ICS632: Assignment #4

For this assignment turn in an archive of a single directory with your code in source files as specified in the exercise(s). Provide a report that answers specific questions in the exercises. For this report turn in a PDF file named `README.pdf` with both text and graphs (no points awarded for prettiness, only for readability and content).

Run your code on a machine on which you have SimGrid and SMPI installed. You also will need to run code on our Cray system. Running the programs takes a bit of time. It's not a good idea to start running code right before the due date as you may not get your results in time (e.g., due to queue waiting time).

Of course, it is recommended to write scripts/makefiles/whatever to automate running the code and gathering performance numbers.

This assignment appears very long, but it's because it includes a lot of tutorial material that's will be useful for simulation-based projects.

Overview

The goal of this assignment is to gain exposure to the traditional ~~joys~~ difficulties of MPI distributed memory programming, in particular the issue of local vs. global indices. The involved mental gymnastics is actually found in many other domains (e.g., computing on the GPU). To make your life easier and avoid intricate arithmetic, all questions below use the simplest of assumptions (the platform is a homogeneous cluster, everything divides everything, everything that needs to be a perfect square is, etc.)

We focus on the ~~boring~~ age-old matrix multiplication. It's not exciting, but it's among the easiest. It may be a good idea to use at least part of the program skeleton that was provided for the previous programming assignment as a starting point.

Each process will perform local (non-MPI) matrix multiplication. To that end use a simple sequential multiplication algorithm with 3-nested loops, with a good (locality-wise) loop ordering.

Warning: Unlike in the previous assignment, we simulate computation, which SimGrid does by benchmarking your code on the fly. Therefore you should run simulations on a dedicated (i.e., idle) system, and you may want to run multiple trials of your experiments.

Calibrating your simulations

We will simulate parallel platforms in which each processor computes 32 times faster than one core of my current laptop. This means that you should pass a particular value to smpirun using the `--cfg=smpi/running_power:VALUE` command-line option. In the previous assignment we always passed a value of 1, meaning that you were simulating a platform in which each processor was as fast as your own machine (it didn't matter in the previous assignment since there was no computation).

To find out what value you should pass to `--cfg=smpi/running_power` on your machine, a Python script called `calibrate_flops.py` is provided. Simply run this script and, at the end of its output, it will tell you the value to use. You'll have to re-run this script if you change machine (and re-run all your experiments hereafter).

2-D Matrix Multiplication

We consider the standard outer-product parallel matrix multiplication described in class for a 2-D block data distribution. We consider the $C = A \times B$ multiplication where all three matrices are square of dimensions $N \times N$ and contain **double precision floating point numbers**. The execution takes place on p processors, where p is a perfect square. Each processor thus holds a $N/\sqrt{p} \times N/\sqrt{p}$ square block of each matrix.

All your programs should abort if p is not a perfect square, or if $\sqrt{p}$ doesn't divide N.

To make it possible to check that your code multiplies the matrices correctly, we multiply particular matrices defined as:

$$A_{i,j} = i$$

$$B_{i,j} = i + j$$

(indices go from 0 to $N-1$, as in the C language). Each process in your MPI program should thus allocate three matrix blocks, and populate the A block and the B block with the correct elements (the C block should be initialized to all zeros). Of course, the above is for global indices, not local indices!

We analytically check that the result of the matrix multiplication is correct by summing all the computed elements of C. The sum should be:

$$\sum C_{i,j} = N^3(N-1)^2/2 \; .$$

The MPI processes should be arranged logically in a 2-D processor grid of dimensions $\sqrt{p} \times \sqrt{p}$, and thus identified by two coordinates. For instance, for 4 processes, here is the mapping of the MPI ranks to your logical coordinates: $0 \rightarrow (0,0)$, $1 \rightarrow (0,1)$, $2 \rightarrow (1,0)$, $3 \rightarrow (1,1)$ (i.e., a row-major order).

A 1600-host XML cluster platform description file and accompanying hostfile are provided for simulating your code's execution.

Question #1: Creating the matrices [10 pts]

Implement an MPI program called `matmul_init` that simply generates the A and B matrix blocks. The program takes a single integer input, N (provided as a command-line argument or declared by a `#define N xxx` clause, up to you).

The program should have each process print out its matrix blocks. Here is an example output from rank 2 in a 4-process execution for a 8×8 matrix:

```
Block of A on rank 2 at coordinates (1,0)
4.0 4.0 4.0 4.0
5.0 5.0 5.0 5.0
6.0 6.0 6.0 6.0
7.0 7.0 7.0 7.0
Block of B on rank 2 at coordinates (1,0)
4.0 5.0 6.0 7.0
5.0 6.0 7.0 8.0
6.0 7.0 8.0 9.0
7.0 8.0 9.0 10.0
```

Note that due to the convenience of simulation, the output from one process will not be interleaved with the output of another process. The order in which the processes output their blocks is inconsequential. You may want to use a barrier to first see all blocks of A and then all blocks of B.

Obviously, we won't run this program for large N. The goal is simply to make sure that matrix blocks are initialized correctly on all processes.

Hints:

- Feel free to use the same .c code for this entire assignment, and use C macros to generate the different executables.
- It's fine (and some prefer) to not use 2-D arrays in C, but instead to use 1-D arrays, i.e., `A[i*n+j]` for a 1-D array declared `double A[n*n]` or allocated as `malloc(n*n*sizeof(double))`, instead of `A[i][j]` for a 2-D array declared `double A[n][n]`. Note that using 1-D arrays is required if N is unknown at compile time (e.g., passed as a command-line argument).

Question #2: Multiplying the matrices [40pts]

Augment your program from the previous question, in a new program called `matmul_outerproduct_v1`, so that the matrix multiplication happens using the outer-product matrix multiplication algorithm. At the end of the program, each process should sum up its computed elements of C. These sums should then be reduced (i.e., using `MPI_Reduce`) so that the process of rank 0 has the sum of all

the elements. This process then checks that the sum is correct by comparing it to the analytical formula. The program should output the (simulated) wall-clock time (which should only include the computation, not the memory allocation, memory deallocation, matrix initialization, result checking, etc.).

One way to implement the algorithm without losing one's mind is to create a bunch of MPI communicators to allow straightforward broadcasts along the rows/columns of the processor grid. Check out the MPI documentation/tutorial for relevant communicator creation functions. The learning curve is well-worth the benefits.

Hints:

- For debugging and making sense of your output, using `MPI_Barrier` is often a good idea (adding barriers here and there can force a step-by-step synchronous execution that's easier to follow/understand).
- Note that unnecessary barriers slow down execution (both for the simulation and for a real execution). In general, one tries to avoid global synchronizations at all cost.

Question #3: Impact of the network speed [10pts]

Using $N = 1600$, run experiments with `matmul_outerproduct_v1` for $p = 1, 4, 16, 64, 100, 400$ to measure the average (simulated) wall-clock time using the provided XML platform file (`cluster_1600.xml`). Then modify the platform file to set the network latency to 0 and the bandwidth to a very large value, so as to simulate an ideal platform in which the network is very fast. Re-run the experiments.

Plot speed-up curves (speedup vs. number of processors) for the two series of experiments. What parallel efficiency can you achieve with $p = 1600$ processors on the ideal vs. the non-ideal platform? What do you conclude about the application's prospects for this matrix size on a real machine?

The simulation takes time and memory. You can decrease the scale as needed in case you can't run it on your machine. Just for reference, this runs in about 10 minutes on my laptop (with 10 trials per experiment).

Question #4: Communication share [20pts]

Instrument your code so that it also reports the time spent in communication. Using $p = 16$ processors, run experiments with the original platform file for $N = 8 \times x$ for $x \in \{100, 110, 120, \ldots, 400\}$. This runs in about 20min on my laptop (with 10 trials per experiments).

On the same plot show the computation time and the communication time vs. the matrix size. What do the trends look like? Do they make sense? Is this good news or bad news for multiplying large matrices on our platform?

Question #5: "Cheating" with Simulation [15pts]

We want to simulate execution with large matrices so that we can observe reasonable scaling for large numbers of processors. The problem is that experimenting with large matrices is time consuming (whether on a real platform or with our simulator). Here, we take this opportunity to use simulation to "cheat" and not perform any actual computation. This is something researchers often do in simulation and here we're using this assignment as an opportunity to learn how to do it. This may come in handy for all kinds of course projects. In fact, what's below is verbose and a bit lecture-like rather than assignment-like.

When our simulator encounters a zone of computational code it executes and times this code so as to generate the correct delay in simulation. If this code takes 1 hour to run on your machine, then the simulation will take 1 hour and change. If you're simulating 100 processes each computing for 1 hour, then the simulation will take 100 hours! There are many ways in which this can be avoided with SimGrid. For instance, one can tell the simulator: "Execute/time the code only the first time you encounter it." A more radical option is to simply tell the simulator: "When you encounter this basic block, instead of running it I am telling you how many flops it corresponds to and you can just compute the delay without computing anything."

We'll use the second option above to avoid computation altogether. This is doable because matrix multiplication is very deterministic and easy to analyze. Here is how this is done for a simple loop that computes something:

```
double flops = (double)N * (double)FLOP_CALIBRATION_FACTOR;
SMPI_SAMPLE_FLOPS(flops) {
   // Real code, commented out
   // for (i=0; i < N; i++) {
   //   sum = sum * 3.0 * x * y;
   // }
}
```

where `FLOP_CALIBRATION_FACTOR` is a constant to be determined. Note that `flops` is linear in N since we abstract away a single loop that has N iterations. `FLOP_CALIBRATION_FACTOR` is usually very low. You should instead have something like this:

```
double flops = (double) N * (double)FLOP_CALIBRATION_FACTOR / 1.0E+10;
```

so that you have a less unwieldy calibration factor.

Since we're not computing anything real, you can also comment out the part of the code that initializes the matrix, and the part that checks the results. Why waste time at this point?

Besides time, another limiting factor for scalable simulation is space. If I want to simulate 100 MPI processes that each uses 1GiB of heap space, then I need a machine with 100GiB of RAM! For this reason, SMPI allows you to have all simulated processes use the same heap allocations!! To this end you must:

- Replace `malloc` by `SMPI_SHARED_MALLOC`.
- Replace `free` by `SMPI_SHARED_FREE`.

After the above modifications, we now have a new "implementation," which we call `matmul_outerproduct_v3`). This program computes nothing useful at all, but still outputs simulated wallclock time! It can be executed faster and for larger matrices. We now have an interesting trade-off. With a lot of processes, we can scale the memory up. For instance, say the memory space needed for the sequential computation is 100GiB. You can't run the simulation of this computation on your laptop. However, if you simulate a parallel execution with 100 processes, then with the above memory trick, you only need 1GiB of RAM to run the simulation. However, with a lot of processes, the time to simulate the broadcasts with large data sizes is large (due to the sheer number of individual point-to-point messages). So, some simulations will take too much space, and some simulations will take too much time. However, the above tricks make the rage of runnable simulations much wider.

1) Re-calibrating the simulation – One problem is that the simulated wallclock time will no longer be coherent with our simulation platform, i.e., it no longer has anything to do with our `--cfg=smpi/running_power` command-line option. So the first thing to do is to determine a reasonable value for `FLOP_CALIBRATION_FACTOR`. Run `matmul_outerproduct_v2` for a matrix size of 2000×2000. Then empirically determine a value of `FLOP_CALIBRATION_FACTOR` that leads to the same (or close) simulated elapsed time for `matmul_outerproduct_v3`. Doing a binary search on `FLOP_CALIBRATION_FACTOR` is great of course, but you can probably just do a quick trial-and-error search.

2) Running new experiments – Now that your simulation is fully calibrated, run simulations on the original cluster (`cluster_1600.xml`) with 100 processors, using matrices of increasing sizes up to as large as you can run them on your machine (in terms of memory and time). It may be a good idea to do a quick back-of-the-envelope memory footprint calculation to make sure you don't exceed your RAM. Plot an efficiency vs. matrix size curve. To compute the efficiency, you can **simply compute what the (simulated) sequential time would be as:** $N^3 \times$`FLOP_CALIBRATION_FACTOR` / 10^{10}.

The plot should reveal our expectation from Question #4 that for large matrices we should be fine.

Question #6: Running on our cluster [15pts]

Execute your code on **4 nodes** of the Cray cluster:

1. Using **16 sequential MPI processes per core** (wasting 4 cores), i.e., for a total of $4 \times 16 = 64$ MPI processes; and
2. Using **one 16-way multi-threaded (with OpenMP) MPI process per node**, i.e., for a total of 4 MPI processes (see our 2nd assignment).

Using a reasonably large matrix size, compare the performance of both these implementations and report on the performance. Use multiple trials and comment on variability across trials. Which implementation is the most efficient overall? Feel free to run whatever experiments you think make sense (e.g., at larger scale).

Beyond this assignment towards a project?

Matrix Multiplication:

- One issue with `MPI_Bcast` is that it is blocking! Therefore you cannot overlap communication and computation. This partly explains the terrible parallel efficiency in Question #3. You can implement your own broadcast scheme (e.g., inspired by the previous programming assignment) so as to hide as much as the communication overhead as possible.
- There are many matrix multiplication algorithms (e.g., Cannon, Fox, Snyder, etc.), and implementing/comparing them is of course interesting.
- One difficult question is that of platform heterogeneity that could be explored in a project as well... (we'll discuss heterogeneity in class).

LU Factorization:

- There are many known candidates for parallization in linear albegra, and in class we've talked about LU factorization.
- A whole project and be done for LU, with all kinds of bells and whistles.
- It's better to process blocks of columns rather an individual columns (i.e., fewer iterations mean fewer communication operations)
- Using `MPI_Bcast` is overkill, and there are clever ways to use point-to-point communication to better overlap computation and communication
- Using asynchronous communication should allow you to achieve better overlap as well
- Extending your code to handle heterogeneous platforms is interesting