Chapter 26
Cluster Heuristic Search

► 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. Big Data
► Appendices
A massively parallel randomized approximation (MPRA) algorithm for solving a hard combinatorial problem—like the minimum vertex cover problem from Chapters 15 and 16—can potentially find an approximate solution closer to the exact solution by doing more iterations, thus examining more candidate solutions. And because the candidates can all be examined independently, with no sequential dependencies, an MPRA program is ideally suited for a parallel computer. Indeed, in Chapter 16 we developed a single-node multicore MPRA program for the minimum vertex cover problem, and we scaled it up to the full number of cores on one node by increasing the number of iterations in proportion to the number of cores (weak scaling).

On a cluster parallel computer, we can scale an MPRA program up even further, utilizing all the cores on multiple nodes instead of just one node.

Recall the design of the multicore parallel MinVerCovSmp3 program from Chapter 16. It follows the typical massively parallel with reduction pattern. The design of the cluster parallel MinVerCovClu3 program follows the cluster parallel reduction pattern (Figure 26.1). The program consists of multiple worker tasks. Each worker task consists of multiple threads. The iterations, each iteration examining a randomly chosen candidate cover, are partitioned among the workers and threads. Each thread finds the best candidate, the one with the fewest vertices, among the candidates the thread examines. Within each worker, the per-thread results are reduced together, yielding the best candidate for that worker. Each worker puts a tuple containing the worker’s semifinal result into tuple space. A separate reduction task takes these tuples, reduces the semifinal results together, and outputs the best candidate found by all the threads in all the workers.

The design of the cluster parallel minimum vertex cover MPRA program, MinVerCovClu3, is similar to that of the cluster parallel PiClu program from Chapter 22. The program’s command line arguments are a constructor expression for the graph spec to be analyzed, the number of candidate covers to examine, and a random seed.

Turning to the code in Listing 26.1, the MinVerCovClu3 program begins with the job main program (line 12). It defines a group of worker tasks using the master-worker parallel loop pattern (line 27), passing the graph spec constructor expression and the random seed to each worker task. The job main program also defines a reduction task that will run in the job process when the worker tasks have finished (lines 30–31).

The worker task class (line 49) is almost the same as the multicore parallel minimum vertex cover program from Chapter 16. The worker constructs a graph spec from the constructor expression (line 73) and uses the graph spec to initialize an adjacency matrix for the graph to be analyzed (line 72). The worker performs the worker portion of the master-worker parallel loop (line 79), examining a series of randomly chosen candidate covers and performing a reduction to determine the cover with the fewest vertices (a bitset). The
Figure 26.1. Cluster parallel minimum vertex cover program
worker puts this partially reduced result into tuple space (line 104). This works because class BitSetVbl is a tuple subclass (it extends class Tuple) as well as a reduction variable (it implements interface Vbl).

Finally, the reduction task (line 109) is similar to the one in the cluster parallel \( \pi \) estimating program. The reduction task reads the bitset semifinal results put into tuple space by the worker tasks, reduces the semifinal results together into the final overall bitset result, and prints the result.

Note that the MinVerCovClu3 program does very little network communication. When each worker task is launched, the graph spec to be analyzed is sent to the worker as a command line argument; but the graph spec is just a short string. When each worker finishes, the worker sends its semifinal result to the reduction task via tuple space; but each worker’s tuple contains just one bitset with a single bit for each vertex in the graph. On the other hand, the MinVerCovClu3 program typically does a massive amount of computation, examining perhaps millions or billions of candidate covers. Because the computation time is so much larger than the communication time, the MinVerCovClu3 program ought to exhibit good scaling behavior.

To study the MinVerCovClu3 program’s weak scaling performance, I ran the program on the tardis cluster on five different random graphs, with 50, 100, 150, 200, and 250 vertices, each graph with a density of 0.2—the same graphs as I used to study the MinVerCovSmp3 program in Chapter 16. On one core I ran the program with 25 million iterations. I scaled the program up from one to ten workers (12 to 120 cores). I increased the number of iterations in proportion to the number of cores: 300 million iterations on 12 cores, 600 million iterations on 24 cores, and so on, up to three billion iterations on 120 cores. Figure 26.2 plots the running times, sizeups, and efficiencies I observed. The program experiences fairly good weak scaling efficiencies, about 80 percent or better, out to 120 cores.

An exhaustive search program would require \( 2^{50} \) to \( 2^{250} \) iterations to find the exact minimum vertex covers for these 50- to 250-vertex graphs—far too long to be practical. The heuristic search MinVerCovClu3 program found approximate solutions with about one to three minutes of computation, depending on the graph’s size. These are the same running times as the multicore parallel MinVerCovSmp3 program in Chapter 16, except now I am scaling up to 120 cores across all the nodes in the cluster.

How close were the approximate solutions to the exact solutions? There’s no way to know without doing an exhaustive search. However, in general, the size of the minimum vertex cover the heuristic search finds does go down a bit as the number of iterations increases. The table below shows the sizes of the solutions the MinVerCovClu3 program found for each graph after doing 25 million iterations and after doing three billion iterations:
package edu.rit.pj2.example;
import edu.rit.pj2.Job;
import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Task;
import edu.rit.pj2.vbl.BitSetVbl;
import edu.rit.util.BitSet;
import edu.rit.util.GraphSpec;
import edu.rit.util.Instance;
import edu.rit.util.IntAction;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
public class MinVerCovClu3
extends Job
{
    // Job main program.
    public void main
    (String[] args)
    throws Exception
    {
        // Parse command line arguments.
        if (args.length != 3) usage();
        String ctor = args[0];
        long seed = Long.parseLong (args[1]);
        long N = Long.parseLong (args[2]);

        // Set up a task group of K worker tasks.
        masterFor (0, N - 1, WorkerTask.class) .args (ctor, ""+seed);

        // Set up reduction task.
        rule() .atFinish() .task (ReduceTask.class)
            .runInJobProcess();
    }

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println ("Usage: java pj2 [workers=<K>] " +
            "edu.rit.pj2.example.MinVerCovClu3 \"<ctor>\" <seed> <N>\");
        System.err.println ("<K> = Number of worker tasks (default " +
            "1)\");
        System.err.println ("<ctor> = GraphSpec constructor " +
            "expression\");
        System.err.println ("<seed> = Random seed\");
        System.err.println ("<N> = Number of trials\");
        terminate (1);
    }

    // Worker task class.
    private static class WorkerTask
    extends Task
    {
        // Random seed.
        long seed;

        // Graph being analyzed.
        AMGraph graph;
        int V;
    }
}

Listing 26.1. MinVerCovClu3.java (part 1)
Figure 26.2. MinVerCovClu3 weak scaling performance metrics
```java
// Minimum vertex cover.
BitSetVbl minCover;

// Worker task main program.
public void main
(String[] args)
throws Exception
{
    // Parse command line arguments.
    String ctor = args[0];
    seed = Long.parseLong (args[1]);

    // Set up adjacency matrix.
    graph = new AMGraph
        ((GraphSpec) Instance.newInstance (ctor));
    V = graph.V();

    // Check randomly chosen candidate covers.
    minCover = new BitSetVbl.MinSize (new BitSet (V));
    minCover.bitset.add (0, V);
    workerFor() .exec (new LongLoop()
    {
        BitSetVbl thrMinCover;
        BitSet candidate;
        Random prng;
        RandomSubset rsg;
        public void start()
        {
            thrMinCover = threadLocal (minCover);
            candidate = new BitSet (V);
            prng = new Random (seed + 1000*taskRank() + rank());
            rsg = new RandomSubset (prng, V, true);
        }
        public void run (long i)
        {
            candidate.clear();
            rsg.restart();
            while (! graph.isVertexCover (candidate))
                candidate.add (rsg.next());
            if (candidate.size() < thrMinCover.bitset.size())
                thrMinCover.bitset.copy (candidate);
        }
    });

    // Send best candidate cover to reduction task.
    putTuple (minCover);
}

// Reduction task class.
private static class ReduceTask
extends Task
{
    // Reduction task main program.
    public void main
    (String[] args)
    throws Exception
    {
```

**Listing 26.1.** MinVerCovClu3.java (part 2)
By doing weak scaling from 25 million to three billion iterations for these graphs, the program was able to shave one or two vertices off the size of the (approximate) minimum vertex cover. In other words, the parallel program got a more accurate solution in roughly the same amount of time as the sequential program. Because I scaled the cluster parallel MinVerCovClu3 program up to even more cores than the multicore parallel MinVerCovSmp3 program in Chapter 16 (120 cores instead of 12 cores), the cluster parallel program was able to get even better solutions (smaller covers) than the multicore parallel program.

<table>
<thead>
<tr>
<th>Graph size (vertices)</th>
<th>Cover size, 25 million iterations</th>
<th>Cover size, three billion iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>38</td>
<td>36</td>
</tr>
<tr>
<td>100</td>
<td>88</td>
<td>86</td>
</tr>
<tr>
<td>150</td>
<td>137</td>
<td>136</td>
</tr>
<tr>
<td>200</td>
<td>188</td>
<td>186</td>
</tr>
<tr>
<td>250</td>
<td>238</td>
<td>236</td>
</tr>
</tbody>
</table>

Listing 26.1. MinVerCovClu3.java (part 3)