Parallel Programming Lab, Week 8:

Starting your Development Project

The emphasis in labs for the remainder of the term is on your own *development project* involving parallel programming. There are likely to be one or two more short, "scripted" labs this term, but those will be mainly to provide extra skills for the project work.

This week you should start thinking about what you want to do for your project. What follows below is a short list of possible projects, or at least starting points for these projects. Some of these ideas are concerned primarily with exploring different kinds of software infrastructure to support parallel programming, and some take the form of specific applications that you can try to parallelize. You can mix and match between the two different types of project. You are also strongly encouraged to consider *different* applications, or to use different programming frameworks you may have access to - e.g. Cilk, Intel Threaded Building Blocks, task farms using Web Sockets (??), etc. But please discuss these applications or technologies with me, before starting.

In all cases, your report on the project, which forms a major part of the lab book assessment, should include extensive benchmarking and analysis of parallel speedup and efficiencies obtained for different numbers of cores and different problem sizes - preferably as tables and graphs.

Some Project Ideas

- Apply the OpenMP framework for shared memory parallel programming to a selection of the examples introduced in earlier lectures and labs, or to examples given below. If you have experience using C or C++ you can convert the examples to one of those languages and use the GNU C/C++ compiler under CentOS, as described at the end of the Week 4 lab (or any other version of Linux, or Windows if you have access to a suitable compiler that supports OpenMP).
- Undertake a systematic benchmarking of MPJ Express communication primitives in the teaching lab environment. The basic tool would probably be an MPJ *ping-*

pong benchmark program you would write, in which one process sends a message to another, and that process reflects it back. The round trip time is measured and halved to give the basic *message latency*. This is repeated for a range of message sizes from one byte to megabytes to find the *message bandwidth*. Try this between a range of PCs in the same labs, different teaching labs, and with connections to wireless laptops, computers at home, and so on.

Relate your findings to parallel speed-ups obtained in full MPJ programs like the Laplace solver and task farm explored in earlier labs - try to give a model or formula for their performance.

- If you have access to a system with an appropriate GPU card, and you are willing to program in C or C++, try to apply CUDA or OpenCL to one or more computational applications - either applications encountered in these labs or other suitable applications discussed with the tutor (there are also GPU systems in the department - ask me if you need access to these).

  Compare your results to parallel and sequential programs running on multicore CPUs.

  If you are interested in trying GPU programming from Java rather than C/C++, there *are* frameworks that make this possible. You may, for example, look at Aparapi. I will talk about this in this week's lecture.

- Here is a simple Java program for raytracing, adapted from a C++ program from Scratchapixel. It only deals with scenes containing spheres, but produces quite striking results.

  Experiment with parallelizing the Java version of this program using Java threads and also an MPJ Express taskfarm (the C++ could alternatively be parallelized using OpenMP - see above). The main loops that will be parallelized are in the render() method. Expect load-balancing to be an issue.

  If you need to make the problem more computationally demanding, and thus more suitable to exploit parallelism, you can adapt the original version of this program. You should be able to increase the image size, add many more spheres (dozens or hundreds, perhaps?) Because the generated image suffers quite noticeably from "aliasing", you might want to add supersampling, which may also greatly increase the number of computations.

- Here is a simple molecular dynamics program in Java, adapted from an applet by Dan Schroeder. It simulates a two dimensional system of atoms, and exhibits phases that look very like solids, liquids and gases - try adjusting

the VELOCITY constant which effectively changes the temperature of the system.

Try to parallelize this by using Java threads, decomposing the main loops over particles in a way similar to problems we have considered in earlier labs. Also try to parallelize it using MPJ by dividing the particle arrays over processes.

It may be challenging to achieve good parallel speedup in the distributed memory case because the calculation of accelerations requires positions of all particles to be broadcast in every iteration. Cell lists provide a better organization - try to understand why, even if you don't have time to implement this approach.

- In a similar vein, here is a simple program that simulates motion of stars in a galaxy using Newton's laws. I don't really expect you to get any beautiful spiral galaxies, but at least you should produce some "elliptical clusters"!

  Try to parallelize this, as discussed above in the molecular dynamics project (the structure of the program is very similar).

  (Nothing to do with parallelism, but it might be useful to adapt the program so that it displays two different windows - one with a view from "above" (x-y plane) and one from the side (x-z plane)...)

  In the sample program I used the naive algorithm for the "N-body problem", which involves $O(N^2)$ force calculations in every update (one inverse-square law for each star pair). This at least increases your chance of getting parallel speed-up for large enough N. Real large scale simulations of galaxies and the universe often use some variation on the theme of Barnes Hut Simulation. The Barnes-Hut scheme will be briefly discussed in the week 9 lecture, and there is a sequential Barnes-Hut code available here. You may alternatively wish to try to parallelize this code for your project.

- Write some basic matrix arithmetic operations in Java (or another language), and try to parallelize them using Java threads or MPJ Express (or any other approach).

  You should be able to make a parallel version of matrix multiplication, at least, that achieves good parallel speedups. Generate two random matrices of size N by filling them with random numbers, and multiply them together by parallel and sequential algorithms. For testing, make sure your program compares the result of the parallel algorithm with the (presumably correct) result of the sequential version, element by element. Compare timings of both algorithms for parallel speedup.

  For a bigger challenge, try to make a parallel version of solution of linear equations

by the LU decompostion method. (As discussed in the week 1 lecture, this is the basis of the Linpack benchmark used in the TOP500 list.) To simplify, don't worry too much about "pivotting".

- Write a parallel version of a simple prime number sieve. To test a number *n* for primeness, just try dividing it by all numbers from 2 to sqrt(*n*). A fairly naive sieve tests all numbers from 2 to *N* in this way and returns a list of all prime numbers in this range.

  This problem is suitable for implementing as an MPJ task farm. As a single task, the master sends a subrange of numbers to a worker (just a start value and end value for the range will do). The worker tests them for primality, and returns a list (array) of prime numbers. The master should concatenate all the lists together and output them as one long list, in the right order, at the end of the run (you don't need include the final output stage in your timing).

  Can you achieve a parallel speedups? For large enough *N* it should be possible. (If you generate enough prime numbers, maybe you can sell them...)

  A more efficicent algorithm is the classical Sieve of Eratosthenes. Can you parallelize this algorithm?

- There are numerous variations possible on problems considered in earlier labs - calculating variants of the Mandelbrot Set like the Julia Set, simulating cellular automata other than Life like Brian's Brain, solving the three-dimensional Laplace Equation rather than the two-dimensional Laplace Equation, and so on.

  If you are going to attempt such variations, make sure you do a thorough study of the parallelization. For novelty you might, for example, try decompositions using square blocks rather than the simple whole-row or whole-column decompositions we have typically used in earlier labs, or to introduce the use of MPI collective communications.

- (For the more adventurous) investigate the feasibility of parallel Bitcoin mining. It is almost certainly impractical for you to implement a real Bitcoin node, but the central "proof-of-work" algorithm that earns miners new bitcoins is a search problem that involves finding a nonce (essentially a random number) whose inclusion in some given data structure ("block") leads to a SHA-256 hash value for the block less than some threshold (there is a good explanation here - you will have to research the relevant issues).

  You could investigate parallelizing a similar search problem with a farm or a GPU, say.

- In recent years a number of open source parallel or distributed computation frameworks have been developed specifically to support processing of so-called "Big Data". Examples include:
    - Apache Hadoop with its Map-Reduce parallel programming paradigm.
    - Apache Spark is a general computation engine for big data, with many associated libraries.
    - Apache Giraph for processing large graph-oriented data structures.

  The school has a cluster dedicated to teaching and research on these technologies. If you want to do a project using Hadoop or Spark in particular, I can provide an account on this cluster, where these frameworks are already installed. You could for example evaluate the parallelism built into Spark machine learning libraries, or try to use Hadoop or Spark to parallelize some simple algorithms like as the ones we have considered earlier.

In a related previous module, there were many interesting development projects over several years - either variations on themes above or completely original. For your further inspiration, here are a few examples:

- Parallel prime sieve using Java Fork/Join framework.
- Julia Set computation distributed across Web browsers using Web Sockets.
- Parallel edge detection (image processing).
- Parallelization of matrix multiplication by AparAPI and other approaches.
- Simulation of cardiac tissue on GPUs using CUDA.
- Parallel computation of pi digits by BBP and other methods.
- Parallelisation of a genetic algorithm.
- Parallel prime sieve using a GPU.
- "Galaxy" simulation on a GPU using AparAPI.
- Parallel solution of wave equations in one and two dimensions.
- File transfer using MPJ.