Chapter 27
Cluster Work Queues

- Part I. Preliminaries
- Part II. Tightly Coupled Multicore
- Part III. Loosely Coupled Cluster
  - Chapter 18. Massively Parallel
  - Chapter 19. Hybrid Parallel
  - Chapter 20. Tuple Space
  - Chapter 21. Cluster Parallel Loops
  - Chapter 22. Cluster Parallel Reduction
  - Chapter 23. Cluster Load Balancing
  - Chapter 24. File Output on a Cluster
  - Chapter 25. Interacting Tasks
  - Chapter 26. Cluster Heuristic Search
  - Chapter 27. Cluster Work Queues
  - Chapter 28. On-Demand Tasks
- Part IV. GPU Acceleration
- Part V. Map-Reduce
- Appendices
Recall the Hamiltonian cycle problem from Chapter 17. The problem is to decide whether a given graph has a Hamiltonian cycle. A Hamiltonian cycle is a path that visits every vertex in the graph exactly once and returns to the starting point, like the cycle 0, 4, 1, 2, 7, 5, 6, 3 in this graph:

In Chapter 17 I developed a multicore parallel program to solve the Hamiltonian cycle problem by doing an exhaustive search of all paths in the graph. The program was designed to do a parallel breadth first search down to a certain threshold level in the search tree of all paths, resulting in some number of subproblems (partial paths). The program then did multiple depth first searches in parallel, each depth first search working on one subproblem in a single thread.

I implemented this design using the parallel work queue pattern. There was a work queue of work items. Each parallel team thread repeatedly took a work item out of the queue and processed the item, possibly adding further items to the queue, until there were no more work items.
Now I want to make this a cluster parallel program, so I can have even more cores searching for a Hamiltonian cycle in parallel. I need a version of the parallel work queue pattern that runs on a cluster of multiple nodes. The threads processing the work items will be in tasks running on the nodes. However, in a distributed memory setting, the tasks cannot have a common work queue in shared memory. Instead, the program must send work items from task to task using intertask communication, that is, via tuple space.

In a cluster parallel program (a job), the parallel work queue pattern consists of three parts:

- Tuple space itself acts as the work queue. Each work item is packaged into a tuple and is put into tuple space.
- The job consists of a number of worker tasks, each task containing just one worker thread. Each thread takes a work item tuple out of tuple space, processes the work item, puts new work item tuples into tuple space if necessary, and repeats until all work items have been processed.
- The job main program puts the first work item tuple into tuple space to kick off the processing.

In the Hamiltonian cycle cluster parallel program, the work items (tuples) are partial paths, and each task performs either a breadth first or a depth first search on the partial path, depending on the search level. The breadth first search could put additional work items into tuple space, to be processed by other tasks. The threshold level limits the number of work items; once the threshold level is reached, a task does a depth first search rather than spawning more work items.

The Parallel Java 2 Library includes class edu.rit.pj2.JobWorkQueue for accessing the job’s work queue. There can only be one work queue in a job. Under the hood, Class JobWorkQueue handles all the putting and taking of work item tuples. To use class JobWorkQueue:
In the job main program, call the `getJobWorkQueue()` method to get a reference to the job’s work queue. The arguments are the class of the work items and the number of worker tasks. The work item class must be a tuple subclass.

In the worker task, call the `getJobWorkQueue()` method to get a reference to the job’s work queue. The argument is the class of the work items.

In the job main program or the worker task, call the work queue’s `add()` method, passing in the work item to add to the job’s work queue.

In the worker task, call the work queue’s `remove()` method to remove and return the next work item from the job’s work queue. Call `remove()` repeatedly until it returns null, indicating there are no more work items. As a reminder, the worker task must be single threaded.

To represent the work items, the cluster parallel HamCycClu program reuses class `HamCycState` from Chapter 17 (Listing 17.1), which the sequential HamCycSeq program and the multicore parallel HamCycSmp program also used. Class `HamCycState` encapsulates a partial path in the search over all paths in the graph. To allow work items to be sent back and forth via tuple space, class `HamCycState` is a tuple subclass. The HamCycClu program also uses class `HamCycStateClu`. This class is similar to class `HamCycStateSmp` from Chapter 17 (Listing 17.2), except class `HamCycStateClu`’s `enqueue()` method adds a work item to the job’s work queue.

Turning to the code for the HamCycClu program itself (Listing 27.1), the job main program obtains from the command line the constructor expression for the graph to be analyzed as well as the parallel search threshold level (lines 19–20). The program gets a reference to the job’s work queue (lines 23–24). The number of worker tasks is specified by the `workers=` option on the `pj2` command line, which the `workers()` method returns. Using the graph’s constructor expression, the program creates an adjacency matrix for the graph (line 28) and uses that to initialize the work item class `HamCycStateClu` (lines 27–29).

The program adds the first work item to the work queue (line 32). This is an instance of class `HamCycStateClu` containing a partial path with just vertex 0.

The job sets up two rules. The first is start rule (lines 35–36) that defines a task group with instances of the search task, defined further on. The number of search tasks is specified by the `workers=` option on the `pj2` command line. The search task group will sit in the Tracker’s queue until the cluster has enough idle cores to run all the search tasks; this is necessary because the tasks will all be communicating with each other through tuple space. The graph’s constructor expression and the parallel search threshold level are passed to each search task (line 36). The job’s second rule is a finish rule
package edu.rit.pj2.example;
import edu.rit.pj2.Job;
import edu.rit.pj2.JobWorkQueue;
import edu.rit.pj2.Task;
import edu.rit.pj2.TupleListener;
import edu.rit.pj2.tuple.ObjectTuple;
import edu.rit.util.GraphSpec;
import edu.rit.util.Instance;
public class HamCycClu
        extends Job
        {
            // Job 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 job's distributed work queue.
                    JobWorkQueue<HamCycState> queue =
                            getJobWorkQueue (HamCycState.class, workers());
                    // Construct graph spec, set up graph.
                    HamCycStateClu.setGraph
                            (new AMGraph ((GraphSpec) Instance.newInstance (ctor)),
                                threshold, queue);
                    // Add first work item to work queue.
                    queue.add (new HamCycStateClu());
                    // Set up team of worker tasks.
                    rule() .task (workers(), SearchTask.class)
                            .args (ctor, ""+threshold);
                    // Set up task to print results.
                    rule() .atFinish().task (ResultTask.class)
                            .runInJobProcess();
                }
            // Search task.
            private static class SearchTask
                    extends Task
                {
                    // Search task main program.
                    public void main
                            (String[] args)
                        throws Exception
                            {
                        // Parse command line arguments.
                        String ctor = args[0];
                        int threshold = Integer.parseInt (args[1]);
                        // Set up job's distributed work queue.
                        JobWorkQueue<HamCycState> queue =
                                getJobWorkQueue (HamCycState.class, workers());
                        }
                    }
            }
Listing 27.1. HamCycClu.java (part 1)
Once the job commences, each search task obtains the graph’s constructor expression and the parallel search threshold level (lines 53–54). The task gets a reference to the job’s work queue (lines 57–58). Using the graph’s constructor expression, the task creates an adjacency matrix for the graph (line 61) and uses that to initialize the work item class HamCycStateClu (lines 60–62). Thus, all the search tasks will analyze the same graph.

When some search task finds a Hamiltonian cycle, the task will put the solution into tuple space as an ObjectTuple containing a HamCycState object with the solution. This signals that the search should stop. The task detects this situation by setting up a tuple listener for the solution tuple (lines 65–73). When this tuple appears, the tuple listener tells the HamCycState class to stop the search (line 71).

Following the cluster parallel work queue pattern, the search task now begins the search for a Hamiltonian cycle. The task removes a work item (partial path) from the job’s work queue (line 77). The task calls the partial path’s search() method to carry out the actual search (line 79). As described in Chapter 17, if the partial path is below the search threshold level, the search() method does one further level of a breadth first search, possibly adding more partial paths to the work queue. If the partial path is at or above the search threshold level, the search() method does a complete depth first search starting from the partial path. If a Hamiltonian cycle was not found, the search() method returns null, otherwise it returns a HamCycState object containing the Hamiltonian cycle. In the latter case, the task puts the solution into tuple space (line 81), thus triggering all the tasks to stop the search. The task repeats the preceding steps until all work items in the job’s work queue have been processed (line 77), then the task terminates.

The job work queue capability requires that each worker task consists of a single thread. Accordingly, the search task has a sequential while loop (line 77) rather than a parallel loop, and the search task’s coresRequired() method is overridden to specify that the task requires one core (lines 86–89).

The final piece of the program is the result task (lines 93–109), which runs when all the search tasks have finished. If at this point an ObjectTuple containing a solution exists in tuple space, the result task prints the solution. If not, the result task prints “None.” That’s all the result task does.

I ran the sequential HamCycSeq program and the cluster parallel HamCycClu program on the tardis cluster to find Hamiltonian cycles in several different 30-vertex graphs, which had from 72 to 82 edges. The parallel program used 120 search tasks, and the search threshold level was 4. The table below shows the running times and speedups I measured.
// Construct graph spec, set up graph.
HamCycStateClu.setGraph
(new AMGraph ((GraphSpec) Instance.newInstance (ctor)),
    threshold, queue);

// Stop the search when any task finds a Hamiltonian cycle.
addTupleListener
    (new TupleListener<ObjectTuple<HamCycState>>
        (new ObjectTuple<HamCycState>()))
    {
        public void run (ObjectTuple<HamCycState> tuple)
        {
            HamCycState.stop();
        }
    });

// Search for a Hamiltonian cycle.
HamCycState state;
while ((state = queue.remove()) != null)
{
    HamCycState cycle = state.search();
    if (cycle != null)
        putTuple (new ObjectTuple<HamCycState> (cycle));
}

// The search task requires one core.
protected static int coresRequired()
{
    return 1;
}

// Result task.
private static class ResultTask
        extends Task
{
    // Task main program.
    public void main
        (String[] args)
        throws Exception
        {
            ObjectTuple<HamCycState> template =
                new ObjectTuple<HamCycState>();
            ObjectTuple<HamCycState> cycle = tryToReadTuple (template);
            if (cycle != null)
                System.out.println (cycle.item);
            else
                System.out.println ("None");
        }
}

Listing 27.1. HamCycClu.java (part 2)
**BIG CPU, BIG DATA**

The sequential program running on a single core took anywhere from 0.7 to 6.6 minutes to find a Hamiltonian cycle, depending on the graph. In all cases, running on the 120 cores of the 10 nodes of the cluster, the parallel program took less than two seconds.

Like the multicore parallel HamCycSmp program in Chapter 17, the cluster parallel HamCycClu program experienced widely varying speedups, sometimes less than the number of cores, sometimes much more—and for the same reason. On these graphs, with multiple cores in the cluster nodes processing multiple subproblems in parallel, it so happened that at least one of the cores went to work on a subproblem where a Hamiltonian cycle was found right away. Consequently, the program stopped early, and the running time was small.

### Under the Hood

As mentioned previously, class JobWorkQueue manages the job’s work queue in tuple space, handling all putting and getting of tuples to add and remove work items to and from the work queue.

Under the hood, class JobWorkQueue uses two tuple classes. The first tuple, class WorkItemTuple, encapsulates one work item in the queue. The second tuple, class ControlTuple, encapsulates the state of the queue itself. The queue state consists of three items: the total number of worker tasks using the queue; the number of worker tasks waiting to remove a work item from the queue; and the number of work items available in the queue.

When the job main program calls `getJobWorkQueue()`, the queue state is initialized by putting a control tuple containing the number of worker tasks, zero tasks waiting to remove a work item, and zero work items available.

When the job or a task calls the queue’s `add()` method to add a work item to the queue, the `add()` method takes the control tuple; puts a work item tuple containing the work item; increments the number of available work items in the control tuple; and puts the control tuple back.

When a task calls the queue’s `remove()` method to remove and return a work item from the queue, the `remove()` method takes the control tuple; increments the number of waiting tasks; and puts the control tuple back. The

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
<th>Sequential</th>
<th>HamCycClu</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>72</td>
<td>170697</td>
<td>1222</td>
<td>140</td>
</tr>
<tr>
<td>30</td>
<td>74</td>
<td>111421</td>
<td>1492</td>
<td>75</td>
</tr>
<tr>
<td>30</td>
<td>76</td>
<td>44559</td>
<td>1510</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>78</td>
<td>395392</td>
<td>1081</td>
<td>366</td>
</tr>
<tr>
<td>30</td>
<td>80</td>
<td>151292</td>
<td>1036</td>
<td>146</td>
</tr>
<tr>
<td>30</td>
<td>82</td>
<td>234992</td>
<td>1109</td>
<td>212</td>
</tr>
</tbody>
</table>
remove() method then takes the control tuple again, but this time it specifies a template that will cause the take to block until either there are one or more work items available or until all the workers tasks are waiting. In the former case the remove() method takes one of the available work item tuples; decrements the number of waiting tasks and decrements the number of available work items in the control tuple; puts the control tuple back; and returns the work item. In the latter case the remove() method just puts the control tuple back unchanged and returns null.

Why this somewhat complicated logic for removing a work item? Why not simply try to take a work item tuple and return null if there are no work item tuples in tuple space? The reason is that if there are no work item tuples, it’s not necessarily the case that processing is finished. There might still be one or more tasks processing work items, and these tasks might eventually put more work items into tuple space. We can’t conclude that processing is finished until there are no work items and all the worker tasks are waiting to remove a work item, meaning that no worker task is still processing a work item.

Because multiple threads in multiple worker tasks can update the work queue concurrently, thread synchronization is needed to ensure the work queue state is always consistent. Synchronization is achieved by taking the control tuple out of tuple space before manipulating the work queue. The semantics of the tuple space take operation ensure that if multiple threads try to

---

Listing 27.1. HamCycClu.java (part 3)
take the control tuple, only one thread will succeed; the other threads will block inside the take method. The thread that took the control tuple can then operate on the work queue. When that thread finishes its queue operation, the thread puts the control tuple reflecting the new queue state back into tuple space. Another thread can then take the control tuple and do a queue operation. In this way, only one thread at a time operates on the work queue.

**Points to Remember**

- Reiterating Chapter 17, 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.
- In a cluster parallel program, tuple space itself acts as the work queue, and the work items are encapsulated in tuples put into tuple space.
- To access the job’s work queue, use class JobWorkQueue.
- Call the `getJobWorkQueue()` method in the job or the worker task to obtain a reference to the job’s work queue.
- Call the job work queue’s `add()` method to add a work item to the queue.
- Call the job work queue’s `remove()` method to remove and return a work item from the queue.
- In the worker task, repeatedly remove a work item from the queue, process the work item, adding more work items to the queue if necessary, until the queue is empty.
- The worker tasks must be single threaded.