Informatik und Angewandte Kognitionswissenschaft
Lehrstuhl für Hochleistungsrechnen

Rainer Schlönvoigt
Thomas Fogal
Prof. Dr. Jens Krüger

High-Performance Computing

http://hpc.uni-duisburg-essen.de/teaching/wt2013/pp-nbody.html

Exercise 3 (60 Points)

All assignments are to be uploaded to Moodle. Assignments are due at midnight on the due date. No late assignments will be accepted.

All assignments must include a `Makefile` for compiling your assignments. The assignment specification should include what the default target of your makefile should be. Assignments which do not compile will receive 0 points. Sometimes, we provide sample inputs and outputs; assignments which do not satisfy these test inputs will receive very few points.

Please do not include additional output other than what was requested by the assignment details. *Hint: if you want more debugging output, use a 'debug' flag in your program's arguments and have it only print when that flag is active.*

Your assignment will be graded on the `duecray.uni-due.de` supercomputer. It does not matter if your program runs correctly on another machine; it must run correctly on `duecray` to receive credit.

1 Scalability

We have attempted to parallelize our n-body simulations along two different axes now: MPI (distributed memory parallelism) and OpenMP (shared memory parallelism). Now it is time to see how well these performance 'improvements' have worked.

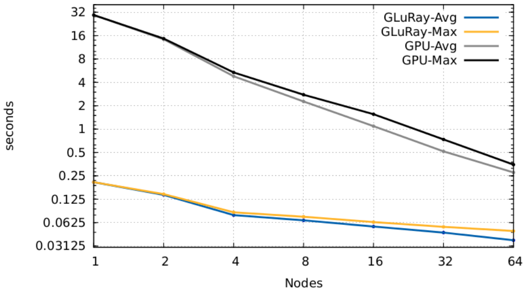

Figure 1: Strong scalability of a parallel rendering application. The application was run in 4 different 'modes' using the same dataset, and those timings were combined to create this chart.

In this assignment, you will measure and improve the *scalability* of your simulation code. Scalability is the changing performance of a parallel program as it utilizes more resources. Of course, the reasons we turn to parallelize simulations in the first place are one or both of:

1. We want to simulate models of **similar complexity**, but *faster*, or
2. we want to run **larger** or **more accurate** models.

we call these aspects of scalability **strong scaling** and **weak scaling**. Strong scaling is identifying how a program gets faster for a fixed problem size. That is, we have the same problem, but we run it with more and more processors, and we hope/expect that it will compute its result faster. Figure 1 depicts a strong scaling study[1]. Weak scaling speaks to the latter point: how effectively can we increase the size of the problem we are dealing with? In a weak scaling study, each processor has the same amount of work, which means that running on more processors increases the total size of the simulation.

The broad task in this assignment is to optimize your n-body simulation so that it scales better. To do this effectively, you must first the establish the baseline scaling characteristics of your simulation. Afterwards, you can modify your program to improve scalability. Your existing benchmarks can then be used to verify the increase in performance. To keep things simple, we will limit ourselves to strong scaling for this assignment.

This assignment is split into 2 parts, with differing deadlines! In the first part, you need only measure the scalability of your code. In the second, you

[1] From http://www.sci.utah.edu/~brownlee/gluray.pdf

will use this information and the scripts you have written in order to improve your application's scalability.

Part 1 will be due on Tuesday, January 14th. Part 2 will be due on January 28th.

2 Part 1: Establishing Scalability

You will establish the scalability of your code on two fronts: strong scalability and weak scalability. To do this you will need to perform a number of runs, and extract the execution time for each run.

In principle, all that needs to be done is to run the program multiple times, changing the `-np` argument to `mpirun` each time. Such a study could be as simple as the following `bash` for loop:

```bash
for procs in 1 2 4 8 16 32 64 ; do
  time mpirun -np ${procs} ./my-program ./the-input
done
```

Listing 1: How to run a strong scaling study.

of course, a real study would need some way to capture the performance of each of those runs. Furthermore, running on `duecray.uni-due.de` is more complicated due to the job submission system (i.e. PBS/`qsub`).

2.1 Scripting

You must script your job executions, so that you can *quickly* measure scalability. You will be making many small changes to your code in the second part of the assignment, and it is important to be able to *quickly* ascertain which version of the code is faster. For this part of the assignment, you will implement two scripts: a strong scaling script, and a weak scaling script.

2.1.1 The *B*ourne-*A*gain *SH*ell (`bash`)

I recommend you use the language of your shell, `bash`. There are many tutorials online for you to learn from; a couple seemingly good ones are:

- `http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`. Your fearless leader started with this tutorial, way back in the good ol' days.

- http://wiki.ubuntuusers.de/Shell/Bash-Skripting-Guide_f%C3%BCr_Anf%C3%A4nger. This is in your native language, which you might prefer.

If you would like to delve deeper, the O’Reilly book “Learning the Bash Shell”[2] is also quite good.

Below you will find details on what, exactly, your script should do, and examples are given using bash syntax. You are free to use any scripting language which is available on duecray, however.

2.2 Strong scaling

Your script should run a strong scaling study. The script should take 2 arguments: the path and name of a binary to execute, and the name of an input file. It should perform multiples runs of the given binary with the given input file (varying the number of processors), collect the execution time for every run, and output those execution times to a file.

To be clear, calling the script in a form such as:

```
$ bash strong-scaling.sh /path/to/my/nbody /path/to/particles.txt
```

should create a number of jobs on the cluster, in which each job runs:

```
/path/to/my/nbody /path/to/particles.txt
```

except that the number of processors used each time varies.

Your script must run on duecray, which uses qsub/aprun to execute jobs. However, you probably run locally with just mpirun or mpiexec. Your script may optionally take a third argument, which changes which mode it uses to submit jobs.

Your script must provide help information on how to run it. If the first argument is ‘-h’, then the script should print out usage information and then exit. An example is given in Listing 2.

```
$ bash scaling.sh  -h
Usage: scaling.sh /path/to/binary /path/to/input [mpirun]
  binary   the program to run
   input   the input file to give as argument to the binary
  mpirun   submit runs with ’mpirun’ instead of qsub/aprun
```

Listing 2: Sample script ‘help’ output.

[2] http://shop.oreilly.com/product/9781565923478.do

The output of the script should be a text file with each line representing the performance at the given number of processors. You must include a 'reasonable' set of runs. Listing 3 gives a sample output file; the first column is the number of processors and the second column gives the runtime[3].

```
 1  560.1325
 4  140.0331
 8   70.0166
16   35.0083
24   23.3389
48   11.6694
96    5.8347
```

Listing 3: Output format of strong scaling script. This represents 7 runs from a strong scaling study.

I recommend the following stages for implementing this script:

1. create a script which writes a `.pbs` file for execution on a single processor.
2. Modify the script to generate n `.pbs` files, with each iteration using a different number of processors.
3. Add the `qsub` command after creating the files and ensure the jobs execute through the queue system acceptably.
4. Create some way to uniquely identify the performance of each run.
5. Collate all the performance metrics into a single file.

In what follows I give details/recommendations on UNIX tools and scripting approaches you can use to implement your assignment.

2.2.1 Creating .pbs files

To create the `.pbs` file, you can use standard shell redirection with a feature of `cat` that allows you to write a block of text into a file.

```
#!/bin/bash
cat > fname <<EOF
#PBS -n job-name
#PBS -l mppwidth=42
#PBS ...
EOF
```

Listing 4: Bash file creation example.

[3] You may include additional columns—for example, if you wanted to differentiate between number of MPI processors and number of OpenMP processors.

2.2.2 Generate n **.pbs** files

This will require a `for` loop over the number of processors, as detailed in Listing 1. Furthermore, you will need to change the number of processors in each iteration. Using the `cat` idiom above, you can simply insert variable references:

```
#!/bin/bash
meaning=42
cat > abc <<END
The meaning of life is ${meaning}.
END
```

will create a file `abc` with the contents

> The meaning of life is 42.

2.2.3 Ensure the jobs run through the queue system

Add `qsub` into your scripts. Test to make sure it 'works'. You should get output files for every run you submit. Use `qstat` and `qstat -a` to watch the queue after your script executes, and make sure your jobs (eventually) get executed.

2.2.4 Uniquely identify performance

There is a lot of flexibility here. One solution is to modify your job submission script to include the performance information in the (standard) output file.

1. Change the `.pbs` files you generate so that each run generates unique output files ("`#PBS -o ...`"). You could do this by, for example, using the number of processors as part of the filename.

2. Use `date +%s` to identify the current time. Identify the current time both before and after you run your program.

See the date man page for more information.

Another solution is to modify your simulation to measure the time and have it output this information in some way: either to a specially-named file, or to standard output on rank 0.

2.2.5 Collate performance numbers

This depends on how you have uniquely output the performance numbers. One problem is waiting until all the jobs are finished: `qsub` returns instantly, but the job may not run until (much) later. So if you try to instantly collate the performance metrics at the end of your script, much or all of the data may not be available yet!

One solution is to put the collation algorithm into a separate script. You can then submit your collation script as a 'job' to run, and use PBS to force it to run later. For details, see:

`http://beige.ucs.indiana.edu/I590/node45.html`

but the gist of it is to save each `qsub` execution jobID and make your collation script `depend` on it. z.B.:

```
job1=$(qsub 1procs.sh)
job2=$(qsub 2procs.sh)
job3=$(qsub 4procs.sh)
qsub -W depend=afterok:${job1}:${job2}:${job3} collate.sh
```

Listing 5: Using PBS job dependencies.

Now the queueing system itself will make sure `collate.sh` does not run until all of `1procs.sh`, `2procs.sh`, and `4procs.sh` have completed.

2.3 Submission

As always, you submission should be a single (optionally compressed) tarball. The tarball should extract all files into a single subdirectory. In the tarball you should include:

- Your simulation's source code.
- A `makefile` to compile your code.
- Any and all scripts to measure the strong scalability of your code.
- A graph (image!) which depicts the scalability of your code (i.e. something similar to Figure 1).

You will not be graded on the scalability of your code for this portion of the assignment—do not worry if your program is slower with more processors.

As always, submit your tarball on Moodle.

3 Part 2: Competition

In the second part of the assignment, you will use your script to improve the performance of your simulations. Your programs will be pitted together on the performance battlefield, with the best-performing program author taking home two glorious rewards:

1. the highest grade possible on the assignment, and
2. bragging rights.

Your submission will include information detailing to what extent your efforts have improved the performance of your program, in the form of a scalability graph.

In this stage, you should submit a tarball with:

- Your simulation's source code.
- A `makefile` to compile your code.
- Any and all scripts to measure the strong scalability of your code.
- One or more graphs (images) which depict the strong scalability of your code both *before* and *after* you have improved performance.

Make sure the program still works ***correctly*** *after you optimize any aspect.* Programs which do not generate the correct answers after your efforts will not receive points for improved performance, and will not be considered for the competition. I will provide a program which is the standard for correctness.

As always, submit your tarball on Moodle.

4 Grading details

The grading on this assignment is meant to encourage some friendly competition. First, I will emphasize that the majority of the points on this assignment are completely independent. Second, there is a fallback so that you can get 100% credit without regard to your peers' efforts.

The grading rubric is summarized in Table 1. In the first part of the assignment, you will measure and present your programs scalability: this is

Aspect	**Points available**	**Evaluated From**
Independent grading		
Correctly measuring/presenting scalability	10	Part 1
Simulation correctness	30	Part 2
Improved performance	5	Part 2
Competitive grading		
Performance	15	Part 2

Table 1: Grading breakdown for this assignment.

worth 10 points. In the second part of the assignment, you must submit a correct simulation—which you should already have—and demonstrate that you have improved the performance in some way. If your simulation is not already giving the correct answers, this should be your first priority.

To receive the final 15 points, your best bet is to meet the performance of a program I will provide. This program uses all the high-performance knowledge we have learned in class, as well as knowledge about the programs' environment. There are no 'tricks' to the performance of this program, beyond what has been given in class or publicly available in the documentation for the tools we have used in this course. However, I will not directly describe how this program works. Hints will be given to enterprising students[4].

The other option to ensure you get all of the final 15 points is to beat your peers. The fastest program will receive full credit. Other programs receive some percentage of the 15 points, depending on how closely they match the fastest program's performance. For example, if the fastest program requires 100 seconds, and your program is 30% slower (i.e. it takes 130 seconds for the same problem), then you receive 70% of the credit—10.5 points[5].

For the purposes of this competition, the program I provide cannot be the 'fastest program'.

5 Increased Resources

To aid in scalability testing, your cluster walltimes have been increased to 30 minutes, and you can run on up to 96 cores. These increased allocations will

[4] i.e., you need to ask.

[5] Moodle does not allow fractional points. We will round as per the normal rules of math: $\geq .5$ rounds up, $< .5$ rounds down.

continue for the duration of the assignment.

Be aware that this makes your jobs more contentious! While you are probably used to running on 24 cores instantly, this may not be the case anymore when you try to enqueue a 96-core job. “I could not get enough cores to effectively run my benchmarks” is *not* a valid excuse—start early.

6 Extras: isolating performance problems

It can be useful to be able to evaluate MPI scalability and the OpenMP scalability separately: this will help you narrow down which parts of your code are the major inhibitors to good scalability. Here I outline some tricks you can use to change which aspects you measure. You are not required to implement these for your strong scaling script, but they will be useful in the second half of the assignment.

6.0.1 OpenMP

You can effectively disable OpenMP by setting both the environment variables `OMP_NUM_THREADS` and `OMP_THREAD_LIMIT` to 1 before you run your program (note you do not need to recompile your program). This can allow you to isolate MPI-specific performance issues from OpenMP ones.

6.0.2 MPI

Removing MPI is a bit harder. I recommend creating a special command line option, perhaps “`-serial`”, and creating wrappers around all of the MPI functions. You can then disable them if the command line option is given. Listing 6 demonstrates this approach. You can either run this program as: ‘`mpirun -np 9 ./a.out`’ or you could run ‘`./a.out -serial`’.

```
#include <stdbool.h>
#include <string.h>
#include <stdio.h>
#include <mpi.h>

/* should we utilize MPI or not? */
static bool mpi = true;

static int
par_init(int* argc, char*** argv) {
  if(mpi) {
    return MPI_Init(argc, argv);
  }
  return 0;
}
static int par_finalize() { return mpi ? MPI_Finalize() : 0; }
static size_t
```

```
par_rank()
{
  if(!mpi) { return 0; }
  int rank=-1;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  return (size_t)rank;
}
static size_t
par_size()
{
  if(!mpi) { return 1; }
  int sz=-1;
  MPI_Comm_size(MPI_COMM_WORLD, &sz);
  return (size_t)sz;
}
int
par_barrier()
{
  return mpi && MPI_Barrier(MPI_COMM_WORLD);
}
int
par_bcast(void* buffer, int count, MPI_Datatype dt)
{
  if(mpi) {
    return MPI_Bcast(buffer, count, dt, 0 /* root */, MPI_COMM_WORLD);
  }
  return 0;
}

int
main(int argc, char* argv[]) {
  for(int i=0; i < argc; i++) {
    if(strcmp(argv[i], "-serial") == 0) {
      mpi = false;
    }
  }
  par_init(&argc, &argv);
  fprintf(stderr, "i am %zu of %zu\n", par_rank(), par_size());
  par_finalize();
  return 0;
}
```

Listing 6: Example method for abstracting MPI calls so that a program can also be run in serial mode without recompiling.

Through the use of the aforementioned environment variables and an option like the '`-serial`' one above, you should be able to benchmark different components of your program independently.