15-418: Assignment 3:
OpenMP Programming on the Blacklight Supercomputer

Due: Tue Mar 6, 11:59PM

1 Overview

In this assignment you will optimize parallel programs using OpenMP in a shared-memory environment. In particular, you will have to parallelize two programs: (1) a simple prime number generator and (2) a variant of the Traveling Salesman Problem, called the Wandering Salesman Problem (WSP). The first program is very simple and is intended to get you familiar with OpenMP. The second (WSP) might appear simple at first, but exposes a number of subtle issues related to achieving high performance. You are allowed to work in groups of two people (one hand-in per group).

2 Environment Setup

For this assignment you will be using "Blacklight", a 4096-core supercomputer hosted at the Pittsburgh Supercomputing Center (PSC). There are two ways to access Blacklight to develop, compile, and test your code. The first is through the XSEDE web portal. You will need to login into your XSEDE account at `http://portal.xsede.org`. Once logged in, navigate to "MY XSEDE", then "SSH Terminal" and select "Blacklight" in the dropdown menu to initiate an SSH session. The second is via ssh to `blacklight.psc.teragrid.org` (we expect that the latter option will be far more convenient). To login via ssh you may first need to change your Blacklight password by following the instructions found here: `http://www.psc.edu/machines/sgi/uv/blacklight.php#password`

To get started:

- Download the Assignment 3 starter code and documentation from the course directory: `/afs/cs.cmu.edu/academic/class/15418-s12/assignments/asst3.tgz` All paths mentioned through the rest of this document are relative to the directory in which you extracted the asst3.tgz file.
- For more details about the Blacklight, please see: `http://www.psc.edu/machines/sgi/uv/blacklight.php`
- For more information on using Blacklight, please see: `tutorials/blacklight_tutorial.pdf`

3 Assignment

3.1 Program 1: Parallelizing a Prime Number Generator (10 pts)

The file `prog1_primes/primes.c` contains a uniprocessor program written in C for determining prime numbers. The program works by testing each odd number (up to a specified limit) for divisibility by all of the factors from 3 to the square root of that number. The algorithm is not very smart, but it is easy to parallelize.

Your task is to parallelize this existing algorithm in a number of different ways using the OpenMP API on the shared-memory Blacklight supercomputer. Further details on how to use OpenMP can be found under `tutorials/openmp.pdf`, along with example C programs in `examples`. Note that you *may not* change the underlying algorithm. Instead, you should view it as a workload that is to be parallelized.

The program takes two main parameters, which are read in from the command line:

P: the number of processors (`numProcs` in the code); and

N: the problem size (`size` in the code).

In addition, the program takes arguments that output all of the primes generated, either to a file or standard out. This should assist you in ensuring that the parallel version of the algorithm works correctly. Usage instructions can be seen by running `./primes -h`.

The main data structure of the program is an array which holds boolean values indicating whether the corresponding numbers are prime. The array only holds odd numbers, since no even numbers (except 2) are prime. In both the functions `Primes()` and `ParallelPrimes()`, the core code is executed 200 times to provide a reasonable run time. You should parallelize only the prime computation code not this outer loop! Also, you may not change the output data structure.

As you already learned from Assignment 1, good load balancing is crucial in achieving high performance for parallel workloads. You should be able to achieve very good speedups on this program (i.e. close to linear).

What you need to do:

1. Parallelize the prime number generation program. Insert timers at the beginning and ending of `Primes()` and `ParallelPrimes()` (measure the overall elapsed time of the 200 iterations) to measure performance.
2. Produce a report of your results. Your report should include the following items:
   (a) A brief discussion of the design and rationale behind your approach to parallelizing the algorithm.
   (b) The number of primes, and the last prime for N = 200,000; 1,000,000; 2,000,000; and 4,000,000.
   (c) A plot of speedup vs. N for $P = 16$ and N = 200,000; 1,000,000; 2,000,000; and 4,000,000. (A larger value of P is also acceptable, and may be more interesting.)
   (d) Explain the effect of problem size on observed speedup, if any.

3.2 Program 2: The Wandering Salesman Problem (50 pts)

In this program you will be solving the Wandering Salesman Problem (WSP) using the Branch-and-Bound technique. WSP is representative of combinatorial optimization problems. The Branch-and-Bound technique is an exhaustive evaluation technique that tries to make use of knowledge of the underlying problem to reduce the amount of computation.

The Wandering Salesman Problem (WSP). The objective of WSP is to find the shortest route for a traveling salesman so that the salesman visits every one of a set of cities exactly once. (Note: the salesman doesn't return home. This is the difference between the wandering salesman problem and the traveling salesman problem.) This is a hard combinatorial optimization problem since for N cities there are at most N! possible routes (we assume that the cities are fully connected).

The input to the problem is given in the form of a matrix. An element of the matrix, `d[i][j]` gives the distance between city i and city j. The input to your program is a file organized as follows:

```
N
d[1][2]
d[1][3] d[2][3]
d[1][4] d[2][4] d[3][4]
.
.
.
d[1][N] d[2][N] d[3][N] ... d[N-1][N]
```

where `N` is the number of cities, and `d[i][j]` is an integer giving the distance between cities i and j. The output from the program should be an ordered list of cities (numbers between 1 and N). Clearly, there are 2 equivalent permutations - either one is acceptable.

Branch-and-Bound Solutions to Combinatorial Optimization Problems. First, consider a simple exhaustive evaluation as a way of solving the WSP. One way you might think about evaluating every possible route is to construct a tree that describes all of the possible routes from the first city, as shown in Figure 1 for a four city example.

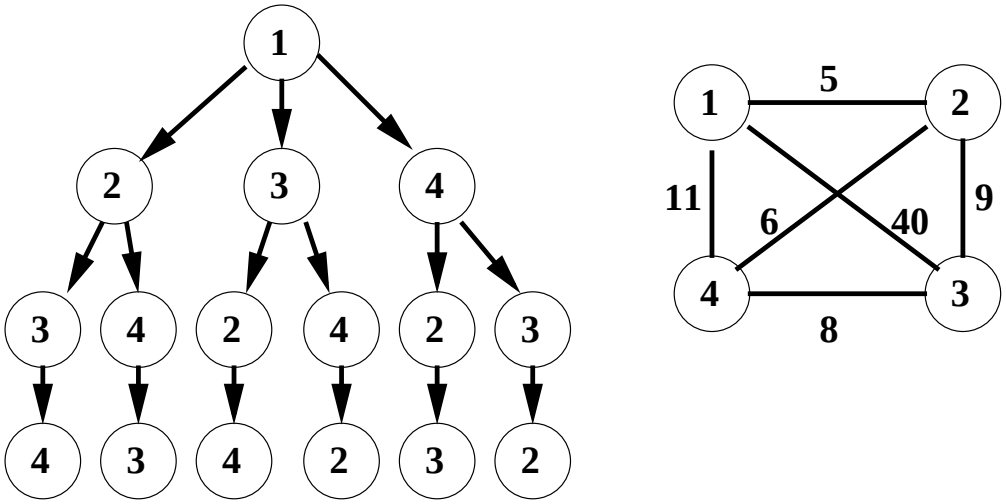

Figure 1: *A four-city wandering salesman problem: Left: a tree representing all unique paths through all four cities. Right: A graph providing distances between cities.*

For this WSP, `d[1][2]` = 5, `d[1][3]` = 40, `d[1][4]` = 11, `d[2][3]` = 9, `d[2][4]` = 6, `d[3][4]` = 8. All of the possible routes can be found by traveling from the root to a different leaf node three times, once for each unique path and then back to the root. For example, by taking the middle branch from the root node and the right branch after that, the route $1 \rightarrow 3 \rightarrow 4 \rightarrow 2$ is produced, and has a distance of 40+8+6 = 54. There are six unique root to leaf paths in this tree. Each possible route is represented twice, once in each direction.

A simple exhaustive evaluation of WSP for this example would then be to follow the six root to leaf paths and determine the total distance for each path by adding up the distances indicated by each edge in the tree. The route with the smallest distance is then chosen.

A better way to traverse each tree is to do it recursively. Here the summation of the earlier parts of each route is not repeated every time that route portion is reused in several routes. The Branch-and-Bound approach uses this type of problem formulation, but with some added intelligence. It uses more knowledge of the problem to prune the tree as much as possible so that less evaluation is necessary. The basic approach works as follows:

1. Evaluate one route of the tree in its entirety, (say $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$) and determine the distance of that path. Call this distance the current "bound" of the problem. The bound for this path in the above tree is $5 + 9 + 8 = 22$.
2. Next, suppose that a second path is partially evaluated, say path $1 \rightarrow 3$, and the partial distance, 40, is already greater than the *bound*. If that is the case, then there is no need to complete the traversal of any part of the tree from there on, because all of those possible routes (in this case there are two) must have a distance greater than the bound. In this way the tree is *pruned* and therefore does not have to be entirely traversed.
3. Whenever any route is discovered that has a better distance than the current bound, then the bound is updated to this new value.

The Branch-and-Bound approach always remembers the best path it has found so far, and uses that to prevent search down parts of the tree that couldn't possibly produce better routes. You can see that for larger trees, this could result in the removal of many possible evaluations.

What you need to do:

1. (40 points) Implement a parallel Branch-and-Bound program for the WSP, using OpenMP on Blacklight. The objective is to obtain the best speedup possible. Your program should accept the command-line arguments `-p` and `-i` to specify the number of processors used and the input file respectively. It should also use the input file format described above. (File loading and command-line parsing are already implemented for you in the starter code).
2. (10 points) Produce a brief report describing and analyzing your solution. Your report should include the following items:
   - A brief description of how your program works. Describe the general program flow and all significant data structures. How is the problem decomposed? How is the work assigned to threads? Where is the communication and synchronization?
   - Give your solution to the problem given in `prog2_wsp/input/distances`. This file contains a 17 city problem. (For debugging purposes, you may want to use some of the smaller input files included in the same directory. The city locations corresponding the distance files are provided in the corresponding 'city' files. You can use them for visualization if needed.)
   - Execution time and speedup (both total and computation) for 1, 2, 4, 8, 16, 32, 64, 128 and 256 processors on Blacklight. See Section 4 for definitions of total and computation time.
   - Discuss the results you expected and explain the reasons for any non-ideal behavior you observe. In particular, if your program does not achieve perfect speedup, explain why. Is it due to work imbalance? Communication/synchronization overhead? Performing redundant (or unnecessary) work? Is it possible to achieve better than perfect speedup? Provide measurements to back up your explanations.

4 Measuring Performance

While it may be helpful to compile the `-g` flag when you are debugging your code, **please be sure to use the `-O3` flag to generate any programs that you will be timing!** There can be significant differences in performance between different levels of compiler optimization, and we are only interested in the speedup of optimized code. The provided Makefile generates two executables `wsp` and `wsp_debug` (corresponding to an optimized and debug build of the code).

To evaluate the performance of your parallel programs, measure the following times using the provided `gettime()` function which returns the current time in seconds:

1. *Initialization Time:* the time required to do all the sundry initialization, read the command line arguments, and create the separate processes. Start timing when the program starts, and end just before the main computation starts.
2. *Computation Time:* this is strictly the time to compute the answer. Start timing when the main computation starts (after all the processes have been created), and finish when the answer has been calculated.

Note that: *Total Time* = *Initialization Time* + *Computation Time*. *Speedup* is calculated as $\frac{T_1}{T_p}$, where T_1 is the time for one processor, and T_p is the time for P processors. *Computation Speedup* uses only Computation Time, and *Total Speedup* uses the Total Time.

5 Performance Analysis

The goal of this assignment is for you to think carefully about how real-world effects in a cc-NUMA machine limiting your speedup, and how you can improve your program to get better performance. If your performance is disappointing, then it is likely that you can restructure your code to make things better. We are especially interested in hearing about the thought process that went into designing your program, and how it evolved over time based on your experiments.

5.1 Hand-in Instructions

Hand-in directories have been created at:
`/afs/cs.cmu.edu/academic/class/15418-s12/handin/asst3/<ANDREWID>/`.

Copy your `asst3/` directory into your hand-in directory. If working in a group, copy your `asst3/` directory in either group member's hand-in directory. The written answers should be in `asst3/writeup.pdf`. **Your code should compile and run on Blacklight without any modifications!** This means that we should be able to `make` and execute your programs without manual intervention.

5.2 Resources And Notes

- Blacklight
  - Short tutorial on using Blacklight: `tutorials/blacklight_tutorial.pdf`,
  - Detailed info on Blacklight: `http://www.psc.edu/machines/sgi/uv/blacklight.php`,
- OpenMP
  - Short tutorial on OpenMP: `tutorials/openmp.pdf`,
  - NCSA online course on OpenMP: `http://www.citutor.org/login.php?course=24` (you need to register for a free account)
  - OpenMP tutorial from Lawrence Livermore National Laboratory: `https://computing.llnl.gov/tutorials/openMP/`
  - Official OpenMP website: `http://openmp.org/`