Chapter 11
Exhaustive Search

► Part I. Preliminaries
▼ Part II. Tightly Coupled Multicore
  Chapter 2. Parallel Loops
  Chapter 3. Parallel Loop Schedules
  Chapter 4. Parallel Reduction
  Chapter 5. Reduction Variables
  Chapter 6. Load Balancing
  Chapter 7. Overlapping
  Chapter 8. Sequential Dependencies
  Chapter 9. Strong Scaling
  Chapter 10. Weak Scaling
  Chapter 11. Exhaustive Search
  Chapter 12. Heuristic Search
► Part III. Loosely Coupled Cluster
► Part IV. GPU Acceleration
► Part V. Map-Reduce
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?

![Building floor plan](image)

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:

![Cameras placement](image)

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:

![Graph representation](image)
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.)

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 \(i\).

- **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 is the adjacency matrix corresponding to the building floor plan graph:
BIG CPU, BIG DATA

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:

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

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 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:

\[
\begin{array}{cccccccc}
8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
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 bitset implementation is attractive. What’s a bitset? Consider the first row of the above adjacency matrix:

```
0 0 0 0 0 1 0 1 0
```

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 numbered 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 \texttt{int}, a bitset can represent a set with up to 32 elements; using type \texttt{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 \texttt{edu.rit.util}. Class \texttt{BitSet32} holds up to 32 elements; class \texttt{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 subclassess 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 can handle graphs 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.

Finally, I can start writing some code. Listing 11.1 is the first multicore parallel minimum vertex cover program, MinVerCovSmp. The program begins by reading the graph from a file in this format: The first pair of numbers gives $V$, the number of vertices, and $E$, the number of edges. Each subsequent pair of numbers gives two vertex numbers each in the range 0 through $V - 1$, defining an edge between those vertices. For example, here is the file for the building graph:

```
9 12
0 1 1 2 0 3 1 4
2 5 3 4 4 5 3 6
4 7 5 8 6 7 7 8
```

The program reads the file given on the command line using a Scanner (line 32). After reading $V$ and $E$, the program initializes the adjacency matrix to an array of $V$ vertex sets (bitsets), all initially empty (lines 36–38). Upon reading each subsequent pair of vertices $a$ and $b$, the program turns on elements $[a, b]$ and $[b, a]$ in the adjacency matrix, thus ensuring the matrix is symmetric (lines 39–45).

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 49). 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 $V - 1$. 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 50).
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 java.io.File;
import java.util.Scanner;
public class MinVerCovSmp
    extends Task
{
    // Number of vertices and edges.
    int V;
    int E;

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

    // Minimum vertex cover.
    BitSet64Vbl minCover;

    // Main program.
    public void main
        (String[] args)
        throws Exception
    {
        // Parse command line arguments.
        if (args.length != 1) usage();
        File file = new File (args[0]);
        // Read input file, set up adjacency matrix.
        Scanner s = new Scanner (file);
        V = s.nextInt();
        E = s.nextInt();
        if (V < 1 || V > 63) usage ("V must be >= 1 and <= 63");
        adjacent = new BitSet64 [V];
        for (int i = 0; i < V; ++ i)
            adjacent[i] = new BitSet64();
        for (int i = 0; i < E; ++ i)
            { int a = s.nextInt();
                int b = s.nextInt();
                adjacent[a].add (b);
                adjacent[b].add (a);
            }
        s.close();

        // Check all candidate covers (sets of vertices).
        minCover = new BitSet64Vbl.MinSize() .add (0, V);
        long full = minCover.bitmap();
        parallelFor (0L, full) .exec (new LongLoop()
            { BitSet64Vbl thrMinCover;
                public void start()
                {
                    thrMinCover = threadLocal (minCover);
                }
        });
    }

Listing 11.1. MinVerCovSmp.java (part 1)
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):

- 0000
- 0001
- 0010
- 0011
- 0100
- 0101
- 0110
- 0111
- 1000
- 1001
- 1010
- 1011
- 1100
- 1101
- 1110
- 1111

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 35).

Inside the parallel loop, the program declares a thread-local variable thrMinCover of type BitSet64Vbl (line 53) linked to the global reduction variable (line 56). 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 60). 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 61–63). Otherwise, the thread’s minimum cover variable remains as it was.

Note that on line 61, 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.

The program checks whether the candidate is a cover using the procedure described earlier, embodied in the isCover() method (lines 77–85). The method uses the bitset’s contains() and isSubsetOf() methods to do its check. If the method discovers the candidate is not a cover, the method stops immediately, which again 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 68–73).

To test the MinVerCovSmp program and measure its performance, I need graphs stored in files to use as input. I wrote a program to create a random graph with a given number of vertices and edges (Listing 11.2). The program is self-explanatory.
public void run (long elems) {
    BitSet64 candidate = new BitSet64 (elems);
    if (candidate.size() < thrMinCover.size() &&
        isCover (candidate))
        thrMinCover.copy (candidate);
}

// Print results.
System.out.printf ("Cover =");
for (int i = 0; i < V; ++ i)
    if (minCover.contains (i))
        System.out.printf (" %d", i);
System.out.println();
System.out.printf ("Size = %d\n", minCover.size());

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

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

// Print a usage message and exit.
private static void usage() {
    System.err.println ("Usage: java pj2 " +
        "edu.rit.pj2.example.MinVerCovSmp <file>\n");
    System.err.println ("<file> = Graph file");
    throw new IllegalArgumentException();
}

Listing 11.1. MinVerCovSmp.java (part 2)

package edu.rit.pj2.example;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
public class RandomGraph {

Listing 11.2. RandomGraph.java (part 1)
I ran the sequential MinVerCovSeq program on one core and the multi-core parallel MinVerCovSmp program on one to four 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
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSeq \  
g31.txt
Cover = 0 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 18 19 20 22 23 24 25 26 27 28
Size = 26
Job 3 makespan 289506 msec
$ java pj2 debug=makespan cores=4 \  
edu.rit.pj2.example.MinVerCovSmp g31.txt
Cover = 0 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 18 19 20 22 23 24 25 26 27 28
Size = 26
Job 17 makespan 111606 msec
```

Figure 11.1 plots the program’s running times 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 $K$:

$$T = 2.61 \times 10^{-8} N + (25.8 + 1.10 \times 10^{-7} N) / K.$$  \hspace{1cm} (11.1)

For a given problem size, the first term in the formula represents the program’s sequential portion, and the second term represents the parallelizable portion. The sequential fraction is quite large—17 to 19 percent for the runs I measured. This large sequential fraction causes the speedups and efficiencies to droop drastically, as is apparent in Figure 11.1.

This sequential fraction comes from the way the program uses vertex set objects. The first statement in the parallel loop body (line 60) constructs a new instance of class BitSet64 and assigns its reference to the candidate local variable. The first term in formula (11.1) 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. Continuously constructing new objects takes time. For a graph with 31 vertices, $2^{31}$, or over two billion, vertex set objects 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
Chapter 11. Exhaustive Search

Listing 11.1. RandomGraph.java (part 2)
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 large sequential fraction.) 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:

```java
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, once 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 constructing a new object. The bitmap() method replaces the candidate variable’s elements with those of the loop index, elems. Coding the loop this way eliminates 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 four 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 program 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 \
g31.txt
Cover = 0 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 18 19 20 22 23
Size = 26
Job 78 makespan 52265 msec
$ java pj2 debug=makespan cores=4 \
edu.rit.pj2.example.MinVerCovSmp2 g31.txt
Cover = 0 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 18 19 20 22 23
24 25 26 27 28
```
The new version’s running times are significantly—about 81 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:

\[ T = 3.13 \times 10^{-10} N + 2.41 \times 10^{-8} N/K. \]  

Now the sequential fraction is only 1.3 percent for the problem sizes I studied. The new version’s plots (Figure 11.2) show that the efficiencies degrade...
hardly at all as the number of cores increases.

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, re-use 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 re-
sulting 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 \text{contains}(7) method called on the set \{1, 3, 5, 7\}:

\[
\begin{array}{c}
\text{bitmap} \\
00000000000000000000000101010101 \\
00000000000000000000000000000001 \\
00000000000000000000000000000000 \\
00000000000000000000000000000000 \\
\end{array}
\]

\[
\begin{array}{c}
1 \\
1 \ll 7 \\
1 \ll 7 \\
1 \ll 7 \\
\end{array}
\]

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 \text{contains}(9) method called on the set \{1, 3, 5, 7\}:

\[
\begin{array}{c}
\text{bitmap} \\
00000000000000000000000101010101 \\
00000000000000000000000000000001 \\
00000000000000000000000000000000 \\
00000000000000000000000000000000 \\
\end{array}
\]

\[
\begin{array}{c}
1 \\
1 \ll 9 \\
1 \ll 9 \\
1 \ll 9 \\
\end{array}
\]

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

Class \text{BitSet32}'s \text{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 \text{BitSet32}'s \text{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 \text{BitSet64} is the same as class \text{BitSet32}, except the bitmap uses a long integer (type \text{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!
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.
• 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.