Christopher Pac
CSCI-GA.3033-009
Multicore Processors: Architecture & Programming
Lab Assignment #3

Section 1 – Compilation

gcc version 4.8.2 [module load gcc-4.8.2]

```
g++ -std=c++0x -D MAX_CITIES=x -D MY_INIT_THREADS=y -D PRINT_T_LOAD -o vBaB
finalBaB.cpp -lpthread
```

Where *x* is the maximum number of cities and *y* is the initial number of threads to be used (number of threads can also be specified as a second command line argument). Omitting `MAX_CITIES` flag will result in 10 cities being used as the maximum and omitting `MY_INIT_THREADS` will result in 4 threads (plus main) being used. Specify `PRINT_T_LOAD` flag to have each thread print the amount of work preformed.

sequential version:
```
g++ -std=c++0x -D MY_INIT_THREADS=0 -o sequential finalBaB.cpp –lpthread
```

parallel version:
```
g++ -std=c++0x -o parallel finalBaB.cpp –lpthread
```

Section 2 – Which programming language did you pick? and why?

For this problem I used c++ with pthreads. I used c++ for the convenience of being able to use the Standard Template Library (STL) and out of STL I used the stack container class.

I used pthreads because I'm more familiar with it and because pthreads was more natural tool for my TSP algorithm. In my solution I don't use `for` loops and the solution is based on semi-pool of threads like scheme. Therefore, I needed a mechanism to arbitrarily put threads to sleep and wake them up later, something which OpenMP does not provide. Furthermore, OpenMP is designed for programs that will have fixed number of threads; that is once a certain parallel task starts the number of threads cannot be dynamically changed. In my current implementation of TSP I use fix number of threads but it would be trivial to make the change to spawn additional threads in the middle of tsp computation. Lastly, pthreads has better performance compared to OpenMP as we have seen in the previous lab. These reasons for using pthreads will become more evident as I explain my solution in the next section.

Section 3 – How did you parallelize your program?

My sequential algorithm uses the branch and bound technique.

`put node 0 on a stack` (I used stack not queue to do a depth first analysis; using queue (i.e. breadth first) was much slower)

```
while(stack is not empty)
      node = stack.pop
if (node is Leaf node)
      if (node.distance < bestDistance)
            bestDistance = node.distance
            bestPath = node.path
else
      if (node.distance < bestDistance)
            stack.push( all node.children < bestDistance )
```

For the parallel version I took the sequential branch and bound algorithm and made few changes. Each thread would still run the sequential version but when its own stack is empty it will get items from a shared stack. If both are empty then the thread will increment a shared idle counter and call `pthread_cond_wait`. When another thread sees that the idle counter is positive, it will push left child onto its own stack and all remaining children onto a shared stack and then it will call `pthread_cond_broadcast`. When the idle thread is awakened it will get an item from the shared stack, decrement the idle counter, and continue. It will also check if the idle counter is equal to the number of threads, if so it will finish (without decrementing the idle counter).

Initially the idle counter is set to the value of all threads (minus the main), then the threads are created and they will either `_cond_wait` or find an item in the shared stack depending on timing. This is because after the threads are created the main thread starts, adds the node 0 to its local stack and then discovers that there are idle threads thus it will add items to the shared stack and awake any idle threads.

Furthermore, each thread uses global *bestDistance* to limit its path. Like the idle counter, these global values are not mutex locked when *only* reading will be performed. The hardware will take care of concurrency and it is not required to read the *most* recent versions of these values as long as they will eventually be updated for each thread. Only efficiency decreases when we don’t have the latest values not correctness.

The load balance of this algorithm is quite optimal. I’ve found that for higher number of cities (>5) the amount of work each thread did was on the same order of magnitude as all other threads. Here are two examples with 14 cities executing with 4 and 8 threads:

```
Thread: 4 has done this much work 60419836
Thread: 2 has done this much work 60086693
Thread: 3 has done this much work 61450157
Thread: 1 has done this much work 60117775
Thread: 0 has done this much work 61238333
Best path: 0 1 8 10 13 9 2 4 6 11 12 7 5 3
Distance: 59
real  0m16.538s

Thread: 3 has done this much work 38260448
Thread: 1 has done this much work 37846709
Thread: 0 has done this much work 38575453
Thread: 2 has done this much work 39466423
Thread: 7 has done this much work 37812478
Thread: 6 has done this much work 38014403
Thread: 4 has done this much work 39255040
Thread: 5 has done this much work 39016020
Best path: 0 1 8 10 13 9 2 4 6 11 12 7 5 3
Distance: 59
real  0m11.069s
```

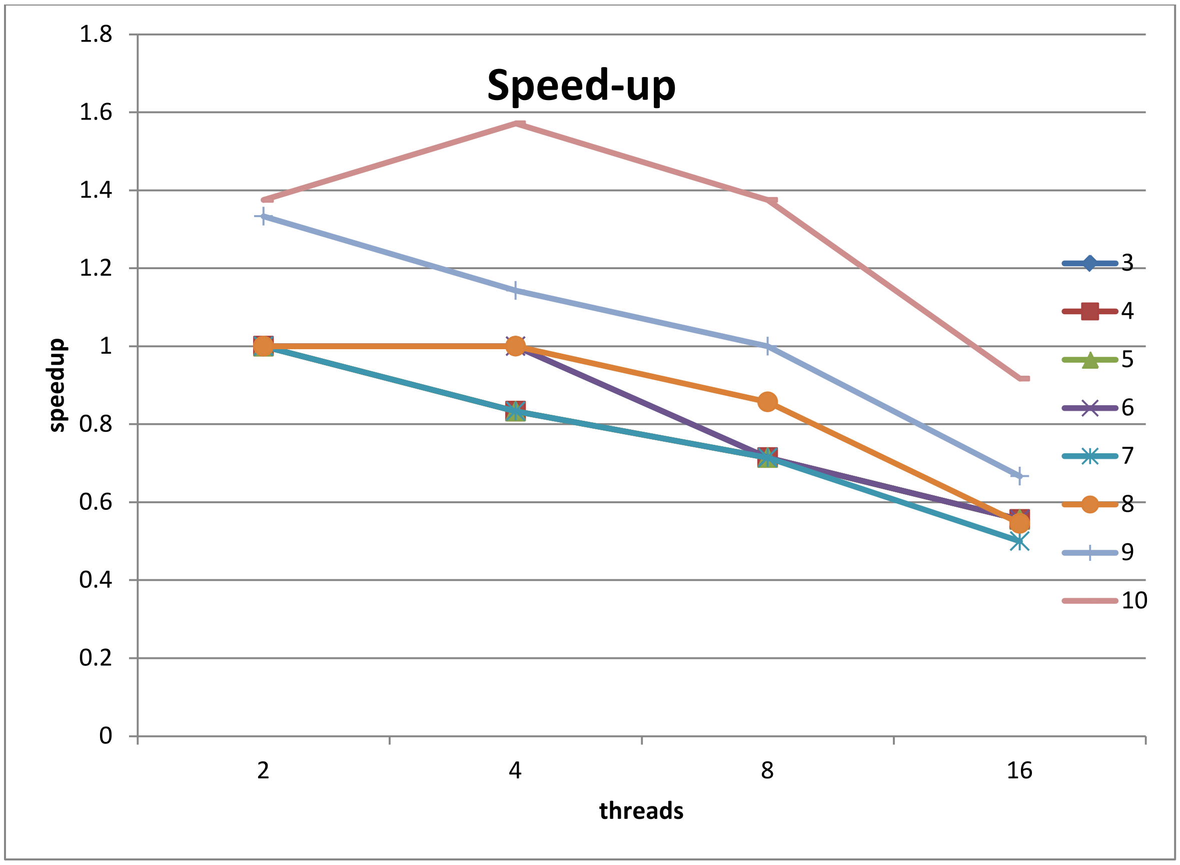
I’ve also added an optimization where I use a greedy algorithm to find the initial *bestDistance* upper bound value (this is done for both sequential and parallel version).

This solution has the nice benefit of being semi-pool of threads like where with additional performance/hardware check we *could* dynamically add more threads at run time.

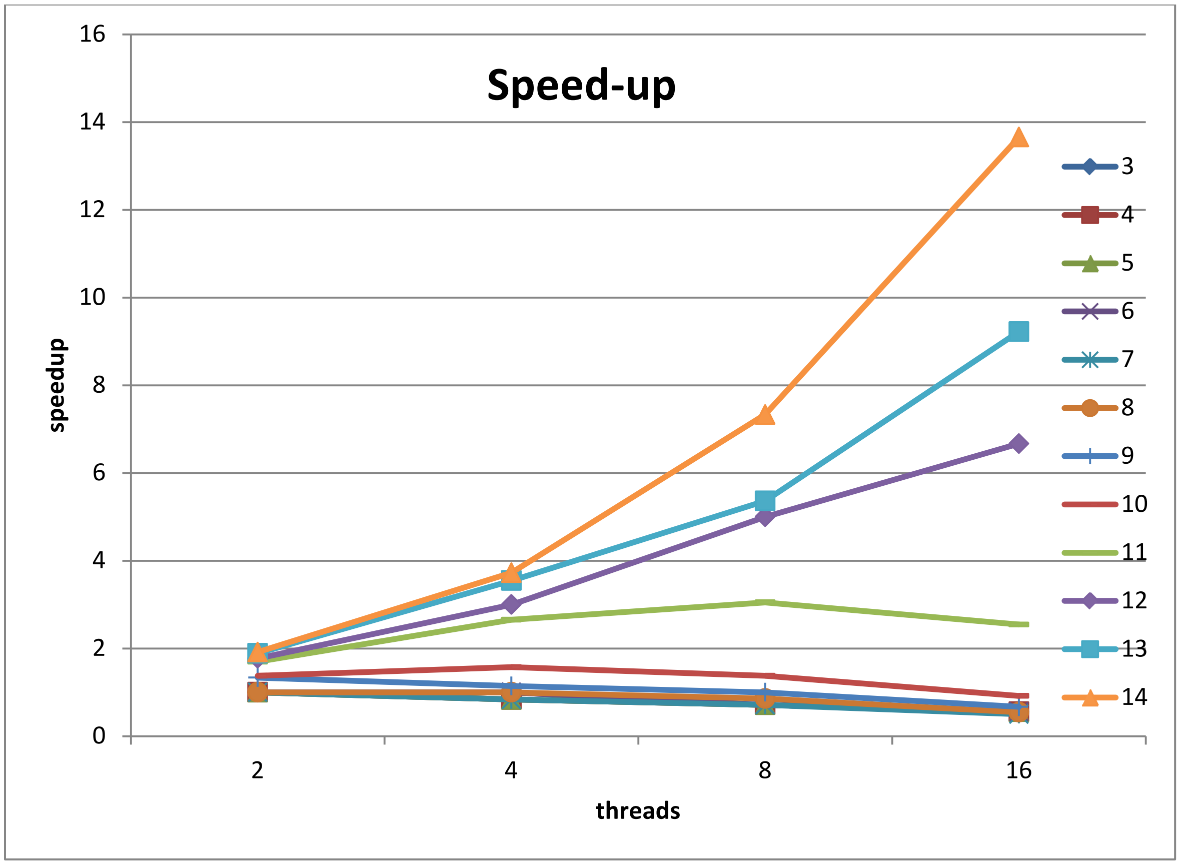
Here is the pseudocode for the parallelized algorithm:

```
Thread 0: put node 0 on a local stack
while(true)
      if (local stack not empty)
            node = local_stack.pop
      else if (global stack not empty)
            node = global_stack.pop
      else
            increment idle flag
            conditional wait broadcast
            while(global stack empty and idle flag != number of threads)
                  conditional wait
            if (idle flag == number of threads)
                  return NULL
            decrement idle flag
if (node is Leaf node)
      if (node.distance < bestDistance)
            bestDistance = node.distance
            bestPath = node.path
else
      if (node.distance < bestDistance)
            local_stack.push (first node.child < bestDistance )
            if (idle counter)
                  global_stack.push (remaining node.children < bestDistance )
                  conditional wait broadcast
            else
                  local_stack.push (remaining node.children < bestDistance )
```

Section 4 – Speedup graph

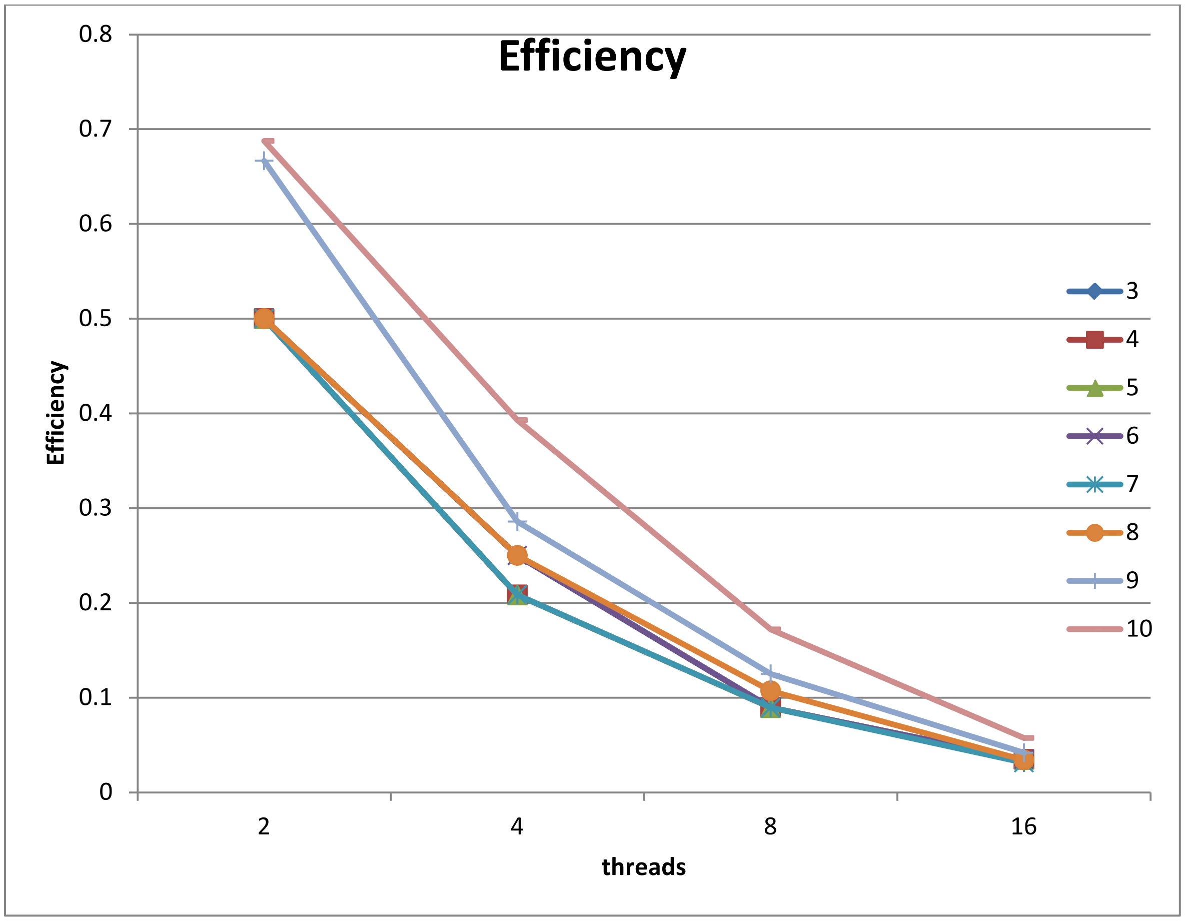


Graph 1 - Speedup: Speedup graph for **3 to 10** cities and each line is the number of cities.

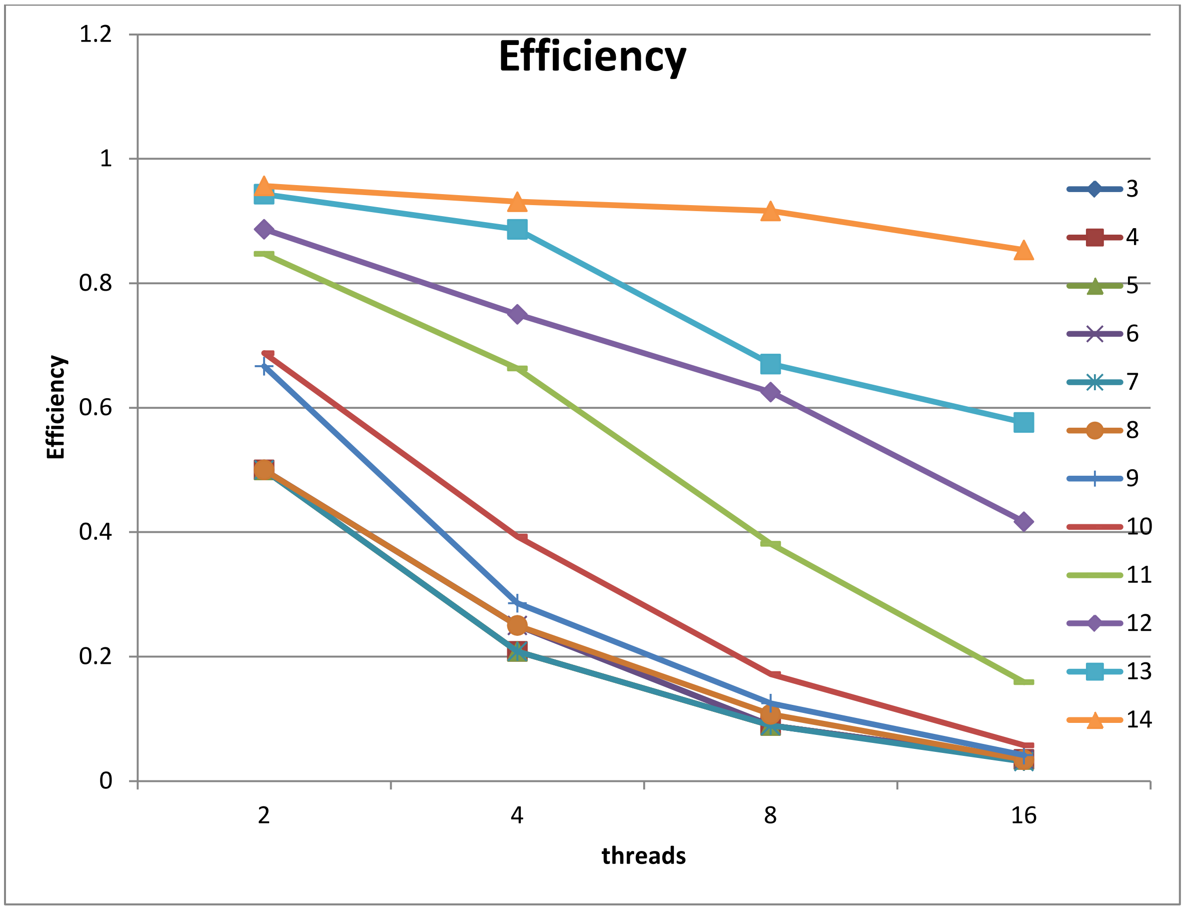


Graph 2 - Speedup: Speedup graph for **3 to 14** cities and each line is number of the cities.

Section 5 – Efficiency/Scalability graph



Graph 3 - Efficiency: Efficiency graph for **3 to 10** cities and each line is the number of cities.



Graph 4 - Efficiency: Efficiency graph for **3 to 14** cities and each line is the number of cities.

Section 6 – Speedup graph analysis

I've included two graphs for speedup because the graph with less than 10 cities does not show an adequate behavior for the traveling salesman problem. Simply stated, the cost of the thread overhead is not sufficiently amortized for small number of cities. A brute force algorithm with less than 9 cities will actually outperform the dynamic programming and branch and bound algorithms. Furthermore, the sequential version of the branch and bound algorithm is quite efficient and for less than 9 cities it does better than the parallel versions. This is simply because for less than 9 cities there is just not enough work to compensate for the thread management, communication, and synchronization overhead.

The above explanation leads directly to the results shown in Graph 1 (10 cities). For tasks with less than 9 cities there is no speedup over the sequential version and as we add additional threads the overhead of managing them increases and causes the speedup to decline. The additional threads do not have sufficient amount of work to perform. Depending on timing of thread scheduling, some threads will finish without performing any work when the problem size is small and large number of threads are created. This behavior is expected and can be observed with specifying the `PRINT_T_LOAD` flag during compile time. In such a case, the cost of the idle thread was incurred without any benefit.

Once the problem size increases passed 8 cities, we observe speedup over the sequential version (Graph 1). The 9 cities task shows increased speedup with 2 threads and 4 threads. However, as we add more threads passed 2 the speedup decreases as expected due to overhead. The 10 cities task shows speedup with 2 threads but interestingly the speedup increases again for 4 threads. Finally, there is enough work for 4 threads to offset their cost and cause additional speedup.

Graph 2 (14 cities) shows a more accurate picture of the parallel branch and bound algorithm. First it is clear that there is significant speedup over the sequential version and the speedup increases as we increase the problem size and the number of threads. This is again expected since the traveling salesman problem is a CPU bound problem. For 14 cities task we see an exponential speedup curve as we increase the number of threads. This is very encouraging and demonstrates that we have a good load balance and communication and synchronization are efficient. In fact this is an ideal speedup model for this kind of a problem and it means that the algorithm scales very well. This efficiency could not have been observed with insufficient problem size.

Section 7 – Efficiency/Scalability graph analysis

Once again I have provided two graphs to show efficiency as the problem size increases passed the point where additional threads have adequate work. As stated before, the higher the number of cities the better the thread overhead can be amortized.

In Graph 3 (10 cities) we can see that efficiency decreases as the number of threads increase for all work sizes. For the work sizes consisting of less than 9 cities the efficiency curves are *overlapping* which indicates that the solution to the problem is at the very least weakly scalable. That is, as the problems size increases and the thread count increases efficiency stays the same at that point.

Once the problem size increases to 9 or more cities, efficiency per work increases but it still decreases as the number of threads increase. This indicates that the solution is not yet exhibiting strong scalability but it’s doing better that weak scalability.

This trend continue in Graph 4 (14 cities) where as the work increases efficiency for the same number of threads increases. The line showing the efficiency of 14 cities task approaches horizontal line. This means that efficiency stays the same while the number of threads increases for the same work size (14 cities) indicating strong scalability.