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

Madhura Kumaraswamy
kumarasw@in.tum.de

Prof. Michael Gerndt
gerndt@in.tum.de

30-11-2018

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 the third and the fourth 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.

Point-to-point communication performance is 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 exchange data 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 is 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 assignment submission for Programming of Supercomputers will consist of 2 parts:

1. A 5 to 15 page report with the required answers.
    - Submit the report in PDF format.
    - Plots and figures should be used to enhance any explanations.
    - Provide PRECISE answers for each task in the same order as the questions.
    - The report must contain the contribution of each group member to the assignment (max. 2 sentences/member).
2. A compressed tar archive with the required files described in each task.

2 Gaussian Elimination

Gaussian Elimination is a widely used technique used to solve systems of equations. To solve a linear system, transformations are applied so that the system matrix is transformed into an upper triangular form. After the matrix is in this form, back substitution is performed to find the solution vector.

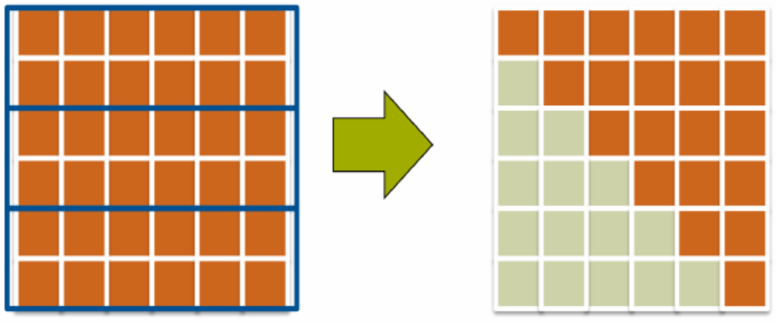

Figure 1: Upper triangular transformation with Gaussian Elimination

An implementation is provided where the work is divided in equal parts among processes. This requires that the square matrix's size is divisible by the number of MPI processes evenly.

Figure 1 illustrates a 6x6 matrix that is partitioned across 3 processes, and then transformed into upper triangular for back substitution.

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 provided implementation takes as parameters the base name for the matrix and vector inputs. The source files as well as the input data for the algorithm are provided on Moodle.

A Load-Leveler batch script called job.ll is provided together with the implementation:

```
#!/bin/bash
#@ wall_clock_limit = 00:30:00
#@ job_name = pos-gauss-mpi-intel
#@ job_type = MPICH
#@ output = out_gauss_64_intel_$(jobid).out
#@ error = out_gauss_64_intel_$(jobid).out
#@ class = test
#@ node = 4
#@ total_tasks = 64
#@ node_usage = not_shared
#@ energy_policy_tag = gauss
#@ minimize_time_to_solution = yes
#@ notification = never
#@ island_count = 1
#@ queue

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

module unload mpi.ibm
module load mpi.intel

#Uncomment the following lines for tracing
#module load scorep
#ulimit -c unlimited
```

```
#export SCOREP_TIMER="clock_gettime"
#export SCOREP_ENABLE_PROFILING=false
#export SCOREP_ENABLE_TRACING=true
#export SCOREP_TOTAL_MEMORY=2GB

date
module list

echo "Tests starting..."
echo

mpiexec -n 8 ./gauss ./gs_data/size64x64
date
mpiexec -n 16 ./gauss ./gs_data/size64x64
date
mpiexec -n 32 ./gauss ./gs_data/size64x64
date
...
```

In this assignment the tests will be performed with a maximum of 4 nodes and 64 MPI processes with the input files for sizes: 64x64, 512x512, 1024x1024, 2048x2048, 4096x4096 and 8192x8192. Make sure to verify this in the job script.

Instructions for generating traces:

1. Load the scorep and vampir/9.0 modules.
2. Build the application using `make clean && make -f makefile_scorep`.
3. Uncomment the lines after `#Uncomment the following lines for tracing` in the jobscript. Note that there is no scorep.filt file for this assignment.
4. Don't generate traces for every combination of processes and problem size in the jobscript. Select 2-3 from the entire set, and comment out the other lines.
5. Submit the job.
6. Use Vampir for visualization.

3 Setting a Baseline *(26 points)*

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: address the variance between runs) to determine an accurate application's performance baseline. Please note that the application already reports compute and MPI times for each participating process.

For this assignment, we will be working on the Phase 1 thin (Sandy Bridge) nodes, and the Phase 2 (Haswell) nodes with the Intel MPI library and Intel compilers. Make sure that you have the correct modules at all times and that you are logged into the correct login nodes:

```
Sandy Bridge: <user_id>@sb.supermuc.lrz.de
Haswell: <user_id>@hw.supermuc.lrz.de
```

The output of `module list` should show:

```
Currently Loaded Modulefiles:
1) admin/1.0 2) tempdir/1.0 3) lrztools/1.0 4) intel/17.0 5) mkl/2017 6) poe/1.4
  7) lrz/default 8) mpi.intel/2017
```

Instructions:

1. Pick a target SuperMUC partition (Haswell or Sandy Bridge).
2. Copy the provided materials from moodle to your SuperMUC account.
3. Make sure that you load the Intel MPI and Intel compiler modules.
4. Build the provided Gaussian Elimination implementation using `make clean && make`.
5. Run the Load-Leveler batch script.
6. Modify the batch script to address the variance between the runs.
7. Run the updated Load-Leveler batch script.
8. Collect and plot the measured IO, setup, compute, MPI and total times.
9. Repeat steps 3-5 and 7-8 on the other SuperMUC partition.

3.1 Required submission files

1. The updated Load-Leveler batch script. (3 points)
2. The performance plots and description in the report. (4 points)

3.2 Questions

1. Briefly describe the Gaussian Elimination and the provided implementation. (3 points)
2. How is data distributed among the processes? (2 points)
3. Explain the changes applied to the provided Load-Leveler batch script. (3 points)
4. What were the challenges in getting an accurate baseline time for Gaussian Elimination. (3 points)
5. Describe the compute and MPI times scalability with fixed process counts and varying size of input files for the Sandy Bridge and Haswell nodes. Did you observe any differences? (4 points)
6. Describe the compute and MPI times scalability with fixed input sets and varying process counts for the Sandy Bridge and Haswell nodes. Did you observe any differences? (4 points)

4 MPI Point-to-Point Communication *(33 points)*

After understanding 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.

Instructions:

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. Convert the communication parts of the application.
4. Analyze the optimized version by following steps 3-5 and 8-9 from Section 3 for both Sandy Bridge and Haswell nodes.
5. Generate new plots with the new optimized binary.

4.1 Required Submission Files

1. The updated `gauss.c` file. (12 points)
2. The new performance plots and the description in the report. (5 points)

4.2 Questions

1. Which non-blocking operations were used? Justify your choice. (6 points)
2. Was communication and computation overlap achieved? Use Vampir. (5 points)
3. Was a speedup observed versus the baseline for the Sandy Bridge and Haswell nodes? (5 points)

5 MPI One-Sided Communication *(41 points)*

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, if possible, to allow for computation and communication overlap. Additionally, one sided communication should in theory reduce synchronization costs and intermediate buffering.

Instructions:

1. Study the set of one-sided operations in MPI.
2. Select which operations to use in the benchmark.
3. Convert the communication parts of the application.
4. Analyze the optimized version by following steps 3-5 and 8-9 from Section 3 for Sandy Bridge and Haswell nodes.
5. Generate new plots with the new optimized binary.

5.1 Required Submission Files

1. The updated `gauss.c` file. (15 points)
2. The new performance plots and the description in the report. (5 points)

5.2 Questions

1. Which one-sided operations were used? Justify your choice. (6 points)
2. Was communication and computation overlap achieved? Use Vampir. (5 points)
3. Was a speedup observed versus the baseline for the Sandy Bridge and Haswell nodes? (5 points)
4. Was a speedup observed versus the non-blocking version for the Sandy Bridge and Haswell nodes? (5 points)