Chapter 15
Exhaustive Search

- 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
Imagine a building with a number of intersecting straight corridors. To improve security, you want to install surveillance cameras—the black hemispherical ceiling mounted kind that can scan in all directions—so that you can monitor every corridor. But to save money, you want to install as few cameras as possible while still covering every corridor. Here’s the building’s floor plan. How many cameras do you need, and where do you put them?

A little thought should convince you that one, two, or even three cameras can’t cover all the corridors. But four cameras can do the job, if properly placed:

This is an example of the Minimum Vertex Cover Problem, a well-known problem in graph theory. A graph is a set of vertices plus a set of edges connecting pairs of vertices. A graph is often drawn with circles depicting the vertices and lines depicting the edges. Here is a graph representing the building floor plan; the vertices are the corridor intersections, the edges are the corridor segments:
Chapter 15. Exhaustive Search

A vertex cover of a graph is a subset of the vertices such that every edge in the graph is attached to at least one vertex in the subset. A minimum vertex cover is a vertex cover with as few vertices as possible. For the above graph, a minimum vertex cover is the subset \{1, 3, 5, 7\}.

In this chapter and the next, I’m going to build some parallel programs to solve the Minimum Vertex Cover Problem. These programs will further illustrate features of the Parallel Java 2 Library, including parallel loops and reduction variables. These programs will also illustrate bitsets, a technique that utilizes machine instructions, rather than multiple threads, to achieve parallelism.

The first program finds a minimum vertex cover via exhaustive search. The program considers every possible subset of the set of \(V\) vertices as potential candidates. For each candidate, the program checks whether the candidate is in fact a cover. Of the candidates that are covers, the program keeps whichever one has the fewest elements. At the end, the candidate left standing is a minimum vertex cover. (I say “a” minimum vertex cover because there might be more than one vertex cover with the same minimum size. In that case, I don’t care which one the program finds.)

This exhaustive search process is guaranteed to find a minimum vertex cover. However, the program has to look at all subsets of the set of vertices, and there are \(2^V\) possible subsets. For all but the smallest graphs, the program is going to take a long time to run. I’ll need to have a speedy algorithm, and then I’ll want to run it on a multicore parallel computer. Even so, for graphs with more than around 40 or so vertices, the program is going to take too much time to be useful. Solving the minimum vertex cover problem for larger graphs will require a different approach. (See the next chapter.)

Before I can write the minimum vertex cover program, I need to devise a way for the program to obtain the graph to be analyzed. I’ll do this with a graph specification, or graph spec, object. Listing 15.1 gives the interface for a graph spec, namely interface edu.rit.util.GraphSpec in the Parallel Java 2 Library. I can call the graph spec’s \(V()\) and \(E()\) methods to discover how many vertices and edges the graph has. The graph spec acts as a standard Java iterator over the edges in the graph. Each time I call the graph spec’s next() method, I get another edge; the returned GraphSpec.Edge object identifies the vertices the edge connects.

Class edu.rit.util.RandomGraph in the Library (Listing 15.2) is a graph spec class. It implements interface GraphSpec (line 4). The RandomGraph constructor arguments (line 16) are the number of vertices \(V\), the number of edges \(E\), and a seed for a pseudorandom number generator. Class RandomGraph generates a graph with \(V\) vertices and with \(E\) edges chosen uniformly at random from the set of all possible edges. However, note that class RandomGraph has no data structures to store the actual graph. Rather, when the next() method is called (line 77), the object generates the random graph on
the fly, and it is up to the caller to store the graph’s edges in whatever data structures are deemed necessary. This decouples the code for specifying a graph from the data structures for storing a graph. For further information about how class RandomGraph works, see the “Under the Hood” section below.

I need a way for the minimum vertex cover program to create a graph spec object, such as an instance of class RandomGraph. If I were hard-coding this into the program, I’d write a constructor expression like this:

```
GraphSpec gspec = new RandomGraph (10, 20, 142857);
```

This would create a random graph spec with 10 vertices and 20 edges. However, I don’t want to hard-code the graph spec object into the program. I want to specify the graph spec constructor expression at run time on the command line, like this:

```
$ java pj2 edu.rit.pj2.example.MinVerCovSmp "edu.rit.util.RandomGraph(10,20,142857)"
```

The Parallel Java 2 Library includes class edu.rit.util.Instance, which lets me do what I want. The `Instance.newInstance()` method takes as its argument a constructor expression string, like the one enclosed in quotes above; parses the string to get the fully-qualified class name and the constructor arguments; constructs a new instance of the specified class, passing the specified arguments to the constructor; and returns a reference to the new instance. There are restrictions on the types of constructor arguments the `newInstance()` method can handle; see the Javadoc for class `edu.rit.util.Instance`. For further information about how class Instance works, see the “Under the Hood” section below.

Using this technique, the same minimum vertex cover program can analyze any kind or size of graph, without needing to change the program’s source code. Simply define a class that implements the GraphSpec interface, with a constructor that causes the class to generate the desired vertices and edges. Then specify the appropriate constructor expression on the program’s command line.

Next, I need a data structure to represent a graph in the program. Like many abstract data types, there are several ways to implement a graph data structure. The appropriate data structure depends on the problem; there is no one-size-fits-all graph data structure. Graph data structures include:

- **Adjacency matrix.** This is a $V \times V$ matrix of 0s and 1s, with rows and columns indexed by the vertices. Matrix element $[r, c]$ is 1 if vertices $r$ and $c$ are adjacent (if there is an edge between vertex $r$ and vertex $c$); element $[r, c]$ is 0 otherwise.
- **Adjacency list.** This is an array of $V$ lists, indexed by the vertices. Array element $[i]$ is a list of the vertices adjacent to vertex $i$. 
package edu.rit.util;
import java.util.Iterator;
public interface GraphSpec
    extends Iterator<GraphSpec.Edge>
{
    // Class GraphSpec.Edge encapsulates one edge in a graph.
    public static class Edge
    {
        // First vertex.
        public int v1;
        // Second vertex.
        public int v2;
        // Construct a new uninitialized edge.
        public Edge()
        {
        }
        // Construct a new edge.
        public Edge
        (int v1,
         int v2)
        {
            this.v1 = v1;
            this.v2 = v2;
        }
    }
    // Get the number of vertices in this graph specification.
    public int V();
    // Get the number of edges in this graph specification.
    public int E();
    // Reset this graph specification.
    public void reset();
    // Determine if there are more edges.
    public boolean hasNext();
    // Get the next edge.
    public GraphSpec.Edge next();
}

Listing 15.1. GraphSpec.java

package edu.rit.util;
import java.util.NoSuchElementException;
public class RandomGraph
    implements GraphSpec
{
    private int V;
    private int E;
    private Random prng;
    private GraphSpec.Edge edge;
    private int v1;
    private int v2;
    private int needed;

Listing 15.2. RandomGraph.java (part 1)
- **Edge list.** This is simply a list of the edges, each edge consisting of a pair of vertices.

As will become apparent shortly, the best data structure for the minimum vertex cover exhaustive search program is the adjacency matrix. Here again is the building floor plan graph:

![Building floor plan graph]

And here is the adjacency matrix corresponding to that graph:

$$
\begin{array}{cccccccc}
8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 2 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 3 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 4 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 5 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 6 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 7 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 8 \\
\end{array}
$$

Note that the matrix is symmetric; an edge between vertex $r$ and vertex $c$ appears both as element $[r, c]$ and element $[c, r]$. (The reason I’ve numbered the columns from right to left will also become apparent shortly.)

Now I need a way to decide if a particular candidate set of vertices is a cover. For example, consider the set $\{4, 6, 8\}$. I shade in rows 4, 6, and 8 and columns 4, 6, and 8 of the adjacency matrix:

$$
\begin{array}{cccccccc}
8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 2 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 3 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 4 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 5 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 6 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 7 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 8 \\
\end{array}
$$

All the 1s in the shaded cells represent edges that are attached to at least one of the vertices in the set $\{4, 6, 8\}$. However, there are several 1s in cells that
private int available;

// Construct a random graph specification with the given seed.
public RandomGraph
(int V,
 int E,
 long seed)
{
 this (V, E, new Random (seed));
}

// Construct a random graph specification with the given pseudorandom number generator.
public RandomGraph
(int V,
 int E,
 Random prng)
{
 if (V < 0)
 throw new IllegalArgumentException (String.format
 ("RandomGraph(): V = %d illegal", V));
 if (E < 0)
 throw new IllegalArgumentException (String.format
 ("RandomGraph(): E = %d illegal", E));
 if (E > maxE (V))
 throw new IllegalArgumentException (String.format
 ("RandomGraph(): V = %d, E = %d illegal", V, E));
 if (prng == null)
 throw new NullPointerException
 ("RandomGraph(): prng is null");

 this.V = V;
 this.E = E;
 this.prng = prng;
 this.edge = new GraphSpec.Edge();
 reset();
}

// Get the number of vertices in this graph.
public int V()
{
 return V;
}

// Get the number of edges in this graph.
public int E()
{
 return E;
}

// Reset this graph specification.
public void reset()
{
 v1 = -1;
 v2 = V - 1;
 needed = E;
 available = maxE (V);
}

Listing 15.2. RandomGraph.java (part 2)
are not shaded; these represent edges that are attached to none of the vertices in the set. Therefore, \{4, 6, 8\} is not a cover.

On the other hand, the candidate vertex set \{1, 3, 5, 7\} is a cover, as is apparent from the adjacency matrix—all the 1s are in shaded cells:

\[
\begin{array}{cccccccc}
8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

I need to express this procedure in a way that can be coded in a computer program. I don’t need to look at the matrix rows corresponding to vertices in the candidate set (shaded rows); all 1s in those rows are automatically covered. I only need to look at the matrix rows corresponding to vertices not in the candidate set (unshaded rows). I view each unshaded row as itself a vertex set, with 1s indicating which vertices are in the row set. All the 1s in an unshaded row are covered if the row set is a subset of the candidate set. For example, consider row 0. The row set \{1, 3\} is a subset of the candidate set \{1, 3, 5, 7\}; therefore the 1s in row 0 are covered. The same is true of rows 2, 4, 6, and 8. Therefore, \{1, 3, 5, 7\} is a cover for the whole graph.

So in the program code, I don’t want to implement the adjacency matrix as an actual matrix. Rather, I want to implement it as an array of rows, where each row is a vertex set. And I can see that I’ll need at least two operations on a vertex set: an operation to determine if a vertex is or is not a member of a vertex set (to decide if a given row is not in the candidate set); and an operation to determine if one vertex set is a subset of another (to decide if a given row set is a subset of the candidate set).

Now I need a way to represent a vertex set in the program. For sets with a limited number of possible elements, a \textit{bitset} implementation is attractive. What’s a bitset? Consider the first row of the above adjacency matrix:

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Now take away the cell borders:

00001010

This looks suspiciously like a binary number, which is how a computer stores an integer. A bitset uses an integer to indicate which elements are members of the set. Suppose the set can contain \(n\) different elements. The elements are numbered from 0 to \(n - 1\). The bit positions of the integer are likewise num-
// Determine if there are more edges.
public boolean hasNext()
{
    return needed > 0;
}

// Get the next edge.
public GraphSpec.Edge next()
{
    if (! hasNext())
        throw new NoSuchElementException
            ("RandomGraph.next(): No more edges");

    for (;;)
    {
        ++ v2;
        if (v2 == V)
        {
            ++ v1;
            v2 = v1 + 1;
        }
        if (prng.nextDouble() < (double)needed/(double)available)
            {
                -- needed;
                -- available;
                break;
            }
        else
            {
                -- available;
            }
    }
    edge.v1 = v1;
    edge.v2 = v2;
    return edge;
}

// Unsupported operation.
public void remove()
{
    throw new UnsupportedOperationException
        ("RandomGraph.remove(): Unsupported operation");
}

// Returns the maximum number of edges for a graph with V vertices.
private static int maxE
(int V)
{
    return Math.max (0, V*(V - 1)/2);
}

Listing 15.2. RandomGraph.java (part 3)
bered from 0 to \( n - 1 \), with bit position 0 being the least significant (rightmost) bit and bit position \( n - 1 \) being the most significant (leftmost) bit. In the integer, bit \( i \) is on (1) if element \( i \) is a member of the set; bit \( i \) is off (0) if element \( i \) is not a member of the set. Using type `int`, a bitset can represent a set with up to 32 elements; using type `long`, up to 64 elements.

With a bitset representation, I can do set operations on all the elements simultaneously with just one or two integer operations. These usually involve the bitwise Boolean operations on integers—bitwise and (\&), bitwise or (|), bitwise exclusive-or (^), bitwise complement (~), and bitwise shifts (<<, >>, >>>). In effect, bitset operations utilize the CPU’s integer functional unit to manipulate all 32 or 64 set elements in parallel with just a few machine instructions. Consequently, operations on a bitset are quite fast.

The Parallel Java 2 Library includes bitset classes in package edu.rit.util. Class `BitSet32` holds up to 32 elements; class `BitSet64`, up to 64 elements. The Library also includes bitset reduction variable classes in package edu.rit.pj2.vbl, namely classes `BitSet32Vbl` and `BitSet64Vbl`, as well as several subclasses that do various reduction operations on bitsets. For further information about how the bitset classes work, see the “Under the Hood” section below.

To represent a vertex set, my program will use class `BitSet64`. Thus, the program could handle adjacency matrices with as many as \( V = 64 \) vertices. This is more than large enough to accommodate an exhaustive search program that has to look at \( 2^V \) subsets. My program will also use class `BitSet64Vbl` to do parallel reductions.

Now that I have a class for a bitset, I can write a class to hold the graph’s adjacency matrix, namely class `AMGraph64` (Listing 15.3). As discussed earlier, this class represents the adjacency matrix as a singly dimensioned array of bitsets (line 11). The constructor is given a graph spec (line 20) and calls the `set()` method (line 27). The `set()` method queries the graph spec to get the number of vertices \( V \) (line 30) and initializes the adjacency matrix to an array of \( V \) vertex sets (bitsets), all initially empty (lines 36–38). The `set()` method then iterates over the edges (lines 45–47) and, for each edge, turns on elements \([v1, v2]\) and \([v2, v1]\) in the adjacency matrix, thus ensuring the matrix is symmetric (lines 48–49).

Why put the code to initialize the adjacency matrix in its own public `set()` method? Why not just put this code in the constructor? For flexibility. In the future, I might want to write a program in which the same adjacency matrix object will hold different graphs at different times. The `set()` method lets me change the contents of an existing adjacency matrix object without needing to construct a new adjacency matrix object. Soon we will see a very good reason for reusing existing objects as we study the parallel minimum vertex cover program.
package edu.rit.pj2.example;
import edu.rit.util.BitSet64;
import edu.rit.util.GraphSpec;
public class AMGraph64
{
    // Number of vertices.
    private int V;

    // The graph's adjacency matrix. adjacent[i] is the set of
    // vertices adjacent to vertex i.
    private BitSet64[] adjacent;

    // Construct a new uninitialized adjacency matrix.
    public AMGraph64()
    {
    }

    // Construct a new adjacency matrix for the given graph
    // specification.
    public AMGraph64
    (GraphSpec gspec)
    {
        set (gspec);
    }

    // Set this adjacency matrix to the given graph specification.
    public void set
    (GraphSpec gspec)
    {
        V = gspec.V();
        if (0 > V || V > 64)
            throw new IllegalArgumentException (String.format
                ("AMGraph64.set(): V = %d illegal", V));
        if (adjacent == null || adjacent.length != V)
        {
            adjacent = new BitSet64[V];
            for (int i = 0; i < V; ++ i)
                adjacent[i] = new BitSet64();
        }
        else
        {
            for (int i = 0; i < V; ++ i)
                adjacent[i].clear();
        }
        while (gspec.hasNext())
        {
            GraphSpec.Edge edge = gspec.next();
            adjacent[edge.v1].add (edge.v2);
            adjacent[edge.v2].add (edge.v1);
        }
    }

    // Get the number of vertices in this graph.
    public int V()
    {
        return V;
    }
The AMGraph64 class also has a method to check whether a given bitset is a vertex cover (line 68). This method uses the procedure described earlier. The method calls the bitset’s contains() and isSubsetOf() methods to do its check. If at any point the method discovers the candidate is not a cover, the method stops immediately without checking the remaining vertices, which saves time.

Why break the design into so many classes? The short answer is that this is standard object oriented design practice. The more cogent answer is that I am going to be writing several minimum vertex cover programs, in this and later chapters. While each program will use a different technique to find a minimum vertex cover, the underlying data structures will be the same in all the programs. To avoid writing the same data structure code over and over again in every program, I put the data structures in their own classes and reused the classes. If I need to change one of the data structure classes—if I discover a bug in the data structure, say—I only need to touch that one class; I don’t need to touch the multiple programs that use the class. This reduces the effort needed to maintain the programs and reduces the likelihood of introducing new bugs.

Finally, I can write the first multicore parallel minimum vertex cover program, edu.rit.pj2.example.MinVerCovSmp (Listing 15.4). The program begins by constructing a graph spec object for the graph to be analyzed (line 30), then using the graph spec to create the graph itself, represented as an adjacency matrix (line 29).

The program is now ready to do a parallel loop over all possible subsets of the set of vertices, with each parallel team thread examining a different portion of the subsets. The program will also use parallel reduction. Each team thread will find its own minimum vertex cover among the vertex subsets the team thread examines. When the parallel loop finishes, these per-thread minimum vertex covers will be reduced together into the overall minimum vertex cover.

Following the parallel reduction pattern, the program creates a global reduction variable minCover of type BitSet64Vbl.MinSize (line 34). This subclass’s reduction operation combines two bitsets (covers) by keeping whichever cover has fewer elements (vertices), which is what I need to do to find a minimum cover. The add(0,V) method initializes minCover to contain all the vertices, from 0 through V – 1 (line 35). The set of all vertices is obviously a cover. Any other cover the program finds will have fewer vertices and will replace this initial cover when the reduction happens. The program sets the variable full to the bitmap corresponding to this set of all vertices (line 36).

Next the program does a parallel loop over all the bitsets from the empty set (0L) to the full set (full). Along the way, the loop index visits every possible subset of the set of vertices. For example, with V = 4, here are the bitsets the loop visits (in binary):
Chapter 15. Exhaustive Search

173

// Determine if the given vertices are adjacent in this graph.
public boolean isAdjacent
(int v1,
 int v2)
{
 return adjacent[v1].contains (v2);
}

// Returns true if the given vertex set is a vertex cover for
// this graph.
public boolean isVertexCover
(BitSet64 vset)
{
 boolean covered = true;
 for (int i = 0; covered && i < V; ++ i)
 if (! vset.contains (i))
 covered = adjacent[i].isSubsetOf (vset);
 return covered;
}

Listing 15.3. AMGraph64.java (part 2)

package edu.rit.pj2.example;
import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Task;
import edu.rit.pj2.vbl.BitSet64Vbl;
import edu.rit.util.BitSet64;
import edu.rit.util.GraphSpec;
import edu.rit.util.Instance;
import edu.rit.util.IntAction;
public class MinVerCovSmp
extends Task
{
 // Graph being analyzed.
 AMGraph64 graph;
 int V;

 // Minimum vertex cover.
 BitSet64Vbl minCover;

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

 // Construct graph spec, set up adjacency matrix.
 graph = new AMGraph64
 ((GraphSpec) Instance.newInstance (ctor));
 V = graph.V();
 if (V > 63) error ("Too many vertices");

Listing 15.4. MinVerCovSmp.java (part 1)
This is another reason to use bitsets; it makes looping over every possible subset a simple matter of incrementing an integer loop index. However, I have to be careful. The maximum positive value for a loop index of type `long` is $2^{63} - 1$. For this reason, I had to restrict the number of vertices $V$ to be 63 or less (line 32).

Inside the parallel loop, the program declares a thread-local variable `thrMinCover` of type `BitSet64Vbl` (line 39) linked to the global reduction variable (line 42). `thrMinCover` will hold the best (smallest-size) cover the parallel team thread has seen so far. In the loop body, the program creates a candidate vertex set from the loop index bitset (line 46). If the candidate is smaller than (has fewer vertices than) the best cover seen so far, and if the candidate is in fact a cover, the program copies the candidate into the thread’s minimum cover variable (lines 47–49). Otherwise, the thread’s minimum cover variable remains as it was.

Note that on line 47, because of the “short-circuit” semantics of the logical and operator `&&`, if the candidate vertex set is not smaller than the best cover seen so far, the program will not even bother to check whether the candidate is a cover. This saves time.

When the parallel loop finishes, each parallel team thread’s thread-local minimum cover variable contains the smallest cover among the vertex subsets the team thread examined. The thread-local minimum cover variables are automatically reduced together into the global minimum cover variable. As mentioned before, the reduction operation is to keep the cover that has the fewer elements. The global minimum cover variable ends up holding the smallest cover among all the possible vertex subsets. Finally, the program prints this minimum vertex cover (lines 54–63).

I ran the sequential MinVerCovSeq program on one core and the multi-core parallel MinVerCovSmp program on one to 12 cores of a tardis node (strong scaling) and measured the running times. I ran the programs on five random graphs, with $V = 31$ to 35 and $E = 310$ to 350. Here are examples of the minimum vertex cover the programs found for the $V = 31$ test case:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSeq "edu.rit.util.RandomGraph(31,310,14285731)"
Cover = 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 18 19 20 21 22 23
Size = 26
Job 1 makespan 134782 msec
```
// Check all candidate covers (sets of vertices).
minCover = new BitSet64Vbl.MinSize();
minCover.bitset.add (0, V);
long full = minCover.bitset.bitmap();
parallelFor (0L, full) .exec (new LongLoop()
{
    BitSet64Vbl thrMinCover;
    public void start()
    {
        thrMinCover = threadLocal (minCover);
    }
public void run (long elems)
    {
        BitSet64 candidate = new BitSet64 (elems);
        if (candidate.size() < thrMinCover.bitset.size() &&
            graph.isVertexCover (candidate))
            thrMinCover.bitset.copy (candidate);
    }
});

// Print results.
System.out.printf ("Cover =")
minCover.bitset.forEachItemDo (new IntAction()
{
    public void run (int i)
    {
        System.out.printf (" %d", i);
    }
});
System.out.println();
System.out.printf ("Size = %d%n", minCover.bitset.size());

// Print an error message and exit.
private static void error
(String msg)
{
    System.err.printf ("MinVerCovSmp: %s%n", msg);
terminate (1);
}

// Print a usage message and exit.
private static void usage()
{
    System.err.println ("Usage: java pj2 " +
        "edu.rit.pj2.example.MinVerCovSmp " +
        "ctor>\"");
    System.err.println ("<ctor> = GraphSpec constructor " +
        "expression");
terminate (1);
}

Listing 15.4. MinVerCovSmp.java (part 2)
$ java pj2 debug=makespan threads=12 \\
edu.rit.pj2.example.MinVerCovSmp \\
"edu.rit.util.RandomGraph(31,310,14285731)"

Cover = 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 18 19 20 21 22 23 
24 25 27 29 30
Size = 26
Job 1 makespan 24742 msec

Figure 15.1 plots the program’s running times, speedups, and efficiencies. Fitting the running time model to the measurements gives this formula for the running time $T$ as a function of the problem size $N (= 2^V)$ and the number of cores $K$:

$$T = (0.0692 + 4.60 \times 10^{-10} N) \cdot K + (6.25 \times 10^{-8} N) / K.$$  \hspace{1cm} (15.1)

While the second term in the formula decreases as $K$ increases, the first term in the formula increases as $K$ increases. This in turn causes the speedups and efficiencies to droop drastically, as is apparent in Figure 15.1.

This scaling behavior comes from the way the program uses vertex set objects. The first statement in the parallel loop body (line 46) constructs a new instance of class BitSet64 and assigns its reference to the candidate local variable. The first term in formula (15.1) is proportional to the number of candidate vertex sets examined, $N$—because each candidate causes a new object to be constructed. Constructing an instance requires allocating storage from the JVM’s heap and setting the new object’s fields to their default initial values. Continually constructing new objects takes time. For a graph with 31 vertices, $2^{31}$ vertex set objects—over two billion!—have to be constructed. Even if the constructor takes only a few nanoseconds, the time spent in the constructor adds up to a noticeable amount.

Furthermore, once the parallel loop body’s `run()` method returns, the local candidate reference goes away, and the vertex set object becomes garbage. Eventually the heap fills up with garbage objects, and the JVM has to run the garbage collector. This takes more time. Worse, when the JVM runs the garbage collector, the JVM typically suspends execution of other threads in the program; thus, the time spent collecting garbage is typically part of the program’s sequential fraction. (I suspect garbage collection is the chief reason for MinVerCovSmp’s scaling behavior.) It would be better all around if the program didn’t create and discard an object on every loop iteration.

To address this problem, I wrote another version of the program. MinVerCovSmp2 is the same as MinVerCovSmp, except I changed the parallel loop body slightly; the differences are highlighted:
Chapter 15. Exhaustive Search

Figure 15.1. MinVerCovSmp strong scaling performance metrics
parallelFor (0L, full) .exec (new LongLoop()
{
  BitSet64Vbl thrMinCover;
  BitSet64 candidate;
  public void start()
  {
    thrMinCover = threadLocal (minCover);
    candidate = new BitSet64();
  }
  public void run (long elems)
  {
    candidate.bitmap (elems);
  ...
This time, the candidate variable is a field of the loop body subclass rather
than a local variable of the run() method. A new vertex set object is created,
one only, in the start() method and is assigned to the candidate variable.
In the run() method, each loop iteration reuses the existing vertex set object
by calling the candidate variable’s bitmap() method, rather than construct-
ing a new object. The bitmap() method replaces the candidate variable’s ele-
ments with those of the loop index, elems. Coding the loop this way elimi-
nates the repeated object creation, thus eliminating the garbage collection as
well.

I ran the sequential MinVerCovSeq2 program on one core and the multi-
core parallel MinVerCovSmp2 program on one to 12 cores of a tardis node
and measured the running times. I ran the programs on five random graphs,
with $V = 31$ to $35$ and $E = 310$ to $350$—the same graphs as the original pro-
gram version. Here are examples of the minimum vertex cover the programs
found for the $V = 31$ test case:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSeq2 \
  "edu.rit.util.RandomGraph(31,310,14285731)"
Cover = 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 18 19 20 21 22 23
Size = 26
Job 1 makespan 24859 msec
```

```
$ java pj2 debug=makespan threads=12 \
  edu.rit.pj2.example.MinVerCovSmp2 \
  "edu.rit.util.RandomGraph(31,310,14285731)"
Cover = 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 18 19 20 21 22 23
Size = 26
Job 1 makespan 2083 msec
```

The new version’s running times are significantly—about 85 percent—
smaller than the original version’s, due to eliminating all the object creation
and garbage collection. Fitting the running time model to the measurements
gives this formula for the new version’s running time:

$$T = (1.08 \times 10^{-12} N)/K .$$  \hspace{1cm} (15.2)
Figure 15.2. MinVerCovSmp2 strong scaling performance metrics
Now there is no longer a large increase in the running time as $K$ increases. The new version’s plots (Figure 15.2) show that the efficiencies degrade hardly at all as the number of cores increases.

Interestingly, Figure 15.2 shows that the efficiencies are all greater than 1.0. The parallel version of the program seemingly executes computations at a greater rate than the sequential version. As I mentioned in Chapter 13, I suspect that this is because the JIT compiler has done a better job of optimizing the parallel version; but I don’t know why the JIT compiler does this.

The moral? It’s okay to use objects in Java parallel programs. However, you have to be careful, and you have to be aware of the consequences. Avoid repeatedly constructing and discarding objects. Rather, if at all possible, reuse existing objects by changing their state as necessary. I realize this might go counter to what you’ve been taught or how you’re accustomed to designing Java programs. However, better program performance trumps conventional design wisdom.

Another moral is that investigating the parallel program’s scalability, as I did in this chapter, can yield insights that lead to design changes that improve the program’s performance. I would not have realized what was going on with object creation and garbage collection if I had not measured the program’s performance.

**Under the Hood**

Let’s look more closely at how bitsets are implemented. Class `edu.rit.util.BitSet32` provides a set with elements from 0 to 31. The set elements are stored in a value of type `int`, which has 32 bits. Each bit position of the integer corresponds to a different set element: bit position 0 (the rightmost, or least significant bit) to element 0, bit position 1 to element 1, and so on. Each bit’s value is a 1 if the set contains the corresponding element; each bit’s value is a 0 if the set does not contain the corresponding element.

This data structure can also be viewed as a mapping from elements (0 through 31) to Boolean values (0 or 1). Each mapping occupies one bit, and thirty-two such mappings are crammed into an `int`. It is a map composed of bits, or a bitmap.

Here is the bitmap representation of the set $\{1, 3, 5, 7\}$:

```
00000000000000000000000010101010
```

Or, expressed as a 32-bit binary integer:

```
00000000000000000000000010101010
```

Class `BitSet32`’s `contains()` method checks whether a bitset contains a given element $e$. It does this by forming a mask for $e$, namely an integer with
a 1 at bit position \( e \) and 0s elsewhere. The mask is generated by the expression \((1 << e)\). The value 1 has a 1 at bit position 0 and 0s elsewhere; left-shifting the value 1 by \( e \) bit positions moves the 1 to bit position \( e \) and leaves 0s elsewhere. The method then does a bitwise Boolean “and” operation between the bitmap and the mask. The resulting value has a 0 bit wherever the mask has a 0 bit, namely in all bit positions except \( e \). The resulting value is the same as the bitmap in bit position \( e \), where the mask has a 1 bit. If the resulting value is 0, namely all 0 bits, then the bitmap at bit position \( e \) is 0, meaning the set does not contain element \( e \). If the resulting value is not 0, then the bitmap at bit position \( e \) is 1, meaning the set does contain element \( e \). Therefore, the expression \(((\text{bitmap} \& (1 << e)) \neq 0)\) is true if the set contains element \( e \). Here is an example of the \texttt{contains(7)} method called on the set \{1, 3, 5, 7\}:

```
00000000000000000000000010101010  bitmap
00000000000000000000000000000001  1
00000000000000000000000010000000  1 << 7
00000000000000000000000000000000  \text{bitmap} \& (1 << 7)
```

The resulting value is not equal to 0, so the method returns true, signifying that \{1, 3, 5, 7\} does contain 7. On the other hand, here is the \texttt{contains(9)} method called on the set \{1, 3, 5, 7\}:

```
00000000000000000000000010101010  bitmap
00000000000000000000000000000001  1
00000000000000000000001000000000  1 << 9
00000000000000000000000000000000  \text{bitmap} \& (1 << 9)
```

The resulting value is equal to 0, so the method returns false, signifying that \{1, 3, 5, 7\} does not contain 9.

Class BitSet32’s \texttt{add(e)} method works in a similar fashion. It forms a mask for \( e \), then it does a bitwise Boolean “or” operation between the bitmap and the mask. The resulting value has a 1 at bit position \( e \), where the mask has a 1 bit, regardless of what was in the bitmap before. The resulting value’s other bit positions, where the mask has 0 bits, are the same as those of the original bitmap. This new value replaces the bitmap’s original value. The bitmap ends up the same as before, except bit position \( e \) has been set to 1; that is, element \( e \) has been added to the set.

Class BitSet32’s \texttt{isSubsetOf()} method forms the bitwise Boolean “and” of the two bitmaps. If the result is equal to the first bitmap, then every bit position that is a 1 in the first map is also a 1 in the second bitmap; that is, every element of the first set is also an element of the second set; that is, the first set is a subset of the second set. Otherwise, the first set is not a subset of the second set.

Class BitSet64 is the same as class BitSet32, except the bitmap uses a 64-bit long integer (type \texttt{long}) to store the bitmap.
All the methods in both bitset classes are implemented like those described above, with just a few operations on the integer bitmaps. As mentioned before, each such operation manipulates all the bits of the bitmap at the same time—that is, in parallel. The parallelism comes, not from multiple threads, but from the CPU’s integer functional unit’s inherent parallelism. Multithreading is not the only way to get a bunch of things to happen at the same time!

I used class RandomGraph to create the graphs for the running time measurements described earlier. Class RandomGraph generates a random graph with a given number of vertices $V$ and a given number of edges $E$, where the edges are chosen at random. With $V$ vertices, there are $N = V \cdot (V - 1) \div 2$ possible edges. The random graph must comprise a size-$E$ subset of the set of possible edges, where each such subset is equally likely to be chosen. Class RandomGraph achieves this by iterating over all the possible edges and choosing to add each edge with a certain probability $p = e/n$, where $e$ is the number of edges still to be added to the graph and $n$ is the number of possible edges still available. Initially, $e = E$ and $n = N$. If a certain edge is added, then both $e$ and $n$ are decremented; otherwise, only $n$ is decremented. Once $e$ goes to zero, the random graph is complete. In this way, class RandomGraph picks edges on the fly without needing to store the actual random graph.

The MinVerCovSmp program uses the `Instance.newInstance()` method to construct an object that implements interface GraphSpec, using which the program obtains the graph to be analyzed. The `newInstance()` method in turn uses Java Reflection in package java.lang.reflect.

The `newInstance()` method’s argument is a constructor expression string consisting of a fully qualified class name, a left parenthesis, zero or more comma-separated arguments, and a right parenthesis. The arguments can be any of the following:

- A binary, octal, decimal, or hexadecimal integer or long integer value.
- A single precision or double precision floating point value.
- A nested constructor expression for an instance of a class.
- An arbitrary string.

The `newInstance()` method gets the java.lang.Class object corresponding to the class name; gets a list of that class’s constructors as well as a list of the argument types for each constructor; and attempts to match each constructor’s argument types to the values in the constructor expression string. If it finds a match, the `newInstance()` method invokes the matching constructor, passing in the argument values, and returns the object that the constructor creates. Otherwise, the `newInstance()` method throws an exception. Java Reflection can do all this and more.

I have found this design pattern to be a very powerful way to decouple specifying a problem from solving a problem:
• Define an interface for an object that encapsulates a problem specification, like interface GraphSpec.
• Define one or more classes that implement this interface to generate various kinds of problems, like class RandomGraph.
• Write one or more problem solving programs that take a constructor expression string as a command line argument. Use the newInstance() method to create a problem specification object. Query that object to obtain the problem parameters.

Using this pattern, I can mix and match various problem specifications with various problem solving programs. The same program can solve different kinds of problems, like structured graphs, random graphs, and other kinds of graphs. The same problem can be solved by different kinds of programs, like MinVerCovSmp, MinVerCovSmp2, and other minimum vertex cover programs we will see later.

In the past, I found myself defining a textual format for problem specifications stored in a file. Then in a problem solving program, I had to write code to parse the file and extract the problem parameters. Because creating problem instances by hand was too tedious, I also had to write programs to generate a problem instance and store it in a file in the proper format.

Using the newInstance() method lets me eliminate all the file writing and parsing code, which reduces my development effort. It also lets me eliminate the files themselves and the disk storage they occupy. This is especially important for large problem instances, which would otherwise suck up lots of disk storage, and which would take a long time for the solver program to read and parse. I’d rather devote CPU cycles to actual problem solving, not file parsing.

Points to Remember

• An exhaustive search solves a problem by looking at every possible candidate solution and keeping the best one.
• However, some problems have exponentially many candidate solutions. Consequently, an exhaustive search will take too long, unless the problem size is small.
• Consider implementing a set of elements using a bitset representation. Bitsets are fast and compact.
• Use the Parallel Java 2 Library’s bitset classes in package edu.rit.util.
• Use the Parallel Java 2 Library’s bitset reduction variable classes in package edu.rit.pj2.vbl.
• Avoid repeatedly creating and discarding objects. Reuse existing objects wherever possible.
• Measure the parallel program’s performance and scalability. Derive the program’s running time formula. If necessary, use the insights gained to change the program’s design to improve its performance.
• Consider using the `Instance.newInstance()` method in the Parallel Java 2 Library to obtain an object that encapsulates a problem’s parameters.