Chapter 17
Parallel Work Queues

Part I. Preliminaries
Part II. Tightly Coupled Multicore
  Chapter 6. Parallel Loops
  Chapter 7. Parallel Loop Schedules
  Chapter 8. Parallel Reduction
  Chapter 9. Reduction Variables
  Chapter 10. Load Balancing
  Chapter 11. Overlapping
  Chapter 12. Sequential Dependencies
  Chapter 13. Strong Scaling
  Chapter 14. Weak Scaling
  Chapter 15. Exhaustive Search
  Chapter 16. Heuristic Search
  Chapter 17. Parallel Work Queues

Part III. Loosely Coupled Cluster
Part IV. GPU Acceleration
Part V. Map-Reduce
Appendices
Consider a computer network with eight nodes, numbered 0 through 7. Certain nodes are connected to each other via the following bidirectional communication links: 1–4, 3–6, 1–7, 5–6, 1–2, 5–7, 0–4, 4–6, 0–7, 2–7, 0–2, 1–6, and 0–3. You want to broadcast a message to all the nodes. You start by sending the message to one of the nodes. That node forwards the message to another node to which it is connected. Each node continues to forward the message to one further node. The last node in the chain sends the message back to the original node as an acknowledgment that the broadcast is complete. Two questions: Is it even possible to send the message from one node to another such that the message reaches every node and returns to the origin? If so, to what sequence of nodes must the message be sent?

Just looking at the list of links, it might take you quite a while to answer those questions. But if I draw the computer network in the form of a graph, like so:

![Graph of computer network](image)

it becomes apparent that the message can in fact be sent to all the nodes in the order 0, 4, 1, 2, 7, 5, 6, 3, and back to 0 again.

This is an example of the Hamiltonian cycle problem, another well-known graph theory problem like the minimum vertex cover problem. A cycle in a graph is a sequence of adjacent vertices (vertices connected by an edge) that starts and ends with the same vertex. A Hamiltonian cycle is a cycle that includes each vertex in the graph exactly once. Hamiltonian cycles are named after nineteenth century Irish physicist Sir William Rowan Hamilton, who invented a puzzle called the “Icosian Game” that involved finding a path that visited all the vertices of a dodecahedron and returned to the starting point.

The Hamiltonian cycle problem is an instance of a class of problems called nondeterministic polynomial-time (NP) problems. An NP problem is a decision problem, one whose answer is “yes” or “no”. In our case, the decision problem is “Does a Hamiltonian cycle exist in a given graph?” If the answer to an NP problem is yes, the solution includes a proof that the answer is correct; furthermore, the proof can be verified by an algorithm whose asymptotic running time is a polynomial function of the problem size. In our case, the proof is simply a list of the vertices in the Hamiltonian cycle; and the proof can be verified in linear time (proportional to the problem size, that is, the number of vertices) by checking that each vertex in the purported Hamiltonian cycle is adjacent to its predecessor.

Although the solution to an NP problem can be verified in polynomial time, this says nothing about the time required to find the solution. There
might be an efficient (polynomial-time) algorithm to find the solution, or there might not.

It’s always possible to solve an NP problem by exhaustive search. In our case, this involves generating every possible path in the graph and checking to see whether any of these paths is a Hamiltonian cycle. But in general, the number of possible paths in a graph, as a function of the number of vertices and edges, grows much faster than any polynomial function. No one knows an efficient algorithm for finding a Hamiltonian cycle in an arbitrary graph.

Let’s develop a parallel program to solve the Hamiltonian cycle problem by exhaustive search. This requires generating and checking each possible path in the graph. There are two approaches for generating all paths: breadth first search and depth first search.

A breadth first search starts with one vertex of the graph. Because all the vertices have to appear in a Hamiltonian cycle, it doesn’t matter where the search starts; I’ll start with vertex 0:

I now have one subproblem: “Find all paths starting with vertex 0.” The breadth first search continues by examining all vertices adjacent to vertex 0:

Now I have four subproblems: “Find all paths starting with vertices 0–2; 0–3; 0–4; and 0–7.” For each of these subproblems, the breadth first search continues by examining all vertices adjacent to the last vertex in the partial path, that do not already appear in the partial path:

Now I have seven subproblems: “Find all paths starting with vertices 0–2–1; 0–2–7; 0–3–6; 0–4–1; 0–4–6; 0–7–1; and 0–7–2.” The breadth first search continues in this fashion, exploring the search tree one level at a time, until every possible path has been visited. At any stage, a path from the root to a leaf of the search tree corresponds to a path in the original graph. If the search reaches a path that contains all the vertices, and the last vertex is adjacent to the first vertex, it’s a Hamiltonian cycle.
A breadth first search is easy to parallelize. Each subproblem can be processed in parallel with all the other subproblems. For the typical case where there are more subproblems than there are cores, a parallel program can maintain a queue of pending subproblems. Each thread takes a subproblem (partial path) off the queue, extends the partial path by each possible adjacent vertex, and adds these new subproblems back into the queue. Whenever a thread finishes processing a subproblem, it goes to work on the next subproblem from the queue.

However, in a breadth first search, the number of subproblems—hence, the amount of storage required for the subproblem queue—becomes enormously large as the search proceeds to each successive level in the search tree. If the average vertex degree (average number of vertices adjacent to each vertex) is $K$, then each level of the search tree has about $K$ times as many nodes as the previous level, the number of nodes at level $L$ is about $K^L$, and the number of leaf nodes at level $V$ is about $K^V$, where $V$ is the number of vertices in the original graph—an exponential function of $V$. No supercomputer in the world has that much storage, even for a moderately sized graph.

The other technique, a depth first search, also starts with one vertex of the graph, such as vertex 0. The search then finds one complete path by visiting a sequence of adjacent vertices (that have not appeared in the path yet) until it can go no farther; for example:

```
0
  2
 1
  4
 6
 3
```

From vertex 3, edges lead to vertices 0 and 6; but vertices 0 and 6 already appear in the path; so the path cannot be extended to any new vertices. Because the path does not include all the vertices, it is not a solution. The depth first search therefore backtracks one or more levels in the search tree until it returns to a vertex that has an edge to an as yet unvisited vertex. In this example, vertex 6 has edges to vertices 1, 3, 4, and 5. The search has already tried vertex 3. Vertices 1 and 4 are already in the path. That leaves vertex 5 as an
alternative. The search branches off vertex 6 to vertex 5, and proceeds from there:

Another brick wall; the path cannot be extended past vertex 7, and the path does not include all the vertices, so this path is not a solution either. Now the depth first search has to backtrack all the way back to vertex 1 before proceeding along an alternate path:

The depth first search continues in this fashion, proceeding as deep as it can go along each branch of the search tree, then backtracking to a different branch, until it has visited every possible path or it has found a Hamiltonian cycle.
Unlike a breadth first search, a depth first search’s storage is not an exponential function of $V$, the number of vertices. Rather, the depth first search requires storage proportional just to $V$, to hold the current partial path—a linear function of $V$. However, a depth first search is difficult to parallelize. Whether the search is implemented recursively or iteratively, the search has to proceed in sequence down each branch of the tree and back up again.

A breadth first search is easy to parallelize but requires too much storage. A depth first search requires just a little storage but is difficult to parallelize. What’s a poor parallel program designer to do?

The answer is to use both strategies. A parallel program that must examine potentially all branches of a search tree should start by doing a breadth first search in parallel until it reaches a certain threshold level in the search tree. At this point the program will have accumulated some number of subproblems in a queue. The program should then switch strategies and do a depth first search on each subproblem. Because the subproblems can be searched independently, the program can do the depth first searches in parallel, each depth first search being done sequentially in a separate thread.

This way, we get the best of both worlds. Stopping the breadth first search at the threshold level ensures that the subproblem queue does not consume an excessive amount of storage. Having multiple subproblems in the queue ensures that all the machine’s cores can be occupied doing depth first searches simultaneously.

What should the threshold level be? That depends on the particular problem. In general, the subproblems’ depth first searches will take different amounts of time. The subproblems should therefore be partitioned among the parallel threads (cores) in a dynamic fashion. The threshold level should be set so that after the breadth first search phase, there are enough subproblems to balance the load across the cores in the machine for the depth first search.
phase. You should experiment with various threshold levels until you find one that yields the smallest running time on the parallel machine.

What I've described is a general pattern that can be applied to any problem involving a search tree, not just the Hamiltonian cycle problem. Many NP problems are like this; the best known algorithm for finding the exact solution requires searching an exponentially large search tree. Doing the search in parallel can speed it up. (Keep in mind that we are talking about an exhaustive search program that is guaranteed to find the exact solution, not a heuristic search program that typically finds only an approximate solution.)

Let's consider how to code a parallel program that does breadth first and depth first searches over a search tree.

In all the parallel programs we've studied so far, the number of loop iterations was known before the loop started. I therefore could iterate using a parallel for loop, and I could specify the known lower and upper bound loop indexes in the \texttt{parallelFor()} statement.

When iterating over the subproblems in a search tree, however, \textit{the number of iterations is not known before the loop starts} (in general). Rather, the loop must iterate \textit{while} there are still subproblems to look at. Furthermore, the act of processing one subproblem might create one or more additional subproblems that have to be processed. To do this kind of iteration, I'll use a new programming pattern: the \textit{parallel work queue}.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{work_queue_diagram}
\caption{Parallel Work Queue Diagram}
\end{figure}

In this pattern, there is a \textit{work queue} containing \textit{work items}. Typically, one or more work items are added to the queue before the parallel processing starts. A parallel thread team then commences operating. Each team thread repeatedly takes one work item out of the queue and does whatever computation is needed for that item. As part of the computation, the thread might create one or more new work items, which the thread adds back into the queue. Work items can also be added to the queue from outside the parallel team. Processing the work items one at a time automatically balances the load, like a dynamic schedule. The threads continue in this fashion until the queue is empty and all the threads have finished computing their final work items.

I’ll use the parallel work queue pattern for the parallel Hamiltonian cycle program. Each work item will be a subproblem in the search, that is, a partial
path. The first work item will be the initial subproblem, namely a partial path consisting of just vertex 0. A parallel team thread will process a work item by doing a breadth first search if the partial path is below the threshold level, or by doing a depth first search if the partial path is at the threshold level.

The Parallel Java 2 Library supports parallel loops over the work items in a work queue. The pattern for writing this kind of parallel loop is

```java
WorkQueue<W> queue = new WorkQueue<W>();
queue.add (new W (...));  // First work item
parallelFor (queue) .exec (new ObjectLoop<W>(){
    public void run (W workitem)
    {
        Loop body code for workitem
        queue().add (new W (...));  // (optional)
    }
});
```

The work queue itself is an instance of the generic class edu.rit.pj2.WorkQueue. Replace the generic type parameter <W> with the data type of the work items; this can be any object type. Add one or more work items to the queue before commencing the parallel loop. (If you don’t, the parallel loop won’t do anything.) The argument of the parallelFor() method is the work queue, rather than the index bounds. The argument of the exec() method is the loop object itself, an instance of the generic class edu.rit.pj2.ObjectLoop. The loop object’s run() method’s argument is a work item; the method body contains the loop body code for processing that work item. The run() method may optionally add more work items to the work queue; to access the work queue from inside the loop object, call the queue() method. Class ObjectLoop has the same capabilities as all the other parallel loops, including per-thread variables (fields), the start() method, parallel reduction with the threadLocal() method, the finish() method, and so on.

To begin designing the multicore parallel Hamiltonian cycle program, I need a class for the work items in the parallel work queue, namely class edu.rit.pj2.example.HamCycState (Listing 17.1). This class extends class Tuple because I will be reusing the class for a cluster parallel Hamiltonian cycle program in a later chapter. For now, we can ignore the tuple.

The multicore parallel Hamiltonian cycle program will have many instances of class HamCycState. However, all instances will be searching the same graph, and all instances will use the same threshold level for switching from breadth first search to depth first search. Therefore, class HamCycState has static fields for the graph (line 11), the number of vertices (line 12), and the threshold level (line 15). The static fields are shared by all instances.

There is also a static stop flag for terminating the search in all the parallel team threads as soon as a solution is found (line 18). By the way, the stop flag is declared with the keyword volatile. There’s a reason for that. I’ll
package edu.rit.pj2.example;
import edu.rit.io.InStream;
import edu.rit.io.OutStream;
import edu.rit.pj2.Tuple;
import java.io.IOException;
import java.util.Formatter;
public class HamCycState
extends Tuple
{
    // The graph being searched.
    static AMGraph graph;
    static int V;
    
    // Search level threshold at which to switch from BFS to DFS.
    static int threshold;
    
    // Flag to stop search.
    static volatile boolean stop;
    
    // Vertices in the path.
    private int[] path;
    
    // Search level = index of last vertex in the path.
    private int level;
    
    // Specify the graph to be analyzed using DFS only.
    public static void setGraph
    (AMGraph graph)
    {
        HamCycState.graph = graph;
        HamCycState.V = graph.V();
        HamCycState.threshold = 0;
        HamCycState.stop = false;
    }
    
    // Construct a new search state object that supports DFS only.
    public HamCycState()
    {
        path = new int [V];
        for (int i = 0; i < V; ++ i)
            path[i] = i;
        level = 0;
    }
    
    // Clone this search state object.
    public Object clone()
    {
        HamCycState graph = (HamCycState) super.clone();
        graph.path = (int[]) this.path.clone();
        return graph;
    }
    
    // Search the graph from this state.
    public HamCycState search()
    {
        return (level < threshold) ? bfs() : dfs();
    }
}

Listing 17.1. HamCycState.java (part 1)
discuss the reason in the “Under the Hood” section below. For now, just ignore the volatile keyword.

The static setGraph() method (line 27) initializes the shared static fields to start a new search for a Hamiltonian cycle in a particular graph. Instances of class HamCycState carry out the actual search.

Each instance of class HamCycState will search starting from a different partial path. The search state is represented by the two instance fields path and level (lines 21–24). These fields are not shared; each state object gets its own separate path and level fields. At all times, the elements path[0] through path[level] contain the vertices in the partial path at which the search is positioned. The remaining elements of path contain the vertices that are not in the partial path, in no particular order. For example, when searching the eight-vertex graph from the beginning of the chapter, the state might be path = {0, 3, 6, 1, 4, 5, 2, 7} and level = 2; this represents the partial path 0–3–6.

The HamCycState constructor (line 37) initializes path to contain all the vertices 0 through V–1 and initializes level to 0; this represents the partial path consisting just of vertex 0. There is also a clone() method (line 46) that makes a deep copy of a state object.

The search() method (line 54) either does one step of a breadth first search or does a complete depth first search, starting from the state object’s current search state, depending on whether the search level is below or above the threshold level. The search is carried out by a separate method. The search() method either returns a state object containing a Hamiltonian cycle or returns null if no Hamiltonian cycle was found.

The breadth first search, in the bfs() method (line 78), iterates over the vertices not already in the partial path, namely, those stored in path[level +1] through path[V-1] (line 80). If one of these vertices is adjacent to the last vertex in the partial path (line 81), the adjacent vertex is appended to the partial path (lines 83–84), forming a new subproblem. A clone of this new subproblem is added to the work queue (line 85), to be processed later. The partial path is put back the way it was (line 86), and the loop goes on to try the next available vertex. The bfs() method returns null, signifying that it did not find a Hamiltonian cycle. (The program assumes that the threshold level is smaller than the number of vertices, so the breadth first search will never go far enough to find a Hamiltonian cycle.)

The depth first search, in the dfs() method (line 92), is implemented recursively. As in any recursive algorithm, there are two cases, the base case and the recursive case. The base case (lines 95–99) happens when all the vertices are in the path. If the last vertex, path[V-1], is adjacent to the first vertex, path[0], then the path is a Hamiltonian cycle, and a reference to this state object (which contains the Hamiltonian cycle) is returned. The recursive case (lines 104–112) happens when some of the vertices are not in the path
// Stop the search in progress.
public static void stop()
{
    stop = true;
}

// Returns a string version of this search state object.
public String toString()
{
    StringBuilder b = new StringBuilder();
    Formatter f = new Formatter (b);
    for (int i = 0; i <= level; ++ i)
    {
        if (i > 0) f.format (" ");
        f.format ("%d", path[i]);
    }
    return b.toString();
}

// Do a breadth first search of the graph from this state.
private HamCycState bfs()
{
    for (int i = level + 1; i < V && ! stop; ++ i)
    {
        if (adjacent (i))
        {
            ++ level;
            swap (level, i);
            enqueue ((HamCycState) this.clone());
            -- level;
        }
    }
    return null;
}

// Do a depth first search of the graph from this state.
private HamCycState dfs()
{
    // Base case.
    if (level == V - 1)
    {
        if (adjacent (0))
            return this;
    }

    // Recursive case.
    else
    {
        for (int i = level + 1; i < V && ! stop; ++ i)
        {
            if (adjacent (i))
            {
                ++ level;
                swap (level, i);
                if (dfs() != null)
                    return this;
                -- level;
            }
        }
    }
    return null;
}

Listing 17.1. HamCycState.java (part 2)
yet. The code is nearly the same as for the breadth first search, except when a new vertex is appended to the path, \texttt{dfs()} is called recursively to handle the new subproblem. If the recursive call finds a solution, again, a reference to this state object is returned; otherwise the depth first search continues. If neither the base case nor the recursive case finds a solution, the \texttt{dfs()} method returns null.

In both \texttt{bfs()} and \texttt{dfs()}, the loop over the adjacent vertices exits early if a solution is found, namely if the \texttt{stop} flag is set to true (lines 80 and 104). Upon finding a solution, the main program sets the \texttt{stop} flag true by calling the static \texttt{stop()} method (line 59). This ensures that if one parallel team thread finds a solution, the program stops immediately without needing to wait for the other threads to finish their searches, which could take quite a while.

The rest of class \texttt{HamCycState} consists of a few subroutines used by the \texttt{bfs()} and \texttt{dfs()} methods, as well as a \texttt{toString()} method for displaying the search state (path) as a string. There is also an \texttt{enqueue()} method, which the \texttt{bfs()} method calls to add a state object (work item) to the parallel work queue. The \texttt{HamCycState} base class does not support enqueuing work items; thus, the base class supports depth first searches only.

The subclass \texttt{HamCycStateSmp} (Listing 17.2) encapsulates a search state object for the multicore parallel program. The static queue field, shared by all instances, holds a reference to the parallel work queue (line 7). The static \texttt{setGraph()} method (line 10) initializes the shared static fields to start a new search for a Hamiltonian cycle in a particular graph, and it specifies the threshold level and the work queue to use during the search. The \texttt{enqueue()} method (line 29) is overridden to add a state object to the work queue. With the search state class designed, I can write the code for the multicore parallel Hamiltonian cycle finding program itself, class \texttt{edu.rit.pj2.example.HamCycSmp} (Listing 17.3). The program’s command line arguments are a constructor expression for the graph to be solved and the threshold level. The program declares a global variable (line 11) that will hold a state object containing the Hamiltonian cycle that was found, or null if no Hamiltonian cycle was found.

The main program (line 14) is where all the parallel processing takes place. The program sets up the parallel work queue (line 24). The program constructs an adjacency matrix for the graph to be solved (line 28) and uses that to initialize the search (lines 27–29). The program adds the first work item to the queue, namely a new state object with a partial path of just vertex 0 (line 32). The program then does a parallel loop over the work items in the work queue (line 35). For each work item (search state object), the program calls the \texttt{search()} method to do a breadth first or depth first search from that search state (line 39). If a Hamiltonian cycle is found, the program stops the search (line 42) and records the Hamiltonian cycle as the solution (line 43).
// Determine if the given path element is adjacent to the current
// path element.
private boolean adjacent
(int a)
{
    return graph.isAdjacent (path[level], path[a]);
}

// Swap the given path elements.
private void swap
(int a,
 int b)
{
    int t = path[a];
    path[a] = path[b];
    path[b] = t;
}

// Enqueue the given search state object during a BFS.
protected void enqueue
(HamCycState state)
{
    throw new UnsupportedOperationException();
}

Listing 17.1. HamCycState.java (part 3)

package edu.rit.pj2.example;
import edu.rit.pj2.WorkQueue;
public class HamCycStateSmp
    extends HamCycState
{
    // Multicore parallel work queue.
    private static WorkQueue<HamCycState> queue;

    // Specify the graph to be analyzed using BFS and DFS.
    public static void setGraph
        (AMGraph graph,
         int threshold,
         WorkQueue<HamCycState> queue)
    {
        if (queue == null)
            throw new NullPointerException
                ("HamCycStateSmp.setGraph(): queue is null");
        HamCycState.setGraph (graph);
        HamCycState.threshold = threshold;
        HamCycStateSmp.queue = queue;
    }

    // Construct a new search state object that supports BFS and DFS.
    public HamCycStateSmp()
    {
        super();
    }

Listing 17.2. HamCycStateSmp.java (part 1)
From this point on, whenever a parallel team thread calls the search() method, the method does nothing and returns immediately; thus, the team threads drain any remaining work items out of the queue without doing further searching. When the parallel loop finishes, the program either prints the solution (lines 48–49) or prints a message that there was no solution (lines 50–51).

To study the program’s scalability, I want to find some input graphs for which the program takes a long time to find a Hamiltonian cycle. Finding such graphs turns out to be a bit difficult. I considered random graphs with \( V = 30 \) vertices and \( E \) = from 50 to 90 edges. For each \( E \), I generated 100 different random graphs; ran each graph through the Hamiltonian cycle program; measured the running times; and counted how many of the 100 graphs had a Hamiltonian cycle. Figure 17.1 plots the median of the program’s running times for the 100 graphs for each \( E \), as well as the probability that a 30-vertex \( E \)-edge random graph has a Hamiltonian cycle for each \( E \).

As \( E \) increases, the running time increases to a peak and then decreases abruptly. Also, the probability of a Hamiltonian cycle stays low until about \( E = 60 \), then increases as \( E \) increases. Why? When \( E \) is small, there aren’t many paths starting from vertex 0, so the program doesn’t take very long to go through them all and either find a solution or conclude none are Hamiltonian cycles. When \( E \) is large, there are many, many paths starting from vertex 0, and most of them are Hamiltonian cycles, so the program doesn’t take very long to find a solution. When \( E \) is somewhere in the middle, however, there are still many, many paths starting from vertex 0, but few if any of them are Hamiltonian cycles, so the program has to look at most or all of them before

![Figure 17.1. Program results for 30-vertex random graphs](image-url)
Chapter 17. Parallel Work Queues

package edu.rit.pj2.example;
import edu.rit.pj2.ObjectLoop;
import edu.rit.pj2.Task;
import edu.rit.pj2.WorkQueue;
import edu.rit.util.GraphSpec;
import edu.rit.util.Instance;
public class HamCycSmp
extends Task
{
    // Hamiltonian cycle that was found.
    HamCycState hamCycle;

    // Task main program.
    public void main (String[] args)
    throws Exception
    {
        // Parse command line arguments.
        if (args.length != 2) usage();
        String ctor = args[0];
        int threshold = Integer.parseInt (args[1]);

        // Set up parallel work queue.
        WorkQueue<HamCycState> queue = new WorkQueue<HamCycState>();

        // Construct graph spec, set up graph.
        HamCycStateSmp.setGraph
            (new AMGraph ((GraphSpec) Instance.newInstance (ctor)),
             threshold, queue);

        // Add first work item to work queue.
        queue.add (new HamCycStateSmp());

        // Search the graph in parallel.
        parallelFor (queue) .exec (new ObjectLoop<HamCycState>()
        {
            public void run (HamCycState state)
            {
                HamCycState cycle = state.search();
                if (cycle != null)
                {
                    HamCycState.stop();
                    hamCycle = cycle;
                }
            }
        });

Listing 17.3. HamCycSmp.java (part 1)
finding a solution or deciding there is no solution. This is what’s taking a long time.

I picked a particular random graph with 30 vertices and 72 edges. I ran the HamCycSmp program on this graph, with the threshold level set to 2, 3, and 4. For each threshold level, I ran the program on one through 12 cores of a tardis node. Figure 17.2 plots the running times and speedups I measured. The sequential program takes somewhat over 100 seconds to find a Hamiltonian cycle. On one core, the parallel program’s running time is similar. But on two through 12 cores, the parallel program’s running times are all over the place. There is not a steady decrease in the running times, as we have seen with other programs under strong scaling. With a threshold of 2, the speedups are mostly less than ideal. With a threshold of 3, the speedups are greater than ideal. With a threshold of 4, the speedups are considerably greater than ideal, reaching a speedup of 50 on 12 cores. What’s going on?

On one core, the program processes the subproblems one by one, from the beginning of the search tree to the end. But on more than one core, the program looks at several subproblems in the middle of the search tree at the same time in multiple threads. Depending on the threshold level, which determines which subproblems are added to the work queue, and depending on the number of threads in the thread team, it might happen that one of the threads goes to work on a subproblem where a Hamiltonian cycle is found right away, so the running time is small. On one core, the one thread has to do complete searches on all the subproblems preceding the subproblem that contains a solution, and this takes a long time.

In general, a parallel program that traverses a search tree will exhibit speedups that vary depending on the particular problem being solved as well.
as the number of cores. The speedups might be roughly the same as the number of cores, or they might be larger or smaller. I just happened to luck out when I picked this graph.

Under the Hood

As previously mentioned, when one of the parallel team threads in the HamCycSmp program finds a Hamiltonian cycle, the searches in the other team threads must stop immediately. To signal this, the global stop flag in class HamCycState (Listing 17.1 line 25) is set to true. This variable is shared by the team threads. All the team threads read the stop flag (lines 80 and 104) and exit the search loop if the flag is true. One of the team threads writes the stop flag (line 61) when it finds a Hamiltonian cycle.

Two issues with the stop flag now arise. The first issue has to do with thread synchronization. Normally, when some threads read a shared variable and other threads write a shared variable, the threads have to synchronize with each other. This ensures that one thread’s operation on the variable does not interfere with another thread’s operation on the variable. But there is no synchronization code around the stop flag; the threads just read or write the

Listing 17.3. HamCycSmp.java (part 2)
variable directly. This works because the JVM ensures that all reads and writes of a Boolean variable are atomic. That is, while a read or a write is in progress on a Boolean variable, no other read or write will be performed on that variable until after the first read or write finishes. This atomic behavior implicitly synchronizes the multiple threads doing reads and writes of the stop flag.

In fact, read and write operations on the primitive types boolean, byte, char, short, int, and float are guaranteed to be atomic (but not reads or writes of types long or double). Also, read and write operations on references to objects and arrays are guaranteed to be atomic (but operations on the objects’ fields or the arrays’ elements might or might not be atomic, depending on the types of the fields or elements). On the other hand, operations that involve both reading and writing a variable, such as incrementing an integer (x++), are not atomic and must be explicitly synchronized.

The second issue has to do with how the JVM deals with a class or object shared by multiple threads, like the HamCycState class. A class’s static fields are located in a block of storage that is allocated when the class is first used. An object’s instance fields are located in a block of storage that is allocated when the object is constructed. This block of storage resides in the computer’s main memory. Suppose multiple threads use a class with static fields, or suppose multiple threads have references to a shared object with instance fields. When a thread reads a static field from the class or an instance field from the shared object, the thread typically gets the field’s value from main memory and puts the value in a register inside the CPU. Thereafter, the JVM is permitted to get the value from the register rather than from the main memory. Furthermore, the JVM is permitted to store a new value in the register without writing the new value back to the main memory right away. The JVM will eventually put the register and the main memory back in sync, but there is no guarantee on how quickly this will happen. Thus, if one thread updates a static field in a class or an instance field in a shared object, other threads might not see the new value immediately.

Consequently, in the HamCycSmp program, when one parallel team thread sets the stop variable (static field) to true, the other team threads might not see this new value right away, and the team threads might not exit their loops and terminate right away. This would cause the program to run longer than it should.

Here’s where the volatile keyword comes in. When the program reads a volatile field, the JVM is required always to obtain the field’s value from main memory, and not use a previous value that might be in a CPU register. When the program writes a volatile field, the JVM is required always to store the field’s new value into main memory immediately, and not just update a CPU register.
Thus, declaring the stop field to be volatile ensures that all reads and writes of the stop flag go to or from main memory, where all the parallel team threads can see the flag’s value. This in turn ensures that the team threads stop the search as soon as the flag is set to true; that is, as soon as a Hamiltonian cycle is found.

**Points to Remember**

- A *nondeterministic polynomial-time (NP) problem* is a decision problem for which a proof of a solution can be verified efficiently (with a polynomial-time algorithm).
- An efficient (polynomial-time) algorithm might or might not be known for finding the solution to an NP problem.
- An NP problem can always be solved by exhaustive search of all possible solutions; but the number of possible solutions typically grows much faster than any polynomial function as the problem size increases.
- Some NP problems can be solved by traversing a *search tree* consisting of all possible solutions.
- A search tree can be traversed by a *breadth first search* or a *depth first search*.
- A parallel program traversing a search tree should do a parallel breadth first search down to a certain threshold level, then should do multiple depth first searches in parallel thereafter.
- Use the *parallel work queue* pattern to do a parallel loop when the number of iterations is not known before the loop starts.
- The JVM guarantees that reads and writes of the primitive types boolean, byte, char, short, int, and float, as well as reads and writes of object and array references, are atomic.
- When a class has a static field of one of the above primitive types that is read and written by multiple threads, declare the field to be volatile.
- When a shared object has an instance field of one of the above primitive types that is read and written by multiple threads, declare the field to be volatile.