**UNIT V**

**PARALLEL PROGRAM DEVELOPMENT**

Case studies - n-Body solvers – Tree Search – OpenMP and MPI implementations and comparison.

**5. INTRODUCTION**

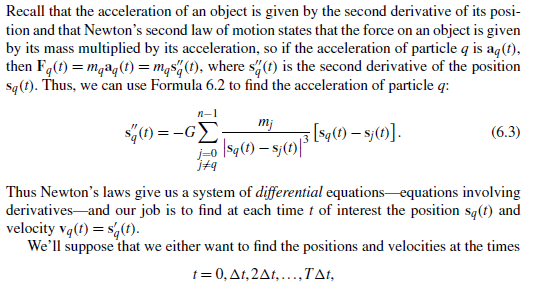
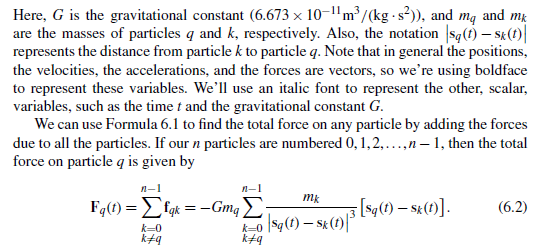
1. In this unit we’ll look at a couple of larger examples: solving n-body problems and solving the traveling salesperson problem.
2. For each problem, we’ll start by looking at a serial solution and examining modiﬁcations to the serial solution. As we apply Foster’s methodology, we’ll see that there are some striking similarities between developing shared- and distributed-memory programs.

**5.1 n-BODY SOLVERS**

1. In an n-body problem, we need to ﬁnd the positions and velocities of a collection of interacting particles over a period of time.
2. For example, an astrophysicist might want to know the positions and velocities of a collection of stars, while a chemist might want to know the positions and velocities of a collection of molecules or atoms.
3. **An n-body solver is a program that ﬁnds the solution to an n-body problem by simulating the behavior of the particles**. The input to the problem is the mass, position, and velocity of each particle at the start of the simulation, and the output is typically the position and velocity of each particle at a sequence of user-speciﬁed times, or simply the position and velocity of each particle at the end of a user-speciﬁed time period.

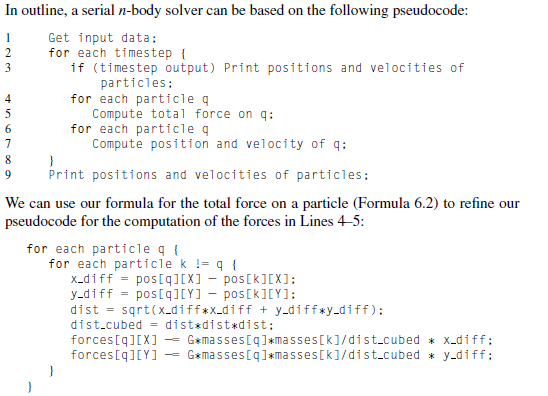
**The problem**

For the sake of explicitness, let’s write an n-body solver that simulates the motions of planets or stars. We’ll use Newton’s second law of motion and his law of universal gravitation to determine the positions and velocities. Thus, if particle q has position sq(t) at time t, and particle k has position sq(t), then the force on particle q exerted by particle k is given by

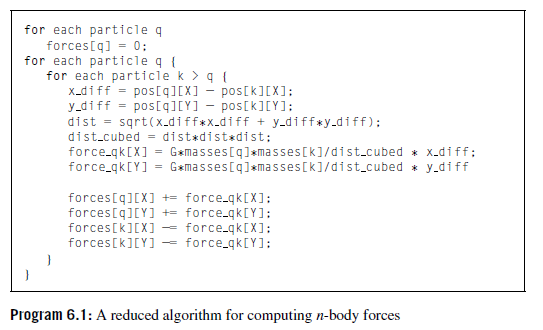


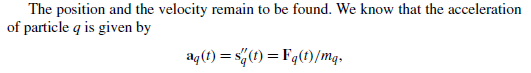
or,moreoften,simplythepositionsandvelocitiesattheﬁnaltimeT1t.Here, 1t and T are speciﬁed by the user, so the input to the program will be n, the number of particles, 1t, T, and, for each particle, its mass, its initial position, and its initial velocity.

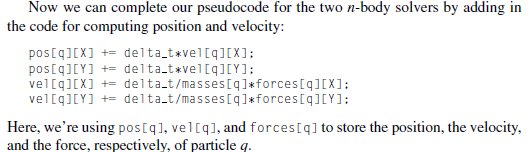
**Two serial programs**



Here, we’re assuming that the forces and the positions of the particles are stored as two-dimensional arrays, forces and pos, respectively. We’re also assuming we’ve deﬁned constants X = 0 and Y = 1. So the x-component of the force on particle q is forces[q][X] and the y-component is forces[q][Y]. Similarly, the components of the position are pos[q][X] and pos[q][Y]. (We’ll take a closer look at data structures shortly.)

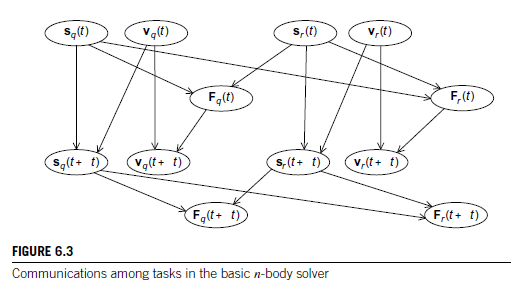


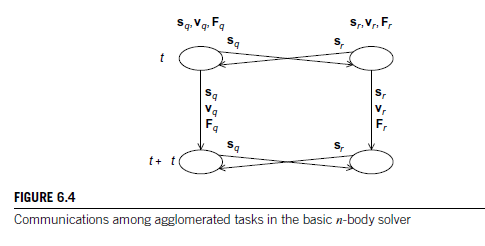




**Parallelizing the n-body solver**

1. Let’s try to apply Foster’s methodology to the n-body solver. Since we initially want lots of tasks, we can start by making our tasks the computations of the positions, the velocities, and the total forces at each timestep.
2. In the basic algorithm, the algorithm in which the total force on each particle is calculated directly from Formula 6.2,





**5.1.1 Parallelizing the basic solver using OpenMP**

How can we use OpenMP to map tasks/particles to cores in the basic version of our n-body solver? Let’s take a look at the pseudocode for the serial program:

for each timestep

{

if (timestep output)

Print positions and velocities of particles;

for each particle q

Compute total force on q;

for each particle q

Compute position and velocity of q;

}

The two inner loops are both iterating over particles. So, in principle, parallelizing the two inner for loops will map tasks/particles to cores, and we migh try something like this:

for each timestep

{

if (timestep output)

Print positions and velocities of particles;

# pragma omp parallel for

for each particle q

Compute total force on q;

# pragma omp parallel for

for each particle q

Compute position and velocity of q;

}

We may not like the fact that this code could do a lot of forking and joining of threads, but before dealing with that, let’s take a look at the loops themselves: we need to see if there are any race conditions caused by loop-carried dependences. In the basic version the ﬁrst loop has the following form:

# pragma omp parallel for

for each particle q

{

forces[q][X] = forces[q][Y] = 0;

for each particle k != q

{ x\_diff = pos[q][X] − pos[k][X];

Y\_diff = pos[q][Y] − pos[k][Y];

dist = sqrt(x\_diff∗x\_diff + y\_diff∗y\_diff);

dist\_cubed = dist∗dist∗dist;

forces[q][X] −= G∗masses[q]∗masses[k]/dist\_cubed ∗ x\_diff;

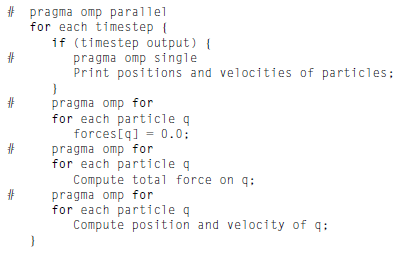
forces[q][Y] −= G∗masses[q]∗masses[k]/dist\_cubed ∗ y\_diff;

}

}

**Parallelizing the reduced solver using OpenMP**

The reduced solver has an additional inner loop: the initialization of the forces array to 0. If we try to use the same parallelization for the reduced solver, we should also parallelize this loop with a for directive. What happens if we try this? That is, what happens if we try to parallelize the reduced solver with the following pseudocode?



Parallelization of the initialization of the forces should be ﬁne, as there’s no dependence among the iterations. The updating of the positions and velocities is the same in both the basic and reduced solvers, so if the computation of the forces is OK, then this should also be OK. How does parallelization affect the correctness of the loop for computing the forces? Recall that in the reduced version, this loop has the following form:

# pragma omp for

for each particle q

{

force\_qk[X] = force\_qk[Y] = 0;

for each particle k > q

{

x\_diff = pos[q][X] − pos[k][X];

y\_diff = pos[q][Y] − pos[k][Y];

dist = sqrt(x\_diff∗x\_diff + y\_diff∗y\_diff);

dist\_cubed = dist∗dist∗dist;

force\_qk[X] = G∗masses[q]∗masses[k]/dist\_cubed ∗ x\_diff;

force\_qk[Y] = G∗masses[q]∗masses[k]/dist\_cubed ∗ y\_diff;

forces[q][X] += force\_qk[X];

forces[q][Y] += force\_qk[Y];

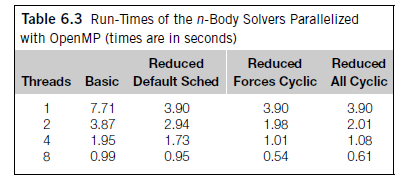
forces[k][X] −= force\_qk[X];

forces[k][Y] −= force\_qk[Y];

}

}

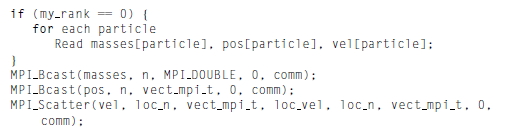
**Evaluating the OpenMP codes**

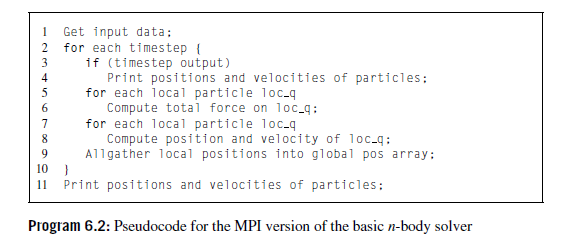
****

**5.1.2 Parallelizing the basic solver using MPI**

1. With our composite tasks corresponding to the individual particles, it’s fairly straight forward to parallelize the basic algorithm using MPI.
2. The only communication among the tasks occurs when we’re computing the forces, and, in order to compute the forces, each task/particle needs the position and mass of every other particle.
3. MPI\_Allgather is expressly designed for this situation, since it collects on each process the same information from every other process.
4. In the shared-memory implementations, we collected most of the data associated with a single particle (mass, position, and velocity) into a single struct.
5. However, if we use this data structure in the MPI implementation, we’ll need to use a derived datatype in the call to MPI\_Allgather, and communications with derived datatypes tend to be slower than communications with basic MPI types.
6. In the program we’ve written, we made the following choices with respect to the data structures:

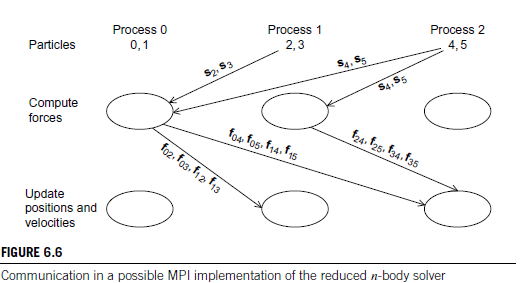
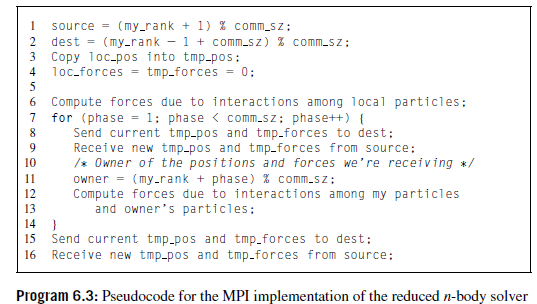
* Each process stores the entire global array of particle masses.
* Each process only uses a single n-element array for the positions.
* Each process uses a pointer loc pos that refers to the start of its block of pos. Thus, on process, 0 local pos = pos, on process 1 local pos = pos + loc n, and, so on.





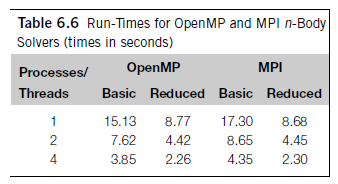
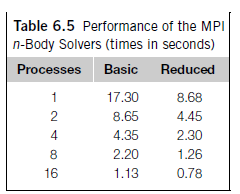
**Parallelizing the reduced solver using MPI**

1. The “obvious” implementation of the reduced algorithm is likely to be extremely complicated. Before computing the forces, each process will need to gather a subset of the positions, and after the computation of the forces, each process will need to scatter some of the individual forces it has computed and add the forces it receives.



**Performance of the MPI solvers**

Table 6.5 shows the run-times of the two n-body solvers when they’re run with 800 particles for 1000 timesteps on an Inﬁniband-connected cluster. All the timings were taken with one process per cluster node. The run-times of the serial solvers differed from the single-process MPI solvers by less than 1%, so we haven’t included them.



**5.2 TREE SEARCH**

1. Many problems can be solved using a tree search. As a simple example, consider the traveling salesperson problem, or TSP.
2. In TSP, a salesperson is given a list of cities she needs to visit and a cost for traveling between each pair of cities. Her problem is to visit each city once, returning to her hometown, and she must do this with the least possible cost.
3. A route that starts in her hometown, visits each city once and returns to her hometown is called a tour; thus, the TSP is to ﬁnd a minimum-cost tour.
4. The number of possible solutions to TSP grows exponentially as the number of cities is increased. For example, if we add one additional city to an n-city problem, we’ll increase the number of possible solutions by a factor of n−1.
5. Thus, although there are only six possible solutions to a four-city problem, there are 4×6=24 to a ﬁve-city problem, 5×24=120 to a six-city problem, 6×120=720 to a seven-city problem, and so on.

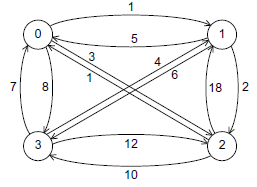


Fig. A four-city TSP

1. In Figure 6.9 we’ve represented a four-city TSP as a labeled, directed graph. A graph (not to be confused with a graph in calculus) is a collection of vertices and edges or line segments joining pairs of vertices.
2. In a directed graph or digraph, the edges are oriented—one end of each edge is the tail, and the other is the head. A graph or digraph is labeled if the vertices and/or edges have labels.
3. In our example, the vertices of the digraph correspond to the cities in an instance of the TSP, the edges correspond to routes between the cities, and the labels on the edges correspond to the costs of the routes. For example, there’s a cost of 1 to go from city 0 to city 1 and a cost of 5 to go from city 1 to city 0.

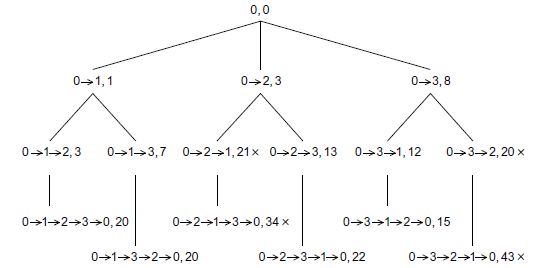


Fig. Search tree for four-city TSP

If we choose vertex 0 as the salesperson’s home city, then the initial partial tour consists only of vertex 0, and since we’ve gone nowhere, it’s cost is 0. Thus, the root of the tree in Figure 6.10 has the partial tour consisting only of the vertex 0 with cost 0. From 0 we can ﬁrst visit 1, 2, or 3, giving us three two-city partial tours with costs 1, 3, and 8, respectively.

In Figure 6.10 this gives us three children of the root. Continuing, we get six three-city partial tours, and since there are only four cities, once we’ve chosen three of the cities, we know what the complete tour is. Now, to ﬁnd a least-cost tour, we should search the tree.

There are many ways to do this, but one of the most commonly used is called depth-ﬁrst search. In depth ﬁrst search, we probe as deeply as we can into the tree.

In our example, we’ll start at the root, and branch left until we reach the leaf labeled

**0→1→2→3→0, Cost 20.**

Then we back up to the tree node labeled 0→1, since it is the deepest ancestor node with unvisited children, and we’ll branch down to get to the leaf labeled

**0→1→3→2→0, Cost 20.**

Continuing, we’ll back up to the root and branch down to the node labeled 0→2. When we visit its child, labeled

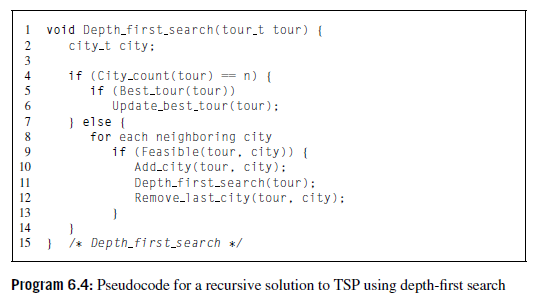
**0→2→1, Cost 21,**

we’ll go no further in this subtree, since we’ve already found a complete tour with cost less than 21. We’ll back up to 0→2 and branch down to its remaining unvisited child. Continuing in this fashion, we eventually ﬁnd the least-cost tour

**0→3→1→2→0, Cost 15.**

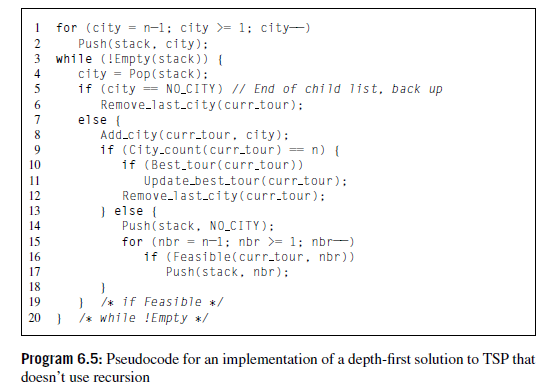
**Recursive depth-ﬁrst search**

1. Using depth-ﬁrst search we can systematically visit each node of the tree that could possibly lead to a least-cost solution. The simplest formulation of depth-ﬁrst search uses recursion (see Program 6.4).
2. Later on it will be useful to have a deﬁnite order in which the cities are visited in the for loop in Lines 8 to 13, so we’ll assume that the cities are visited in order of increasing index, from city 1 to city n−1.
3. The algorithm makes use of several global variables:.
   * *n*: the total number of cities in the problem.
   * *digraph*: a data structure representing the input digraph.
   * *hometown*: a data structure representing vertex or city 0, the salesperson’s hometown.
   * *best\_tour*: a data structure representing the best tour so far
4. The function City count examines the partial tour tour to see if there are n cities on the partial tour. If there are, we know that we simply need to return to the hometown to complete the tour, and we can check to see if the complete tour has a lower cost than the current“best tour” by calling Best\_tour.
5. If it does, we can replace the current best tour with this tour by calling the function Update best tour. Note that before the ﬁrst call to Depth\_first\_search, the best tour variable should be initialized so that its cost is greater than the cost of any possible least-cost tour.



**Nonrecursive depth-ﬁrst search**

1. 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 when we try to parallelize tree search by dividing tree nodes among the threads or processes.
2. It is possible to write a nonrecursive depth-ﬁrst search. The basic idea is modeled on recursive implementation. Recall that recursive function calls can be implemented by pushing the current state of the recursive function onto the run-time stack.
3. Thus, we can try to eliminate recursion by pushing necessary data on our own stack before branching deeper into the tree, and when we need to go back up the tree—either because we’ve reached a leaf or because we’ve found a node that can’t lead to a better solution—we can pop the stack.



**5.2.1 Parallelizing the tree-search programs using OpenMP**

1. The issues involved in implementing the static and dynamic parallel tree-search programs using OpenMP are the same as the issues involved in implementing the programs using Pthreads.
2. There are almost no substantive differences between a static implementation that uses OpenMP and one that uses Pthreads. However, a couple of points should be mentioned:

1. When a single thread executes some code in the Pthreads version, the test

if (my rank == whatever)

can be replaced by the OpenMP directive

# pragma omp single

This will insure that the following structured block of code will be executed by one thread in the team, and the other threads in the team will wait in an implicit barrier at the end of the block until the executing thread is ﬁnished. When whatever is0, the test can also be replaced by the OpenMP directive

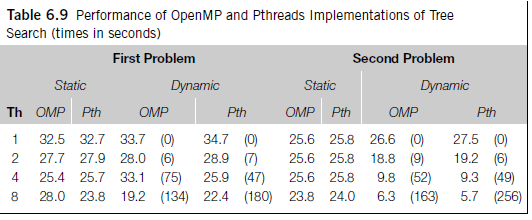
# pragma omp master

This will insure that thread 0 executes the following structured block of code. However, the master directive doesn’t put an implicit barrier at the end of the block, so it may be necessary to also add a barrier directive after a structured block that has been modiﬁed by a master directive.

2. The Pthreads mutex that protects the best tour can be replaced by a single critical directive placed either inside the Update best tour function or immediately before the call to Update best tour. This is the only potential source of a race condition after the distribution of the initial tours, so the simple critical directive won’t cause a thread to block unnecessarily.

**Performance of the OpenMP implementations**

Table 6.9 shows run-times of the two OpenMP implementations on the same two ﬁfteen-city problems



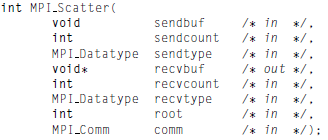
**5.2.2 Implementation of tree search using MPI and static partitioning**

1. The vast majority of the code used in the static parallelizations of tree search using Pthreads and OpenMP is taken straight from the second implementation of serial, iterative tree search. In fact, the only differences are in starting the threads, the initial partitioning of the tree, and the Update best tour function.
2. We might therefore expect that an MPI implementation would also require relatively few changes to the serial code, and this is, in fact, the case.
3. Thereistheusualproblemofdistributingtheinputdataandcollectingtheresults. In order to construct a complete tour, a process will need to choose an edge into each vertex and out of each vertex.
4. Thus, each tour will require an entry from each row and each column for each city that’s added to the tour, so it would clearly be advantageous for each process to have access to the entire adjacency matrix.
5. Oncetheprocesseshavecopiesoftheadjacencymatrix,thebulkofthetreesearch can proceed as it did in the Pthreads and OpenMP implementations. The principal differences lie in

* partitioning the tree,
* checking and updating the best tour, and
* after the search has terminated, making sure that process 0 has a copy of the best tour for output.

**Partitioning the tree**

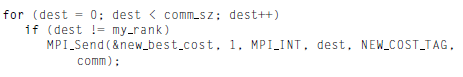
1. In the Pthreads and OpenMP implementations, thread 0 uses breadth-ﬁrst search to search the tree until there are at least thread count partial tours. Each thread then determines which of these initial partial tours it should get and pushes its tours on to its local stack.
2. Certainly MPI process 0 can also generate a list of comm sz partial tours. However, since memory isn’t shared, it will need to send the initial partial tours to the appropriate process. We could do this using a loop of sends, but distributing the initial partial tours looks an awful lot like a call to MPI\_Scatter. First recall the syntax of MPI Scatter:



1. Process root sends sendcount objects of type sendtype from sendbuf to each process in comm. Each process in comm receives recvcount objects of type recvtype into recvbuf.
2. Most of the time, sendtype and recvtype are the same and sendcount and recvcount are also the same. In any case, it’s clear that the root process must send the same number of objects to each process.

**Maintaining the best tour**

1. First note that when a process ﬁnds a new best tour, it really only needs to send its cost to the other processes. Each process only makes use of the cost of the current best tour when it calls Best tour.
2. Also, when a process updates the best tour, it doesn’t care what the actual cities on the former best tour were; it only cares that the cost of the former best tour is greater than the cost of the new best tour.
3. During the tree search, when one process wants to communicate a new best cost to the other processes, it’s important to recognize that we can’t use MPI Bcast, for recall that MPI\_Bcast is blocking and every process in the communicator must call MPI Bcast.
4. However, in parallel tree search the only process that will know that a broad cast should be executed is the process that has found a new best cost.
5. MPI provides several options. The simplest is to have the process that ﬁnds a new best cost use MPI Send to send it to all the other processes:



1. The destination processes can periodically check for the arrival of new best tour costs. We can’t use MPI Recv to check for messages since it’s blocking; if a process calls



**Performance of the MPI programs**

1. Table 6.11 shows the performance of the two MPI programs on the same two ﬁfteen-city problems on which we tested the Pthreads and the OpenMP implementations.
2. These results were obtained on a different system from the system on which we obtained the Pthreads results. We’ve also included the Pthreads results for this system, so that the two sets of results can be compared. The nodes of this system only have four cores, so the Pthreads results don’t include times for 8 or 16 threads. The cutoff number of cities for the MPI runs was 12.

