Programming of Super Computers

Assignment 1: Single Node Performance

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

Prof. Michael Gerndt
gerndt@in.tum.de

23-10-2015

1 Introduction

Systems that reach hundreds of TeraFLOPS to multiple PetaFLOPS of sustained computing performance are considered supercomputers today. These levels of performance are achieved through the aggregation of performance of several computing nodes interconnected by a high-performance network. The performance of each individual node is therefore very important for the overall performance of any application.

Current computing nodes are parallel computers with several cores and a hierarchical memory system. Several cores are integrated into a processor. Multiple of these processors are configured in a single node's main board through sockets. A typical node in today's supercomputers have between 2 and 4 sockets, with each socket connected to a memory bank. Because of this, variable memory latencies and bandwidth are measured at individual cores. These systems are classified as Non Unified Memory Access (NUMA) systems. These NUMA systems, with variable memory performance and large amounts of available parallelism, pose great optimization challenges today.

In this assignment, students will learn how to optimize applications for absolute performance and scalability in a single node. This first optimization effort will later benefit the performance of applications when scaling them to multiple nodes.

2 Submission Instructions and General Information

Your 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.

This section gives recommendations for the creation of the video, as well as links to general resources from the Leibniz Supercomputing Centre (LRZ) and other sources.

2.1 Desktop Capture and Video Mixing Software

Students are allowed to use any video capture, sound capture and mixing software available to them. In this section, we recommend a free software combination of tools available on most GNU/Linux distributions.

2.1.1 Kazam Screen Capture

Kazam is a popular screen capture program available in Debian's package manager (as well as derived distributions such as Ubuntu). To install it in such a distribution, simply issue the following command:

```
apt-get install kazam
```

After that, the tool can be started by running 'kazam' from the terminal.

Kazam can capture a whole screen, an application window or a screen selection through its GUI. The students are encouraged to look at the many video guides and tutorials available online for this tool.

2.1.2 OpenShot Video Editor

After recording small clips, related to each of the tasks described in this assignment, the student is ready to assemble them in the final video for submission. For this activity, we recommend the use of OpenShot. It is a tool for mixing video and audio that is simple to use and also available directly through Debian's package manager (as well as derived distributions such as Ubuntu). To install it, simply issue the following command:

```
apt-get install openshot
```

After that, the tool can be started by running 'openshot' from the terminal.

OpenShot is a more complex piece of software when compared to Kazam, but for attaching video snippets it is very simple. Again, the student is encouraged to look at the many written and video tutorials available.

2.1.3 VideoLAN Player

After completing the video, this popular video player should be used to test the final result. The VideoLAN player is available in multiple platforms, such as Windows, MacOS and GNU/Linux.

To install it on Debian and derived systems, simply issue the following command as root:

```
apt-get install vlc
```

After that, the video player is available from the terminal by running the 'vlc' command.

Note that there are no video or audio format restrictions for this assignment. The only restriction is that the video must be playable with recent versions of this video player; therefore, it is recommended that the students view the video at least once with this player before submitting it.

2.1.4 Resources at the Leibniz Supercomputing Centre (LRZ)

The Leibniz Supercomputing Centre (LRZ) provides documentation about the tools that are offered in the SuperMUC petascale system. Make sure to check their resources. In particular, we would like to bring the following ones to the student's attention:

- IBM's Load-Leveler documentation:
  `https://www.lrz.de/services/compute/supermuc/loadleveler/`
- GNU Compiler Collection:
  `https://www.lrz.de/services/software/programmierung/gcc/`
- Intel Compilers:
  `https://www.lrz.de/services/software/programmierung/intel_compilers/`

3 LULESH Benchmark

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) benchmark is an application that is widely used to evaluate performance. It is commonly referred to as a proxy application since it tries to model the computation and communication patterns common in a domain of computational science. LULESH is a proxy application for the shock hydrodynamics problem. For detailed information about LULESH, please refer to its official page at: `https://codesign.llnl.gov/lulesh.php`.

To prepare for the tasks of this assignment, download and extract version 2.0.3 of the benchmark for CPUs (the one that supports OMP, MPI, and MPI+OMP, and not the one for GPUs). After that, spend a few minutes reading the README file; it contains information about what is implemented in each file as well as some general comments on the benchmark's current development status.

3.1 Building and Running in Single-Threaded Mode

To build the benchmark in single-threaded mode, edit the Makefile and make sure to update the following variables: MPI_INC, MPI_LIB, SERCXX, MPICXX, CXXFLAGS and LDFLAGS. These should match your environment, be it SuperMUC or your personal computer.

To enable the single-threaded (serial) version of the benchmark, modify line 21 of the Makefile so that the serial command is set:

```
CXX = $(SERCXX)
```

The Makefile contains examples for OpenMP and serial versions for CXXFLAGS and LDFLAGS between lines 23 and 29:

```
#Default build suggestions with OpenMP for g++
CXXFLAGS = -g -O3 -fopenmp -I. -Wall
LDFLAGS = -g -O3 -fopenmp

#Below are reasonable default flags for a serial build
#CXXFLAGS = -g -O3 -I. -Wall
#LDFLAGS = -g -O3
```

Since this version of the benchmark includes the MPI, OMP and serial versions together, you will need to update these variables depending on the specific task in this assignment. For all the tasks in sections 4, make sure to enable the serial build without OpenMP. You may also want to disable #pragma warnings in this build. For the serial version of the benchmak, your CXXFLAGS and LDFLAGS should look like this:

```
# flags with OpenMP and #pragma warnings disabled
CXXFLAGS = -O3 -I. -w
LDFLAGS = -O3
```

At this point, you are ready to build the benchmark. Issue a 'make' command and the output should look as follows:

```
Building lulesh.cc
g++ -DUSE_MPI=0 -c -O3 -I. -w -o lulesh.o lulesh.cc
Building lulesh-comm.cc
g++ -DUSE_MPI=0 -c -O3 -I. -w -o lulesh-comm.o lulesh-comm.cc
Building lulesh-viz.cc
g++ -DUSE_MPI=0 -c -O3 -I. -w -o lulesh-viz.o lulesh-viz.cc
Building lulesh-util.cc
g++ -DUSE_MPI=0 -c -O3 -I. -w -o lulesh-util.o lulesh-util.cc
Building lulesh-init.cc
g++ -DUSE_MPI=0 -c -O3 -I. -w -o lulesh-init.o lulesh-init.cc
Linking
g++ -DUSE_MPI=0 lulesh.o lulesh-comm.o lulesh-viz.o lulesh-util.o lulesh-init.o -O3 -lm -o lulesh2.0
```

After building, the file 'lulesh2.0' in the benchmark's binary. If you make any changes to the Makefile, remember to issue a 'make clean' first, to ensure that the benchmark is rebuilt (since 'make' only detects changes to source files). Run it by issuing './lulesh2.0' in the same directory. Here is some example output:

```
Running problem size 30^3 per domain until completion
Num processors: 1
Total number of elements: 27000
...
Run completed:
```

```
  Problem size   = 30
  MPI tasks      = 1
  Iteration count = 932
  Final Origin Energy = 2.025075e+05
  Testing Plane 0 of Energy Array on rank 0:
      MaxAbsDiff = 7.639755e-11
      TotalAbsDiff = 8.590535e-10
      MaxRelDiff = 1.482369e-12


Elapsed time    =    26.72 (s)
Grind time (us/z/c) = 1.0619293 (per dom) ( 1.0619293 overall)
FOM             = 941.68231 (z/s)
```

3.2 Building and Running in Multi-Threaded Mode

For tasks in section 5 you will need to enable support for OpenMP, MPI, or both in the LULESH benchmark. The following sections will describe the necessary changes to the Makefile, as well as how to run the benchmark in each case.

3.2.1 OpenMP

To build the benchmark with OpenMP, add the OpenMP compiler flag in the CXXFLAGS and LDFLAGS variables (as shown in the example included in the original Makefile bellow line 23). When enabling OpenMP, make sure to check that you set the correct flag depending on your compiler (usually -fopenmp with GNU compilers and -openmp with Intel compilers). For the Intel compilers, your flags should look as follows:

```
CXXFLAGS = -O3 -openmp -I.
LDFLAGS = -O3 -openmp
```

That is the only necessary change in the Makefile. Make sure you issue a 'make clean' followed by a 'make' to build the new binary.

The following Load-Leveler batch script can be used to launch the benchmak in SuperMUC with OpenMP:

```
#!/bin/bash
#@ wall_clock_limit = 00:20:00
#@ job_name = pos-lulesh-openmp
#@ job_type = MPICH
#@ class = micro
#@ output = pos_lulesh_openmp_$(jobid).out
#@ error = pos_lulesh_openmp_$(jobid).out
#@ node = 1
#@ total_tasks = 16
#@ node_usage = not_shared
#@ energy_policy_tag = lulesh
#@ minimize_time_to_solution = yes
#@ island_count = 1
#@ queue

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

export OMP_NUM_THREADS=16
./lulesh2.0
```

3.2.2 MPI

To build the benchmark with MPI enabled, modify the CXX variable in line 13 of the Makefile to use the MPICXX command:

```
CXX = $(MPICXX)
```

For the MPI only version of the benchmark, make sure that you do not enable OpenMP in your CXXFLAGS and LDFLAGS variables. They should look as follows:

```
CXXFLAGS = -O3 -I. -w
LDFLAGS = -O3
```

Additionally, make sure that your MPICXX command includes the correct MPI compiler wrapper. In this assignment we will use the Intel MPI wrapper 'mpiCC'. First, make sure that the correct MPI module is loaded:

```
module unload mpi.ibm
module load mpi.intel
```

Afterwards, update the variable in line 12 as follows:

```
MPICXX = mpiCC -DUSE_MPI=1
```

Again, make sure you issue a 'make clean' followed by a 'make' to build the new binary. To run the benchmark with MPI in SuperMUC, you can reuse the following Load-Leveler batch script:

```
#!/bin/bash
#@ wall_clock_limit = 00:20:00
#@ job_name = pos-lulesh-mpi
#@ job_type = MPICH
#@ class = micro
#@ output = pos_lulesh_mpi_$(jobid).out
#@ error = pos_lulesh_mpi_$(jobid).out
#@ node = 1
#@ total_tasks = 16
#@ node_usage = not_shared
#@ energy_policy_tag = lulesh
#@ minimize_time_to_solution = yes
#@ island_count = 1
#@ queue

. /etc/profile
. /etc/profile.d/modules.sh
. $HOME/.bashrc

# load the correct MPI library
module unload mpi.ibm
module load mpi.intel

mpiexec -n 8 ./lulesh2.0
```

Note that we are only requesting the creation of 8 processes. The reason is that LULESH only accepts cubes of integers (i.e. That means, 1, 8, 27, etc.) as its process count and SuperMUC thin nodes have 16 cores total.

3.2.3 MPI + OpenMP

The benchmark can be run with both MPI and OpenMP enabled. This combination is usually referred to as 'hybrid'. To achieve this configuration, we simply enable both of these in the Makefile. First, select the MPI command in the CXX variable:

```
CXX = $(MPICXX)
```

After that, make sure that OpenMP is enabled in your CXXFLAGS and LDFLAGS variables. They should look as follows:

```
CXXFLAGS = -O3 -openmp -I.
LDFLAGS = -O3 -openmp
```

Finally, verify that your MPICXX variable has the correct MPI compiler wrapper:

```
MPICXX = mpiCC -DUSE_MPI=1
```

As always, make sure you issue a 'make clean' followed by a 'make' to build the new binary. To run the benchmark with both MPI and OpenMP in SuperMUC, you can reuse the following Load-Leveler batch script:

```
#!/bin/bash
#@ wall_clock_limit = 00:20:00
#@ job_name = pos-lulesh-hybrid
#@ job_type = MPICH
#@ class = micro
#@ output = pos_lulesh_hybrid_$(jobid).out
#@ error = pos_lulesh_hybrid_$(jobid).out
#@ node = 1
#@ total_tasks = 16
#@ node_usage = not_shared
#@ energy_policy_tag = lulesh
#@ minimize_time_to_solution = yes
#@ island_count = 1
#@ queue

. /etc/profile
. /etc/profile.d/modules.sh
. $HOME/.bashrc

# load the correct MPI library
module unload mpi.ibm
module load mpi.intel

export OMP_NUM_THREADS=2
mpiexec -n 8 ./lulesh2.0
```

Since SuperMUC thin nodes have 16 cores total, we can only run the hybrid version of the benchmark with 1 process and 16 threads, or 8 processes and 2 threads per process.

4 Single-Thread Performance

As mentioned in the introduction1, the performance of each node affects the overall performance of an application in a supercomputer. Similarly, the performance of each thread will affect the overall performance of an application in a node. Because of this, in this section of the assignment students will optimize the performance of a serial application. This application will later be parallelized using threading and multiple processes in Section 5.

4.1 GNU Profiler

Before we start optimizing the benchmark, we need to find out which routines are actually worth optimizing. For that, we can use the GPROF (the GNU profiler). You can find a GPROF guide at the Leibniz Supercomputing Centre (LRZ) under: `https://www.lrz.de/services/compute/supermuc/tuning/gprof/`

To prepare your benchmark, go back to the Makefile and add the flags '-pg' to the compiler flags. After that, make sure that you issue a 'make clean' and the 'make' again to build the new binary. Once the binary is updated, simply run the benchmark again. This will generate a file called 'gmon.out' that can be inspected with the 'gprof' command as follows (where <binary-name> is the name of your benchmark's binary):

```
gprof <binary-name> gmon.out
```

The 'gprof' command will output a report that is divided in two sections: the flat profile and the call graph. The flat profile shows a list of functions, ordered by aggregated time spent. The call graph presents all sequences of calls performed in the program. Look at the flat profile and take note of the functions that together took 80% or more of the total execution time of the program.

4.1.1 Required Submission Files

Redirect the output of 'gprof' to a file, as follows:

```
gprof <binary-name> gmon.out > assignment1.gprof.out
```

Submit the 'assignment1.gprof.out' file.

4.1.2 Required Video Commentary

In the video, make sure to mention which functions represented most of the execution time of the benchmark. Enumerate the functions that make together 80% or more of the execution time.

4.2 Compiler Flags

In this task, the students will investigate the performance effect of available optimization flags with the GNU and Intel compilers. Compilers provide general optimization levels that range typically between level 1 and 4. In addition to general optimization levels, there are several other more specific optimization flags available.

The current default version of GCC in the SuperMUC system is '4.9'. For this version of GNU compilers, set the general optimization level to 3 (with -O3) and evaluate combinations of the following flags:

- "-march=native"
- "-fomit-frame-pointer"
- "-floop-block"
- "-floop-interchange"
- "-floop-strip-mine"
- "-funroll-loops"
- "-flto"

The current default version of Intel compilers in the SuperMUC system is '15.0'. For this version the Intel compilers, set the general optimization level to 3 (with -O3) and evaluate combinations of the following flags:

- "-march=native"
- "-xHost"
- "-unroll"
- "-ipo"

Before you start evaluating the benchmark with new flags, make sure you identify what is the performance metric reported by the application and make a test run (make sure to use a binary built with -O3 only). Record the result and use it for speed-up computations in the following steps. Prepare a spreadsheet with either Microsoft Office, Open Office or Google Docs.

For each of the combinations perform the following steps:

1. Update the compiler flags in the Makefile with the new combination.
2. Rebuild the benchmark by calling 'make clean && make'.
3. Run the benchmark in one SuperMUC node, Phase 1 or Phase 2 (test class).
4. Record the new value of the performance metric.
5. Compute the speed-up versus the baseline.
6. Store the new value of the performance metric and speed-up in the spreadsheet.

4.2.1 Required Submission Files

Submit the Makefile that contains the best combination of parameters (biggest speed-up).

4.2.2 Required Video Commentary

Answer the following questions in the video:

- Look at the compilers' help (by issuing 'icc –help' and 'gcc –help'). How many optimization flags are available for each compiler (approximately)?
- Given how much time it takes to evaluate a combination of compiler flags, would it be realistic to test all possible combinations of available compiler flags in these compilers?
- Which compiler and optimization flags combination produced the fastest binary?

4.3 Optimization Pragmas

After finding a good combination of compiler flags to use when compiling the benchmark described in section 3, you are now ready to look at more fine control when applying compiler optimizations. The compiler flags were applied to the whole binary. Location specific optimizations can be applied with #pragma directives.

For this task, first investigate the purpose of the following #pragma directives in GCC's documentation:

- #pragma GCC optimize
- #pragma GCC ivdep

Second, investigate the purpose of the following #pragma directives in Intel's documentation:

- #pragma simd
- #pragma ivdep
- #pragma loop_count
- #pragma vector
- #pragma inline
- #pragma noinline

- #pragma forceinline
- #pragma unroll
- #pragma nounroll
- #pragma unroll_and_jam
- #pragma nofusion
- #pragma distribute_point

Based on the results of the flat profile of the benchmark, collected in section 4.1, find the function where most of the time is taken. Go to that function, and find a location where to apply one of the loop optimization #pragma directives. Insert the #pragma into the source text and save the file.

4.3.1 Required Submission Files

Submit the modified source file with the added #pragma annotation.

4.3.2 Required Video Commentary

In the video, discuss the difference between Intel's 'simd', 'vector' and 'ivdep' #pragma directives. Additionally, justify your decision to apply the selected #pragma in the particular location on the submitted source file.

5 Multi-Thread Performance

After optimizing the provided application's single thread performance, now the students are ready to scale it and make use of the additional cores available in the node. Threading, multiple processes or a combination of both techniques will be explored in this part of the assignment.

5.1 OpenMP

To configure the benchmark with OpenMP enabled, please refer to section 3.2.1. Build the benchmark and launch it with the provided Load-Leveler script.

As part of this task, modify the variable OMP_NUM_THREADS to measure the scalability of the benchmark. Set the variable to 1, 2, 4, 8, and 16 threads. Record the performance metric of the benchmark and make a plot. To make the plot, use GNU Plot or generate and export a plot from a spreadsheet.

5.1.1 Required Submission Files

Submit the scalability plot in PDF format. Name the file: 'OpenMP_scalability.pdf'.

5.1.2 Required Video Commentary

Comment about the scalability observed:

- Was linear scalability achieved?
- On which thread-count was the maximum performance achieved?

5.2 MPI

To configure the benchmark with MPI enabled, please refer to section 3.2.2. Build the benchmark and launch it with the provided Load-Leveler script.

Take a look at the benchmark's documentation and evaluate scalability with valid process counts. The number of processes must not exceed the total of cores in the node used. Record the performance with each process count and generate a plot. Use GNU Plot or generate and export a plot from a spreadsheet.

5.2.1 Required Submission Files

Submit the scalability plot in PDF format. Name the file: 'MPI_scalability.pdf'.

5.2.2 Required Video Commentary

Comment about the scalability observed:

- What are the valid combinations of processes allowed?
- Was linear scalability achieved?
- On which process-count was the maximum performance achieved?
- How does the performance compare to the results achieved with OpenMP in section 5.1?

5.3 MPI + OpenMP

To configure the benchmark with MPI and OpenMP enabled, please refer to section 3.2.3. Build the benchmark and launch it with the provided Load-Leveler script.

Take a look at the benchmark's documentation and evaluate scalability with valid process and thread counts. The number of processes times the number of threads must not exceed the total of cores in the node used. Record the performance with each process and thread count and generate a plot. Use GNU Plot or generate and export a plot from a spreadsheet.

5.3.1 Required Submission Files

Submit the scalability plot in PDF format. Name the file: 'Hybrid_scalability.pdf'.

5.3.2 Required Video Commentary

Comment about the scalability observed:

- What are the valid combinations of processes and threads?
- Was linear scalability achieved?
- On which combination of processes and threads was the maximum performance achieved?
- How does the performance compare to the results achieved with OpenMP in section 5.1 and with MPI in section 5.2?
- Which solution is overall the fastest?
- Would you have guessed this best combination before performing the experiments in sections 5.1, 5.2 and 5.3?