Programming of Super Computers

Assignment 1: Single Node Performance

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

Prof. Michael Gerndt
gerndt@in.tum.de

21-10-2016

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 AMG2013 Benchmark

The parallel Algebraic Multigrid Solver 2013 (AMG2013) 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. AMG2013 is a proxy application for Laplace type problems on unstructured domains. For detailed information about AMG2013, please refer to its official page at: `https://codesign.llnl.gov/amg2013.php`.

To prepare for the tasks of this assignment, download and extract the benchmark. After that, spend a few minutes reading the provided summary PDF file (on the lecture's website) and the `docs/amg2013.readme` file found on the benchmark's archive. These contain additional details about how the code is organized, provided kernels (and their location in the source code), available problems, performance references, runtime options, build instructions, etc.

3.1 Building and Running with MPI

We will be using the Intel MPI module in SuperMUC; therefore, make sure that you unload the IBM MPI module first, and then load the Intel MPI module before building the benchmark. You can automate this by adding the following module commands to your `/.bashrc` file in SuperMUC, so that the modules are set when you login:

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

To build the benchmark with MPI support, edit the `Makefile.include` file and update the `INCLUDE_CFLAGS` variable in line 27 as follows:

```
INCLUDE_CFLAGS = -O2 -DTIMER_USE_MPI -DHYPRE_LONG_LONG -DHYPRE_NO_GLOBAL_PARTITION
```

In addition to this, make sure that the OpenMP library is not linked into our binary by omitting it from the linker options in line 31:

```
INCLUDE_LFLAGS = -lm
```

At this point, you are ready to build the benchmark. Issue a `make` in the application's directory (this can take several seconds). Note that the application needs MPI headers in all its possible configurations, so do not change the `CC` variable in `Makefile.include`.

After building, the file `amg2013` will be created under the `test` directory. 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). Load Leveler scripts are provided in the lecture's website for the application built with MPI only support; these should be submitted from the `test` directory, where the `amg2013` binary resides. Here is how the 16 processes version looks like:

```
#!/bin/bash
#@ wall_clock_limit = 00:20:00
#@ job_name = pos-amg2013-mpi-16
#@ job_type = MPICH
#@ class = test
#@ output = pos_amg2013_mpi_16_$(jobid).out
#@ error = pos_amg2013_mpi_16_$(jobid).out
#@ node = 1
#@ total_tasks = 16
#@ node_usage = not_shared
#@ energy_policy_tag = amg2013
#@ minimize_time_to_solution = yes
#@ island_count = 1
#@ notification = never
#@ queue

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

module unload mpi.ibm
module load mpi.intel

mpiexec -n 16 ./amg2013 -laplace -n 160 160 160 -P 4 2 2 -printstats
```

Make sure to familiarize with the meaning of the Load Leveler commands found in the script, as well as the module system, by reading the provided SuperMUC material on the LRZ's website (pointed at in section 2.1.4).

3.2 Building and Running with OpenMP

Some tasks in section 5 will require that you enable support for MPI+OpenMP in the AMG2013 benchmark. To add support for OpenMP edit the `Makefile.include` file and update the `INCLUDE_CFLAGS` variable in line 27 as follows:

```
INCLUDE_CFLAGS = -O2 -DTIMER_USE_MPI -DHYPRE_USING_OPENMP -DHYPRE_LONG_LONG -DHYPRE_NO_GLOBAL_PARTITION -fopenmp
```

In addition to this, make sure that the OpenMP library is linked into our binary by adding it to the linker options in line 31:

```
INCLUDE_LFLAGS = -lm -fopenmp
```

These are the only necessary changes. Make sure you issue a `make clean` followed by a `make` to build the new binary.

Load Leveler scripts are provided in the lecture's website for running the application built with MPI+OpenMP support. Here is how the 2 processes and 8 threads per process version looks like:

```
#!/bin/bash
#@ wall_clock_limit = 00:20:00
#@ job_name = pos-amg2013-hybrid-2x8
#@ job_type = MPICH
#@ class = test
#@ output = pos_amg2013_hybrid_2x8_$(jobid).out
#@ error = pos_amg2013_hybrid_2x8_$(jobid).out
#@ node = 1
#@ total_tasks = 2
#@ node_usage = not_shared
#@ energy_policy_tag = amg2013
#@ minimize_time_to_solution = yes
#@ island_count = 1
#@ notification = never
#@ queue

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

module unload mpi.ibm
module load mpi.intel

export OMP_NUM_THREADS=8
mpiexec -n 2 ./amg2013 -laplace -n 320 320 320 -P 2 1 1 -printstats
```

In summary, we rebuild the benchmark with OpenMP support and add control for the thread count per process in the Load Leveler script.

3.2.1 Required Submission Files

Submit one MPI and one MPI+OpenMP output file. The name of these files are already defined in the provided Load Leveler scripts (in the lecture's website).

3.2.2 Required Video Commentary

Answer the following questions in the video:

- What information is being reported by the benchmark?

- Which data is related to problem size, performance or correctness checks?
- Explain each of the metrics in the output that belong to performance.
- Are any of the performance metrics compounded based on others?
- Is this a weak- or strong-scaling benchmark? Explain.
- Do the provided Load Leveler scripts keep the problem size constant? Explain.

4 Performance Baseline

As mentioned in the introduction (section 1), 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 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 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. 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. 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.

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 phase 1 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.

4.1.1 Required Submission Files

Submit the Makefile that contains the best combination of parameters.

4.1.2 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?

4.2 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

Take a look at the source code of the benchmark and identify one of the most important functions of its solver. 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.

4.2.1 Required Submission Files

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

4.2.2 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 Performance Scaling

After optimizing the provided application's baseline 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

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 SuperMUC phase 1 nodes. 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

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

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

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

Build the benchmark with MPI and OpenMP enabled and launch it with the provided Load-Leveler 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

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?