Assignment 3:
MPI Point-to-Point and One-Sided Communication

Isaías A. Comprés Ureña
compresu@in.tum.de

Prof. Michael Gerndt
gerndt@in.tum.de

04-12-2015

1 Introduction

In the first assignment we focused on optimization through the use of compiler flags and pragmas, without modifying the source code in any significant way. In this and the final assignments we will work on optimization by modifying the parallel program. In this case, we will be optimizing the use of MPI point-to-point communication. In that sense, this will be the first assignment focused mainly in programming.

Point-to-point communication performance is still of great importance in MPI applications. There are still many applications that are entirely implemented with this type of communication, where peers in a communicator communicate directly. There are two models of direct peer-to-peer communication available in MPI: point-to-point and one-sided communication.

The point-to-point API consists of mainly send and receive operations. These operations have variants that are blocking, non-blocking, synchronous, ready and buffered. There are combined send and receive operations for swaps between peers. Additionally, there are wait and probe operations to check on the status of ongoing communication.

The one-sided API contains mainly put and get operations. These operations are all non-blocking with explicit transfers and synchronization. The processes need to create windows of memory that are then accessible by others with the put and get operations. In addition to these, there are also a collection of synchronization operations, such as locks and fences.

In this assignment, students will be provided a parallel MPI application that relies on blocking communication. The students will then need to transform it to use non-blocking point-to-point and one-sided communication. In both of these main tasks, the aim is to improve performance by improving MPI communication and allowing overlap of computation and communication.

1.1 Submission Instructions

Your third assignment submission for Programming of Supercomputers will consist of 2 parts:

- A 5 to 10 minute video with the required comments described in each task.
- A compressed tar archive with the required files described in each task.

2 Cannon's Matrix-Matrix Multiply Algorithm

The matrix-matrix multiplication operation in linear algebra has a large degree of parallelism. This fact has lead to several parallel algorithms with different advantages and disadvantages. In this assignment, we will focus on Cannon's algorithm (named after of Lynn Elliot Cannon). This algorithm is well posed for implementations in distributed memory systems.

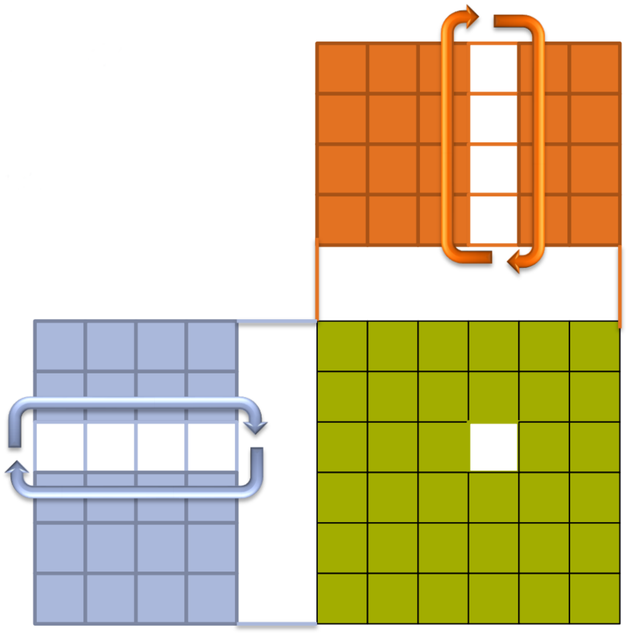

Figure 1: Cannon's Matrix-Matrix multiplication

The algorithm is based on a Cartesian grid, where both parts of the input and output matrices reside. The grid size is determined by the number of processes and after partitioning one block is given to each one.

The core idea of the algorithm (as illustrated in figure 1) is the rotational exchange of blocks from the input matrices. All matrices are divided in the same number of parts and distributed equally to all processes. The processes compute the partial solutions of the multiplication locally and rotate their input matrix parts with neighbors vertically and horizontally. After a number of algorithm steps that equals the square root of the total of processes, each local part of the solution matrix is completed.

2.1 Provided MPI Implementation

You will be provided an implementation of the algorithm. This implementation relies on MPI blocking point-to-point communication and can therefore be run in distributed memory systems. The implementation makes use of MPI's Cartesian virtual topology support.

The provided implementation takes as parameters the input matrices and an optional test flag. Input matrix files are provided with the assignments' materials. The test flag instructs the application to verify the correctness of the parallel algorithm implementation versus a serial execution; therefore, it is recommended that this test is only performed on the smaller multiplications, due to the required time for serial verification.

A Load-Leveler batch script is provided together with the implementation:

```
#!/bin/bash
#@ wall_clock_limit = 00:05:00
#@ job_name = pos-cannon-mpi-ibm
#@ job_type = Parallel
#@ output = cannon_64_$(jobid).out
#@ error = cannon_64_$(jobid).out
#@ class = test
#@ node = 4
#@ total_tasks = 64
#@ node_usage = not_shared
#@ energy_policy_tag = cannon
#@ minimize_time_to_solution = yes
#@ notification = never
#@ island_count = 1
#@ queue

. /etc/profile
. /etc/profile.d/modules.sh

date
mpiexec -n 64 ./cannon 64x64-1.in 64x64-2.in test
date
mpiexec -n 64 ./cannon 128x128-1.in 128x128-2.in test
date
mpiexec -n 64 ./cannon 256x256-1.in 256x256-2.in test
date
mpiexec -n 64 ./cannon 512x512-1.in 512x512-2.in
date
mpiexec -n 64 ./cannon 1024x1024-1.in 1024x1024-2.in
date
mpiexec -n 64 ./cannon 2048x2048-1.in 2048x2048-2.in
date
mpiexec -n 64 ./cannon 4096x4096-1.in 4096x4096-2.in
date
```

The batch script is configured to run the benchmark as required by this assignment: all of the tests will be performed with exactly 4 nodes and 64 MPI processes with the specific input files and options. The 'test' option passed to the 64x64, 128x128 and 256x256 cases tells the application to make a consistency check. When performing the consistency check, the input matrices and output matrix are printed to the screen. It is recommended that the students keep this option enabled during development and testing, to ensure that their MPI optimizations do not affect the correctness of the multiplication.

3 Provided Implementation and Baseline

During optimization activities, it is important to understand the problem that is being optimized and to measure a performance baseline. Take a look at the provided Load-Leveler script and modify it so that good measurements can be collected (hint: think of variance between runs) to measure an accurate application's baseline performance. Please note that the application already reports compute and MPI times. After that, perform the following steps:

1. Pick a target SuperMUC partition (Sandy Bridge or Haswell).
2. Make sure that you load the default IBM MPI and Intel compiler modules.
3. Build the provided implementation of Cannon's algorithm.
4. Run the updated Load-Leveler batch script.
5. Collect and plot the measured compute and MPI times.

Make sure to perform these tasks in both types of nodes (Haswell and Sandy Bridge). We will be evaluating performance on both types of nodes and both baselines need to be determined. This can be controlled by logging in to the relevant login nodes:

```
sb.supermuc.lrz.de
hw.supermuc.lrz.de
```

Please refer to the LRZ's documentation under:
`https://www.lrz.de/services/compute/supermuc/access_and_login/`

3.1 Required Submission Files

Submit the updated Load-Leveler batch script and the compute and MPI time plots. Provide the plots in PDF format.

3.2 Required Video Commentary

Please discuss the following topics in the video:

1. Cannon's algorithm and the provided implementation.
2. Challenges in getting an accurate baseline and changes to the Load-Leveler batch script.
3. Compute time scalability with fixed 64 processes and varying size of input files.
4. MPI time scalability with fixed 64 processes and varying size of input files.
5. Differences in scalability between the Sandy Bridge and Haswell architectures.

4 MPI Point-to-Point Communication

After understand the performance and scalability of the provided implementation and measuring a baseline for performance, we are now ready to optimize the application. In this task, the optimization will be based on converting blocking to non-blocking point-to-point communication. The aim is to reduce communication time as well as allow for the overlap of computation and communication.

Perform the following activities as part of this task:

1. Study the set of non-blocking operations in MPI's point-to-point API.
2. Select which operations to use in the benchmark.
3. Identify the main computational loop of the application.
4. Convert the communication part of this loop with the selected operations.
5. Analise the optimized version by following the steps in 3
6. Generate new plots with the new optimized binary.

4.1 Required Submission Files

Submit the updated `cannon.c` file and new performance plots.

4.2 Required Video Commentary

Answer the following questions in the video:

- Which non-blocking operations were used?
- What is the theoretical maximum overlap that can be achieved? Explain.
- Was communication and computation overlap achieved?
- Was a speedup observed versus the baseline?
- Were there any differences between Sandy Bridge and Haswell nodes?

5 MPI One-Sided Communication

We continue our optimization efforts now with a conversion from blocking point-to-point to one-sided communication. The aim is again to reduce communication time and to allow for computation and communication overlap. Additionally, one sided communication should reduce synchronization costs and intermediate buffering.

Perform the following activities as part of this task:

1. Study the set of one-sided operations in MPI.
2. Select which operations to use in the benchmark.
3. Identify the main computational loop of the application.
4. Convert the communication part of this loop with the selected operations.
5. Analise the optimized version by following the steps in 3
6. Generate new plots with the new optimized binary.

5.1 Required Submission Files

Submit the updated `cannon.c` file and new performance plots.

5.2 Required Video Commentary

Answer the following questions in the video:

- Which one-sided operations were used?
- Was communication and computation overlap achieved?
- Was a speedup observed versus the baseline?
- Was a speedup observed versus the non-blocking version?
- Were there any differences between Sandy Bridge and Haswell nodes?