, May, 2011

Well, How Did We Get Here?

Almost all of the clusters provisioned now and for the forseeable future are constellations, that is they are composed of nodes, each with 1 or more sockets holding CPUs with 2 or more cores, possibly with GPGPUs, connected by a high–speed network fabric.

With their additional levels of memory hierarchy these constellation clusters are more difficult to use efficiently than their predecessors, yet they offer the promise of significantly greater cycles available for science. The Blue Waters computational resource is a good example of this trend.

Often a hybrid approach, utilizing 2 or more of MPI, OpenMP, pthreads and/or GPGPU techniques, can utilize these computational resource more effeciently and effectively than any one of them on their own.

Common Parallel Programming Paradigms, Strengths and Weaknesses

- OpenMP

  1. Strengths - relatively easy to use, portable, relatively easy to adapt to existing serial software, standard standard, low latency/high bandwidth communication, implicit communication model, dynamic load balancing
  2. Weaknesses - shared memory model only, *i.e.* one "node" or memory region, support for C/C++ and FORTRAN only, little explicit control over thread creation and rundown

- Message Passing Interface (MPI)

  1. Strengths - support for C/C++, FORTRAN, Perl, Python and other languages, scales beyond one shared memory image "node", widely used in the scientific software community, enables you to harness more compute cycles and more memory which translate to bigger science
  2. Weaknesses - non–standard standard (to some extent), can support shared memory model for intra–node communication but not all bindings do it efficiently, can be difficult to program, high latency/low bandwidth communication (compared to shared memory), explicit communication model, load balancing can be difficult

- GPGPU with CUDA or OpenCL

  1. Strengths - ability to harness significant computational cycles, becoming widely used in the scientific software community
  2. Weaknesses - can be difficult to program, non–portable proprietary language (CUDA), requires re–thinking many problems to take advantage of the high degree of parallel cardinality required to harness those cycles

- pthreads

  1. Strengths - better granularity of control over thread model than OpenMP, fairly portable, explicit control over thread creation and rundown
  2. Weaknesses - can be difficult to program, only somewhat portable

- Field Programmable Gate Array (FPGA)

  1. Strengths - encode any algorithm in hardware
  2. Weaknesses - can be difficult to program, not portable, expensive hardware

Hybrid Parallel Algorithms

- OpenMP + CUDA
- OpenMP + FPGA
- MPI + OpenMP
- MPI + CUDA
- MPI + FPGA
- MPI + pthreads
- MPI + OpenMP + CUDA
- MPI + pthreads + CUDA

Designing, Building and Debugging Hybrid Parallel Programs

- The basic approach is to find the most efficient way to do the work, whether it be OpenMP (CPU), GPGPU (CUDA or OpenCL), pthreads or a blend of them, and then the most efficient way to distribute the data and harvest the results via MPI.
- Study the communication pattern(s) to insure that the algorithm maps to the architecture.

  1. Embarrassingly parallel
  2. Loosely coupled through tightly coupled

- Look for opportunities to overlap computation and communication, this is a key attribute of efficient hybrid parallel programs.

- Don't design/implement an algorithm that requires more than one type of parallelism to be enabled for it to run. This will make testing and debugging *much* harder than it needs to be.
- Better to {OpenMP, CUDA/OpenCL, Pthread} your MPI code than the other way around. MPI is harder to architect, do that first, then work on the "on–node" parallelism.
- The simplest and least error prone software architecture is to use MPI calls only outside of any parallel regions {OpenMP, pthreads} and only allow the master thread to communicate between MPI processes (this is known as funneling). It's also possible to use MPI calls within parallel regions if you are using a thread–safe MPI binding (not all of them are).

- Debug by running on one node and testing the {OpenMP, GPGPU (CUDA or OpenCL), Pthreads}, then on 2–n nodes with just the MPI enabled to verify the data transfer.
- Take care to have no more than about one process/thread doing network communication contemporaneously, contention for the network port will quickly become a bottleneck unless there is a minimal amount of communication.
- Take care to have no more than about two processes/threads performing GPGPU calculations contemporaneously, contention for the I/O bandwidth to/from the CPU and the GPGPU card can become a bottleneck.

Simple MPI + OpenMP Example

```
#include <omp.h>
#include "mpi.h"
#include <stdio.h>

int main (int argc, char *argv[]) {
  int world_size, my_rank, thread_count, thread_provided;
  const int _NUM_THREADS = 4;

  omp_set_num_threads(_NUM_THREADS);

  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &thread_provided)
  printf("The MPI binding provided thread support of: %d\n", thread_pr
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  #pragma omp parallel reduction(+:thread_count)
  {
    thread_count = omp_get_num_threads();
  }

  printf("MPI Rank %d has reported %d\n", my_rank, thread_count);

  MPI_Finalize();
  return 0;
}
```

Questions?

Lab Exercises

- To compile MPI + OpenMP programs on Sooner add `-fopenmp` to your `mpicc` command line. The same recipe works on Al-salam.
- To run MPI + OpenMP programs on Sooner copy and modify the example `bsub` script found here:
  `~charliep/NCSI2010/hybrid-mpi-openmp/example_parallel_hybrid.bsub`
- Copy, build, run and explore
  `~charliep/NCSI2010/hybrid-mpi-openmp/mpi-openmp-first.c`
- Copy, build, run and explore
  `~charliep/NCSI2010/hybrid-mpi-openmp/mpi-openmp-second.c`
  Fix the total threads problem.
- Copy, add the appropriate OpenMP directives, build, run and explore
  `~charliep/NCSI2010/hybrid-mpi-openmp/area_under_curve_mpi.c`
  Using the input file found in the same directory explore the efficiency of using more or less MPI processes and more or less OpenMP threads spread across different numbers of nodes for that problem size. What's the most efficient mapping?