Informatik und Angewandte Kognitionswissenschaft
Lehrstuhl für Hochleistungsrechnen

Thomas Fogal
Prof. Dr. Jens Krüger

High-Performance Computing

http://hpc.uni-due.de/teaching/wt2014/nbody.html

Exercise 4 (80 Points)

All assignments should be pushed to your personal Git repository. 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. Assignments which do not compile will receive 0 points. Assignments that do not satisfy the test inputs will receive 0 points.

Please **do not include output** other than what was requested by the assignment details.

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 Introduction

In this assignment you will introduce OpenMP-based parallelism into your N-Body simulation. The primary objective is to accelerate the computation when there are a large number of particles.

Some students may be able to implement this assignment using a couple lines of code and some makefile hackery. Good for you! I would encourage you to spend your time cleaning up any loose ends, beginning to evaluate the performance of your simulation, and, most importantly, verifying the correctness of your simulation. Correctness has a profound implication on your grade; performance does not (yet).

1.1 Message passing vs. shared memory

OpenMP is based on a *shared memory* model. This is in strict contrast to the *distributed memory* model that MPI allows. The separate threads of execution within a shared memory model execute within the same address space. This means you do not need to 'send' and 'recv' information: each executing thread can just access the chunk of memory directly.

The aspects of OpenMP that we will use in this course follow the 'fork-join' model. The general idea is that execution of your program is predominantly serial in nature. At designated points within your code, the process will 'fork' a number of threads that will operate on some subset of data. Each thread will proceed until a 'join' point, where all threads wait and eventually collapse, leaving just one thread remaining. That serial thread continues until the next 'fork' location. The contrast to this and the MPI model is displayed in Figure 1.

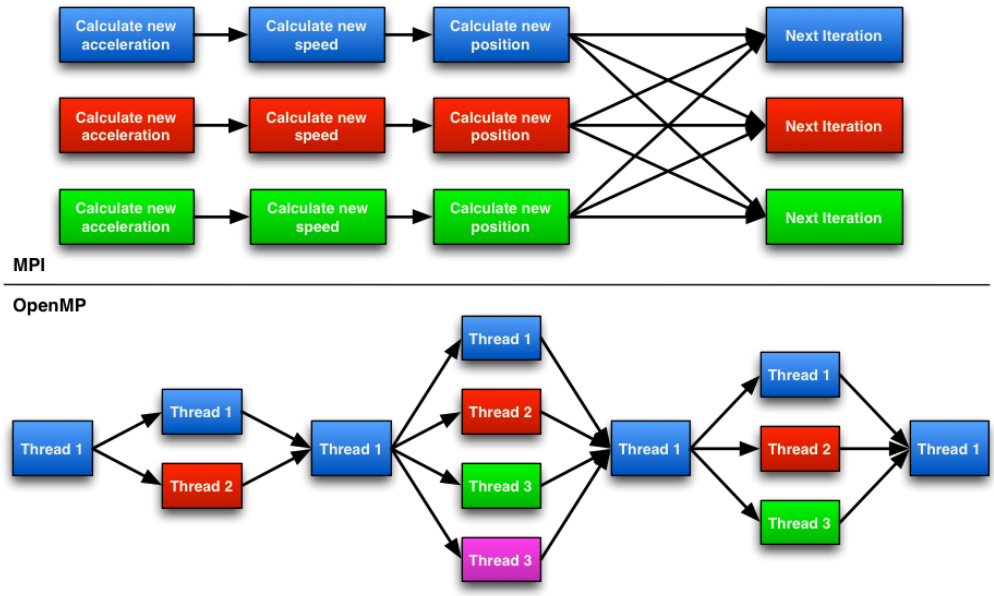

Figure 1: MPI *vs* OpenMP

1.2 OpenMP directives

OpenMP usage is enabled with the use of compiler directives, notably `#pragma`. These directives provide annotations that detail how OpenMP may parallelize nearby code. However, as they are compiler directives, a simple switch in your compiler can disable or enable OpenMP—a feature many find useful when testing.

All OpenMP directives start with `#pragma omp`; we will generally omit this prefix when discussing them. The simplest directive is simply `parallel`: it indicates the the block immediately below should be executed by all threads. Listing 1 gives the OpenMP equivalent of "Hello, world".

```
#include <stdio.h>
int main() {
  #pragma omp parallel
  {
    printf("Hello, world\n");
  }
  return 0;
}
```

Listing 1: OpenMP "Hello world".

When executed, Listing 1 will print out "`Hello, world`" *for each thread executing the program*. Generally, this is equal to the number of cores your system has. However, you can modify this by setting the `OMP_NUM_THREADS` environment variable.

```
$ export OMP_NUM_THREADS=3
$ ./a.out
Hello, world
Hello, world
Hello, world
```

Listing 2: Using OMP_NUM_THREADS to influence how many threads OpenMP utilizes.

However, OpenMP users generally utilize the `for` directive to indicate that the following loop should be parallelized. `for` will divide the loop bound by the number of threads, and manipulate the indices for each thread so that it executes a contiguous subset of those indices. Take Listing 3 as an example: this computes the linear combination of two arrays with their associated constants. If you removed the `#pragma`, the code would still be correct. With the pragma, each thread will execute `n / OMP_NUM_THREADS` iterations of the loop.

```
void lincomb(const float* x, const float* y, size_t n, float a, float b,
             float* out) {
  #pragma omp parallel for
  for(size_t i=0; i < n; i++) {
    out[i] = a*x[i] + b*y[i];
  }
}
```

Listing 3: OpenMP-parallelized for loop

Linear combination is an example of a parallel problem that is deemed *trivially parallelizable*. One might think of this as meaning that the iterations of the loop are independent: one does not need the side effects of iteration `i-1` to be able to compute iteration `i` (or any other iteration). These are the most fruitful computations to parallelize, because they do not require any synchronization.

1.3 Compilation and linking

OpenMP can be enabled or disabled simply by recompiling with different options. The GNU compiler turns OpenMP off by default. To switch it on, you need to compile *and link* with the `-fopenmp` flag. You should thus add this flag to both your `CFLAGS` and `LDFLAGS` variables in your makefile. To ensure your are correctly using OpenMP, you might want to use `print_nthreads` from program 4. If it says you are running on 1 thread, then something is wrong.

```
#include <stdio.h>
#include <omp.h>

static void print_nthreads(const char* program) {
  #pragma omp parallel
  {
    size_t n = omp_get_num_threads();
    if(omp_get_thread_num() == 0) {
      printf("%s is running with %zu threads.\n", program, n);
    }
  }
}

int main(int argc, char* argv[]) {
  (void)argc;
  print_nthreads(argv[0]);
  return 0;
}
```

Listing 4: Example program to print out the number of threads an OpenMP program is using.

As always, make sure you `module swap PrgEnv-pgi PrgEnv-gnu` on our Cray before compiling.

2 Hybrid MPI/OpenMP

Your assignment is to implement a hybrid MPI/OpenMP N-Body simulation. This simulation should use shared memory (OpenMP) for intra-node communication, and distributed memory (MPI) for inter-node communication. Figure 2 outlines the hybrid approach that you should implement.

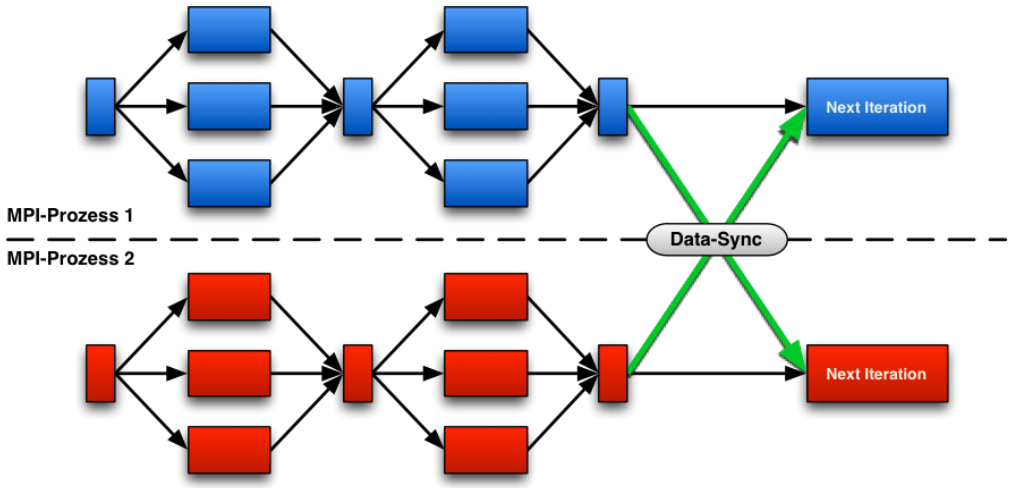

Figure 2: MPI/OpenMP hybrid parallel approach.

Be sure to test your simulation with multiple settings for both the number of nodes as well as the number of threads on each node. Your hybrid parallel simulation should produce the same results as your serial simulation, to within a factor of 0.0001 or so.

3 *sine qua non*

As always, submit your assignment by committing your code to your personal repository. Every assignment utilizes a new repository. The name for this assignment is **as4-*username***, where *username* is your name on our git server.

All assignments must include a makefile to compile your program. No makefile, no points!