Programming of Supercomputers

Assignment 1: Single Node Performance

Madhura Kumaraswamy
kumarasw@in.tum.de

Prof. Michael Gerndt
gerndt@in.tum.de

26-10-2018

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 by aggregating the 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. Current nodes usually have multiple processors, each plugged to a socket in its main board. Most nodes today have between 2 and 4 sockets, and each socket is connected to separate memory banks. In other words, these systems have Non-Unified Memory Access (NUMA). Because of this, variable memory latencies and bandwidth are measured at individual cores. 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 on a single node. The improvements provided by single node optimizations later multiply when scaling applications to multiple nodes.

2 Submission Instructions and General Information

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

1. A 5 to 10 page report with the required answers.
   - Submit the report in PDF format.
   - Provide a SHORT summary of the steps performed in each task (max. 6 lines).
   - 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.

In the remainder of this section, recommendations for the creation of the report, as well as links to general resources from the Leibniz Supercomputing Centre (LRZ) and other sources, will be given. These are only recommendations: the students are free to use other tools to produce the report, such as Microsoft or Adobe products. If a specific tool is required, such as a debugger or a compiler, this will be explicitly stated in the task description.

2.1 Document Preparation Software with PDF Support

Students are allowed to use any document preparation software to produce the report in PDF format. In this section, we recommend a free software combination of tools available on most GNU/Linux distributions.

2.1.1 LaTeX

LaTeXis a document preparation system for high-quality typesetting. It is the most widely used tool for the production of technical or scientific documents, such as those meant to be published at a journal or conference. To install it in Debian based distributions (such as Ubuntu), simply issue the following command:

```
apt-get install texlive-full
```

2.1.2 LibreOffice and OpenOffice Writer

Writer is an open-source cross-platform word processor that is distributed with the LibreOffice suite. LibreOffice was forked from OpenOffice. Both of these office suites can be used for the production of documents in PDF format. To install it in Debian based distributions (such as Ubuntu), simply issue the following command:

```
apt-get install libreoffice
```

2.1.3 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. The following links may be useful:

- IBM's Load-Leveler documentation:
  `https://www.lrz.de/services/compute/supermuc/loadleveler/`
- GNU Compiler Collection:
  `https://www.lrz.de/services/software/programmierung/gcc/`
- GNU Profiler:
  `https://www.lrz.de/services/compute/supermuc/tuning/gprof/`
- Intel Compilers:
  `https://www.lrz.de/services/software/programmierung/intel_compilers/`
- IBM MPI:
  `https://www.lrz.de/services/software/parallel/mpi/ibmmpi/`
- Intel MPI:
  `https://www.lrz.de/services/software/parallel/mpi/intelmpi/`

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. The README file 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 on SuperMUC or your personal computer. In SuperMUC, make sure to get familiar with the module system. For example, the command `module show <module>` can be used to inspect the locations of library and header components provided by a module.

The `Makefile` contains examples for `CXXFLAGS` and `LDFLAGS` for both OpenMP and serial versions for the GNU compiler:

```
#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
```

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

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

Note

- To build using the GNU compiler, set:

  ```
  SERCXX = g++ -DUSE_MPI=0
  ```

- To build using the Intel compiler, set:

  ```
  SERCXX = icpc -DUSE_MPI=0
  ```

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 Section 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 benchmark, 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. The output for the GNU compiler 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` is 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 = 4.001777e-11
TotalAbsDiff = 6.748414e-10
MaxRelDiff = 1.502434e-12


Elapsed time    =    36.68 (s)
Grind time (us/z/c) = 1.4576528 (per dom) ( 1.4576528 overall)
FOM             = 686.03441 (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 below.

```
CXXFLAGS = -O3 <openmp flag> -I.
LDFLAGS = -O3 <openmp flag>
```

When enabling OpenMP, make sure to check that you set the correct flag depending on your compiler (usually -fopenmp with GNU compilers and -qopenmp with Intel compilers). For the Intel compiler, your flags should look as follows:

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

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.

Run the benchmark on a SuperMUC compute node by submitting the following Load-Leveler batch script with OpenMP:

```
#!/bin/bash
#@ wall_clock_limit = 00:20:00
#@ job_name = pos-lulesh-openmp
#@ job_type = MPICH
#@ class = fat
#@ output = pos_lulesh_openmp_$(jobid).out
#@ error = pos_lulesh_openmp_$(jobid).out
#@ node = 1
#@ total_tasks = 40
#@ 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

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
```

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

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

Update the variable `MPICXX` as follows:

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

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
```

Again, make sure you issue a `make clean` followed by a `make` to build the new binary. To run the benchmark with MPI, 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 = fat
#@ output = pos_lulesh_mpi_$(jobid).out
#@ error = pos_lulesh_mpi_$(jobid).out
#@ node = 1
#@ total_tasks = 8
#@ node_usage = not_shared
#@ energy_policy_tag = lulesh
#@ minimize_time_to_solution = yes
#@ island_count = 1
#@ queue

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

# 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., 1, 8, 27, ...). This script needs to be updated based on the type of node used and its maximum core count. Remember to use mpiexec to run MPI and hybrid applications.

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, first make sure that the correct MPI module is loaded:

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

Next, simply enable both MPI and OpenMP 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 -qopenmp -I.
LDFLAGS = -O3 -qopenmp
```

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 on 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 = fat
#@ output = pos_lulesh_hybrid_$(jobid).out
#@ error = pos_lulesh_hybrid_$(jobid).out
#@ node = 1
#@ total_tasks = 32
#@ node_usage = not_shared
#@ energy_policy_tag = lulesh
#@ minimize_time_to_solution = yes
#@ island_count = 1
#@ queue

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

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

export OMP_NUM_THREADS=4
mpiexec -n 8 ./lulesh2.0
```

Remember to ensure that the maximum product of MPI processes and OpenMP threads do not exceed the total core count of the node used.

4 Performance Baseline (*32 points total*)

The performance of each node affects the overall performance of an application in a supercomputer. Similarly, the performance of each process and 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 the benchmark at its minimum possible size. The benchmark will later be scaled using threading and multiple processes in Section 5 .

Please note that due to implementation limitations, or in some cases even bugs, some applications that are developed with MPI mainly for distributed systems cannot run without the MPI library being linked into it. In some cases, some of these applications wont even run with a single process. Make sure to determine what is the minimum number of processes and threads that the benchmark described in Section 3 can run at for the tasks in this section.

4.1 GNU Profiler (*10 points*)

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.

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

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

The text can also be copied from the terminal, if not redirected.

4.1.1 Required Submission Files

Submit the `gprof.out` files.(*1 point*)

4.1.2 Questions

1. Which routines took 80% or more of the execution time of the benchmark? (*1 point*)
2. Is the measured execution time of the application affected by `gprof`? *Hint:* use the `time` command to determine this.(*1 point*)
3. Can `gprof` analyze the loops (for, while, do-while, etc.) of the application?(*1 point*)
4. Is `gprof` adequate for the analysis of long running programs? Explain.(*2 points*)
5. Is `gprof` capable of analyzing parallel applications?(*1 point*)

6. What is necessary to analyze parallel applications?(*2 points*)

7. Were there any performance differences between the Intel compiler and the GNU compiler? (*1 point*)

4.2 Compiler Flags (*10 points*)

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 0 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 (module `gcc/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 16.0 (module `intel/17.0`). 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 the performance metric reported by the benchmark 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. The report should include plots of relevant performance data.

For each of the combinations, prepare a script that performs 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 the appropriate SuperMUC node.
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.

Students are free to use a different sequence of steps as long as the relevant data is collected. *Hint:* scripts can use the `sed` and `awk` commands to update `Makefile` variables.

4.2.1 Required Submission Files

1. The script that automates the evaluation.(*5 points*)

2. Separate `Makefiles` that contain the best combination of parameters for the Intel and GNU compilers.(*1 point*)

4.2.2 Questions

1. Look at the compilers' help (by issuing `icc -help` and `gcc -help`). How many optimization flags are available for each compiler (approximately)?(*1 point*)

2. Given how much time it takes to evaluate a combination of compiler flags, is it realistic to test all possible combinations of available compiler flags? What could be a possible solution? (*2 points*)

3. Which compiler and optimization flags combination produced the fastest binary?(*1 point*)

4.3 Optimization Pragmas (*6 points*)

After finding a good combination of compiler flags to use when compiling the benchmark, 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

Take a look at the source code of the benchmark and identify one of the most important functions of its solver. The results collected in Section 4.1 can be used to help identify this. Such a function is typically called within a loop or has one or more computational loops in its body. Apply one of the loop optimization pragmas in the source code identified.

4.3.1 Required Submission Files

Submit the modified source file with the added #pragma annotation.(*2 points*)

4.3.2 Questions

1. What is the difference between Intel's `simd`, `vector` and `ivdep` #pragma directives? (*2 points*)
2. Why did you choose to apply the selected `#pragma` in the particular location?(*2 points*)

4.4 Inline Assembler (*6 points*)

Both the GNU and Intel compilers allow the programmer to insert assembler code directly into C programs. Do a short investigation online on what an inline assembler is and what its typical uses are. Find equivalent single line (like a simple move operation) code snippets of Intel style and AT&T style inline assembler. The actual purpose of the snippets is not important, but they need to perform the same operations.

4.4.1 Required Submission Files

Submit the pair of matching code snippets with Intel and AT&T styles.(*2 points*)

4.4.2 Questions

- Is the inline assembler necessarily faster than compiler generated code?(*2 points*)
- On the release of a CPU with new instructions, can you use an inline assembler to take advantage of these instructions if the compiler does not support them yet?(*1 point*)
- What is *AVX-512*? Which CPUs support it? Is there any compiler or language support for these instructions at this moment?(*1 point*)

5 Performance Scaling (*68 points total*)

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

5.1 OpenMP (*16 points*)

Build the benchmark with OpenMP enabled and launch it with the provided Load-Leveler script that uses its minimum number of processes.

As part of this task, modify the variable `OMP_NUM_THREADS` to measure the scalability of the benchmark. Set the variable so that the product of the minimum number of processes and the number of threads do not exceed the total number of cores in the specific type of SuperMUC node where the benchmark is being evaluated. Record the most important performance metric of the benchmark and make a plot. To make the plot, GNU Plot or a spreadsheet can be used.

5.1.1 Required Submission Files (*11 points*)

- Include the plots for both the Intel and the GNU compilers in the report.
- Copy the following information from your job script for each test:

```
Test 1:
node =
total_tasks =
OMP threads =
```

- Provide a description of the type of scaling (weak/strong) achieved for one of the compilers.

5.1.2 Questions

1. Was linear scalability achieved?(*2 points*)
2. On which thread-count was the maximum performance achieved? Was it the same for both the Intel and the GNU compilers?(*3 points*)

5.2 MPI (*20 points*)

Build the benchmark with MPI enabled and launch it with the provided Load-Leveler scripts for each possible size. 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 (*11 points*)

- Include the plots for the Intel compiler in the report.
- Copy the following information from your job script for each test:

```
Test 1:
node =
total_tasks =
```

- Provide a description of the type of scaling (weak/strong) achieved for the Intel compiler.

5.2.2 Questions

1. What are the valid combinations of processes allowed?(*2 points*)
2. Was linear scalability achieved?(*2 points*)
3. On which process-count was the maximum performance achieved? (*3 points*)
4. How does the performance compare to the results achieved with OpenMP in Section 5.1?(*2 points*)

5.3 MPI + OpenMP (*32 points*)

Build the benchmark with MPI and OpenMP enabled and launch it with the provided LoadLeveler scripts for all possible sizes. 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 (*13 points*)

- Include the plots for the Intel compiler in the report.
- Copy the following information from your job script for each test:

```
Test 1:
node =
total_tasks =
OMP threads =
```

- Provide a description of the type of scaling (weak/strong) achieved for the Intel compiler.

5.3.2 Questions

1. What are the valid combinations of processes and threads?(*3 points*)
2. Was linear scalability achieved?(*2 points*)
3. On which combination of processes and threads was the maximum performance achieved? (*3 points*)
4. How does the performance compare to the results achieved with OpenMP in Section 5.1 and with MPI in Section 5.2?(*4 points*)
5. Which solution is overall the fastest?(*5 points*)
6. Would you have guessed this best combination before performing the experiments in Sections 5.1, 5.2 and 5.3?(*2 points*)