ARIZONA STATE UNIVERSITY

CSE 430 — **Operating Systems** — Fall 2012

SLN 71250

Instructor: Dr. Violet R. Syrotiuk

Project #1

Available Tuesday 09/04/2012; due Tuesday 10/09/2012

This project has three parts:

1. In the first part, the goal is to familiarize yourself with the tools of **Intel® Parallel Studio**.
2. In the second part, the goal is to learn to use **OpenMP** directives and **Intel® Parallel Studio** tools to develop a parallel program from your serial program to solve the TSP in a directed complete graph.
3. Finally, in the third part, the goal is to get experience using `Saguaro`, a cluster with machines having large numbers of cores, to run experiments, collect and plot data, and interpret the results. You to must register for an account on `Saguaro`; this takes some time to set up so register early (see §4).

1 Intel® Parallel Studio

Intel® Parallel Studio XE is a parallel software development suite that combines a compiler, performance and parallel libraries, error checking, code robustness, and performance profiling tools into a single suite. In our BYENG 214 lab, each Windows machine has **Intel® Parallel Studio XE 2011** integrated with **Microsoft® Visual Studio 2010** installed on **Microsoft® Windows 7**.

Intel® Parallel Studio XE includes:

- **Intel® Composer XE**: Optimizing compilers and high-performance libraries.
- **Intel® Inspector XE**: Powerful thread and memory error checker.
- **Intel® VTune Amplifier XE**: Advanced performance profiler.

We will only learn enough functionality of each of these tools to get a taste of their usefulness. There are a number of "Getting Started" tutorials and videos on each of these tools. Links to some of them are available on our Blackboard site. The tutorials are also accessible from within Visual Studio.

While 30 day evaluation versions of **Intel® Software Development Products** are available for free download for Windows and Linux, you are on your own if you choose to use them. Go to

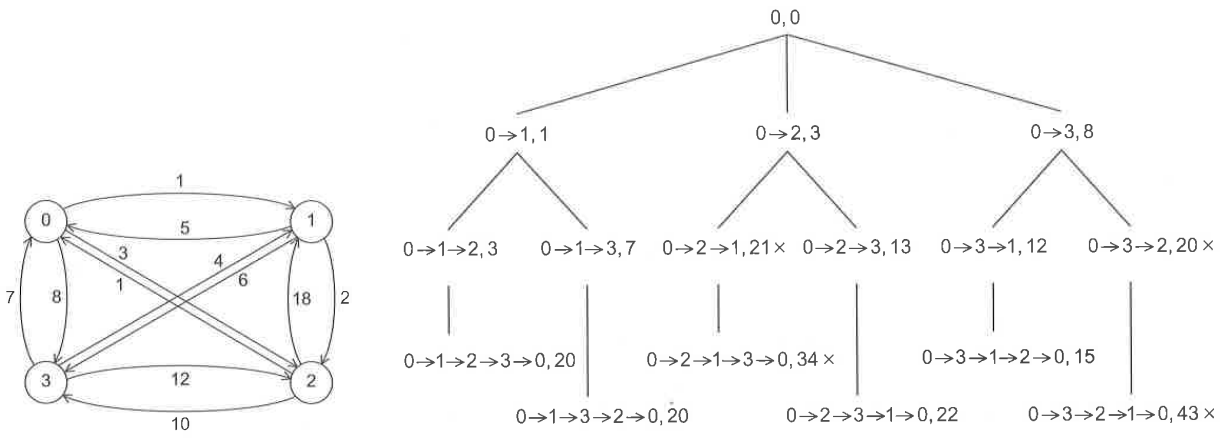
`http://software.intel.com/en-us/articles/intel-software-evaluation-center/`

for system requirements and availability.

2 TSP in a Directed Complete Graph

Many problems can be solved using a tree search. As an example, consider the *Travelling Salesperson Problem* (TSP). In the TSP, a salesperson must visit n cities. Modelling the problem as a complete digraph $G = (V, E)$ with $|V| = n$ vertices, the salesperson wishes to make a *tour* (Hamiltonian cycle) visiting each city exactly once and finishing at the city s/he starts. There is an integer cost $c(i, j)$ to travel from city i to city j, and the salesperson wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour.

Unfortunately, the TSP is known to be NP-complete. From a practical standpoint, that means that there is no algorithm known for solving it that, in all cases, is significantly better than exhaustive search. The



(a) A four city TSP.

(b) Search tree for the four-city TSP.

Figure 1: An instance of a four-city TSP.

number of possible solutions to TSP grows exponentially as the number of cities is increased. For example, if one additional city is added to an n-city problem, it increases the number of possible solutions by a factor of $n+1$. Furthermore, if a solution to TSP were found that is significantly better in all cases than exhaustive search, then there are hundreds of other hard problems for which efficient solutions could be found.

So how can TSP be solved? One solution is a form of tree search. The idea is that in searching for solutions, a tree is built. The leaves of the tree correspond to tours, and the interior nodes correspond to "partial" tours — routes that have visited some, but not all, of the cities.

Each interior or leaf node of the tree has an associated cost, specifically the cost of the partial or complete tour, respectively. The cost of an interior node may be used to eliminate some of its descendants. Thus, the cost of the best tour found so far should be maintained, and, if a partial tour could not possibly lead to a less expensive complete tour, there is no reason to search the children of that node.

Consider an example of an $n = 4$-city TSP in Figure 1(a), represented as a weighted, directed complete graph. In this example, the vertices V of the graph G correspond to the cities, and the weights on the edges $(i, j) \in E, 0 \leq i, j \leq n-1, i \neq j$, correspond to the cost of travelling from city i to city j. For example, the cost of going from city 0 to city 1 is 1 and the cost of going from city 1 to city 0 is 5 (i.e., the cost of travelling between two cities need not be the same in each direction).

If we choose vertex 0 as the starting city, then the initial partial tour consists only of vertex 0 with a cost of 0. Thus, the root of the search tree in Figure 1(b) has the partial tour consisting only of the vertex 0 with cost 0. From vertex 0, the vertices 1, 2, or 3 may be visited, giving three two-city partial tours with costs 1, 3, and 8, respectively. Continuing, there are six three-city partial tours, and since there are only four cities, once three of the cities are chosen, the fourth completes the tour.

To find the least-cost tour the tree must be searched. There are many ways to do this, but one of the most commonly used approaches is *depth-first search* (DFS). Using DFS on the example starts at the root and branches left until the leaf labelled $0 \to 1 \to 2 \to 3 \to 0$ with cost 20 is reached. Then, DFS backs up to the node labelled $0 \to 1$, since it is the deepest ancestor node with unvisited children, and branches down to reach the leaf labelled $0 \to 1 \to 3 \to 2 \to 0$ also with a cost of 20. Continuing, DFS backs up to the root and branches down to the node labelled $0 \to 2$. When its child, labelled $0 \to 2 \to 1$ with a cost of 21, is visited there is no reason to further explore this subtree because a complete tour with cost less than 21 has already been found. Therefore, DFS backs up to $0 \to 2$ and branches down to its remaining unvisited child. Continuing in this fashion the least-cost tour by DFS is found to be $0 \to 3 \to 1 \to 2 \to 0$ with cost 15.

The simplest formulation of DFS uses recursion. Since function calls are expensive, recursion can be slow. It also has the disadvantage that at any given instant of time only the current tree node is accessible.

This could be a problem in trying to parallelize tree search by dividing tree nodes among threads. *Therefore, in this project, your starting point is to write a serial program that implements a nonrecursive DFS to solve the TSP. Then you will develop a parallel program from your serial program using OpenMP.* Assignment #2 and example programs that we discuss in class will help you develop your parallel program.

3 Requirements

1. Write a serial nonrecursive C/C++ program to solve the TSP in a weighted directed complete graph $G = (V, E)$, starting from vertex 0. Your program should read a file containing the graph in the following format:

   (a) The first line contains $n = |V|$, the number of vertices in the directed complete graph G.

   (b) Then $\frac{n(n-1)}{2}$ lines follow, one for each edge (i, j), $0 \leq i, j \leq n-1$, $i \neq j$. Each line has the format:

   $$i \quad j \quad cost(i, j)$$

   where $cost(i, j) \geq 0$ is an integer giving the weight of the edge from i to j. An adjacency matrix representation of the graph is recommended.

2. The output of your program should be a solution to the TSP. Print the least-cost tour, and its cost. If there is more than one tour of least-cost print them all.

3. Now, parallelize your serial nonrecursive program by introducing OpenMP directives. Your objective is to obtain the best speedup you can. A bonus will be given for the program obtaining the best speedup.

4. Ultimately, your programs must run on `Saguaro`. Therefore, it is recommended that you develop the programs for this project using the Intel® compiler in Visual Studio or a supported version of `gcc` because these compilers are available on `Saguaro`. (See the `module` command on `Saguaro` to determine the versions of `gcc`, and other C/C++ compilers, supported.)

A program that generates sample input files will be provided on Blackboard; you may use these to test the correctness of your program.

4 Experimentation on `Saguaro`

You must register for an account on `Saguaro` by clicking on the "Getting Started" link on `a2c2.asu.edu`. From there, you can link to your ASUrite id and list me as your affiliated faculty with CSE430 as your project.

4.1 Requirements

1. Copy both your serial and parallel C/C++ programs onto `Saguaro` and compile them.

2. Submit a series of batch jobs for graphs of increasing size (number of vertices) on one of the compute nodes with at least 12 cores. Collect the run time of your serial and parallel program for each graph. Be sure to use the same type of hardware for all of your experiments!

3. Plot the run time of your serial and parallel programs as a function of size and analyze the results. Use your results to answer the following questions:

   (a) What speed-up, if any, is obtained by your parallel algorithm?

   (b) What size does the graph need to be before a speed-up is observed?

5 Hand-in instructions

Submit electronically, before 9:00am on 10/09/2012 using the submission link on Blackboard for Project #1, a zip[1] file named `FirstName-LastName.zip` containing the following items:

Design and Analysis (30%): Provide a description of the methodology you followed to produce a correct parallel version of your serial program. Specifically:

1. Discuss how you assigned subtrees to threads, how you did load balancing among threads, and how you stored and updated the best tour.
2. Describe any errors that arose (e.g., race conditions) and how you solved them.
3. Describe the activities you performed to improve the speed-up of your parallel program.

In addition, include the results of your analysis, i.e., the graphical results and answers to the questions posed in Point 3 of the Requirements in §4.1.

Implementation (50%): Provide your documented C/C++ source code for both your serial and parallel solution to the TSP in a complete graph. In addition, you must provide scripts to run your serial and parallel programs on the test input files provided; more details about the test input files will be provided in the discussion group closer to the due date.

You may write your code only in C or C++. You must not alter the requirements of this project in any way.

Correctness (20%) Your programs **must** run on `Saguaro` using the scripts you provide. Our TA will test them there.

[1] **Do not** use any other archiving program except `zip`.