Chapter 15
Tuple Space

- Part I. Preliminaries
- Part II. Tightly Coupled Multicore
- Part III. Loosely Coupled Cluster
  - Chapter 13. Massively Parallel
  - Chapter 14. Hybrid Parallel
  - Chapter 15. Tuple Space
  - Chapter 16. Cluster Parallel Loops
  - Chapter 17. Cluster Parallel Reduction
  - Chapter 18. Cluster Load Balancing
  - Chapter 19. File Output on a Cluster
  - Chapter 20. Interacting Tasks
  - Chapter 21. Cluster Heuristic Search
- Part IV. GPU Acceleration
- Part V. Big Data
The massively parallel and hybrid parallel bitcoin mining programs in Chapters 13 and 14 were uncoupled parallel programs; there was no communication between the tasks. But in the next chapter we will start to encounter coupled parallel programs where the tasks do need to communicate with each other. So first, we have to look at how inter-task communication works in Parallel Java 2 programs.

A job maintains a conceptual repository of information called tuple space. David Gelernter originated the tuple space concept in a 1985 paper.* Tuple space holds tuples. Each tuple is an object whose fields contain appropriate information. The job’s tasks can put tuples into tuple space and take tuples out of tuple space; any task can put or take any tuple. Tasks communicate with each other via tuple space. When multiple jobs run on a cluster, each job has its own separate tuple space; it is not possible to communicate between different jobs, only between tasks in the same job.

I liken tuple space to the ocean. Fish (tasks) inhabit the ocean (tuple space), and the water of the ocean surrounds and permeates all the fish. Whatever one fish puts into the water (tuples), any fish can take out of the water.

Recall that a job defines a number of rules. Each rule in turn defines one or more tasks. The rules in a job, the tasks in a rule, and the tuples taken and put by a task are all intimately interrelated.

Figure 15.1 is a timeline of an example job illustrating the three possible kinds of rules. Time increases from left to right as the job executes. This particular job consists of four rules.

The first and second rules are start rules. A job can have any number of start rules. The start rules are coded like this in the job’s main() method:

```java
rule().task(TaskA.class);
rule().task(TaskB.class);
```

The first rule says, “At the start of the job, run an instance of task class

---

Chapter 15. Tuple Space

TaskA.” The second rule says, “At the start of the job, run an instance of task class TaskB.” Other attributes of the task, such as the task’s command line arguments, can also be specified as part of the rule.

Each rule fires at some point during the job’s execution. A start rule fires when the job commences executing. Thus, an instance of Task A and an instance of Task B start at time zero in Figure 15.1. (Keep in mind that the job’s main() method only defines the rules. The job actually begins executing once the job’s main() method returns.)

Task A and Task B each put a tuple into tuple space. A tuple is an object that is an instance of a subclass of class edu.rit.pj2.Tuple. In Figure 15.1, Task A and Task B each create an instance of class Type1Tuple. A tuple object carries content, which is stored in the tuple’s fields as defined in the tuple subclass. Each task calls the putTuple() method to put its tuple into tuple space.

The third rule in this example job is an on-demand rule. A job can have any number of on-demand rules. The on-demand rule is coded like this in the job’s main() method:

```
rule().whenMatch(new Type1Tuple()).task(TaskC.class);
```

This rule says, “Whenever a tuple appears in tuple space that matches the
given Type1Tuple, run an instance of task class TaskC.” The whenMatch() method’s argument is a template tuple. The rule fires when there is a match between the given template and a target tuple in tuple space. By default, a target matches a template if the target is an instance of the same class as the template, or if the target is an instance of a subclass of the template’s class. (The tuple matching criterion can be changed; we will see examples of this later.)

An on-demand rule can fire not at all, once, or more than once during the course of a job. At a certain point in Figure 15.1, Task B puts a Type 1 tuple into tuple space. This tuple matches the third rule’s template, so at this point an instance of Task C starts. Later, Task A puts another Type 1 tuple into tuple space. This tuple also matches the third rule’s template, so at this point another instance of Task C starts. The two Task C instances run independently.

When an on-demand rule fires, the target tuple that triggered the rule is automatically taken out of tuple space and is provided to the task as the matching tuple. The task can retrieve its matching tuple by calling the get-MatchingTuple() method. (A task triggered by a start rule, like Task A or Task B, has no matching tuple.) Information, recorded in the fields of the tuple, flows from one task to another task via tuple space; this is how different parts of the cluster parallel program communicate with each other.

An on-demand rule can specify more than one template, by including more than one whenMatch() clause in the rule definition. In this case the rule fires whenever every template has a matching target in tuple space. Each template must match a different target. If there are matches for some but not all of the templates, the rule does not fire. (The criterion for matching multiple templates can be changed.) When the rule fires, all the matching target tuples are automatically taken out of tuple space and are provided to the task as its multiple matching tuples.

In Figure 15.1, each instance of Task C puts a Type 2 tuple into tuple space. But because there are no rules that match a Type 2 tuple, no tasks are triggered when the Type 2 tuples appear. The tuples simply sit in tuple space.

The fourth and final rule in this example job is a finish rule. A job can have any number of finish rules. The finish rule was coded like this in the job’s main() method:

\[
\text{rule().atFinish().task(TaskD.class);}\]

This rule says, “When all other tasks have finished executing, run an instance of task class TaskD.” In other words, a finish rule triggers a task to run at the end of the job. If a job has more than one finish rule, then more than one finish task is started at the end of the job. In Figure 15.1, Task D starts as soon as the last Task C finishes.

An on-demand task automatically obtains matching tuples when the task
triggers. But that’s not the only way a task can obtain tuples. A task can also explicitly take a tuple out of tuple space by calling the `takeTuple()` method. The `takeTuple()` method’s argument is a template tuple. The `takeTuple()` method waits until a target tuple exists in tuple space that matches the template tuple. Then the `takeTuple()` method removes that target tuple from tuple space and returns the target tuple. This is what Task D does; it repeatedly takes a Type 2 tuple.

Figure 15.2 shows another example of a job. This job has two start rules and one finish rule. Task A and Task B start when the job commences. Each task computes a series of intermediate results. Task B needs Task A’s intermediate results, and Task A needs Task B’s intermediate results. Task A executes code like this:

```java
type2Tuple in;
type2Tuple template = new Type2Tuple();
for (...) {
    // Calculate intermediate result
    putTuple (new Type1Tuple (...));
    in = takeTuple (template);
    ...
}
```

After calculating an intermediate result, Task A packages that up in a Type 1
tuple and puts the tuple into tuple space. Task A then takes a tuple that matches a certain template. The template is a Type 2 tuple. The takeTuple() method blocks until a Type 2 tuple is present in tuple space. When Task B puts such a tuple, the takeTuple() method unblocks, the matching tuple is removed from tuple space, and the takeTuple() method returns the tuple that was removed (in). Task B executes similar code, except the tuple types are reversed:

```java
Type1Tuple in;
Type1Tuple template = new Type1Tuple();
for (...)
{
    ... // Calculate intermediate result
    putTuple (new Type2Tuple (...));
in = takeTuple (template);
...
}
```

Eventually, Task A and Task B complete their calculations. Each task puts its final results into tuple space as a Type 3 tuple. The finish rule then fires, Task C runs, Task C takes the Type 3 tuples out of tuple space, and Task C reports the final results.

The following methods in class edu.rit.pj2.Task allow a task to interact with tuple space. For further information, refer to the Parallel Java 2 documentation.

- **getMatchingTuple()** lets an on-demand rule’s task retrieve its matching tuples, which were automatically taken out of tuple space when the rule fired.
- **putTuple()** lets a task put a tuple—or, optionally, multiple copies of a tuple—into tuple space.
- **takeTuple()** lets a task take a tuple out of tuple space. A target tuple in tuple space that matches a given template tuple is removed and returned. If there are no matching tuples, takeTuple() blocks until a matching tuple shows up. If there is more than one matching tuple, takeTuple() moves and returns one of them; the matching tuple is chosen in an unspecified manner.
- **tryToTakeTuple()** does a **conditional** take. This is the same as a regular take, except if there are no matching tuples, tryToTakeTuple() does not block and immediately returns null.
- **readTuple()** lets a task make a copy of a tuple from tuple space. A target tuple in tuple space that matches a given template tuple is copied and returned, while the original target tuple remains in tuple space. If there are no matching tuples, readTuple() blocks until a matching tuple shows up. If there is more than one matching tuple, readTuple() copies and returns one of them; the matching tuple is chosen in an unspecified manner.
manner.

- `tryToReadTuple()` does a *conditional* read. This is the same as a regular read, except if there are no matching tuples, `tryToReadTuple()` does not block and immediately returns null.

As mentioned previously, by default, a target tuple matches a template tuple if the target is an instance of the same class as the template, or if the target is an instance of a subclass of the template’s class. The matching algorithm for a particular kind of tuple can be changed by overriding the `matchClass()` and `matchContent()` methods in the tuple subclass. The `matchClass()` method lets you specify which type or types of target tuple match a template tuple. The `matchContent()` method lets you compare the content (fields) of the template and the target to decide whether there is a match.

As you’ve probably gathered, tuple space is very flexible. Coupled with the rule-driven parallel task execution, the possibilities are endless for organizing and coordinating the pieces of a cluster parallel program. In the next few chapters we’ll see several example programs that illustrate the various kinds of rules and the various patterns for inter-task communication using tuple space.

To use tuples in a cluster parallel program, you have to define one or more *tuple subclasses*. A tuple subclass must fulfill these requirements:

- The tuple subclass must extend the base class `edu.rit.pj2.Tuple`.
- The tuple subclass must define fields to carry the tuple’s content. The fields can be anything at all—primitive types, objects, arrays, and so on.
- The tuple subclass must define a no-argument constructor.
- The tuple subclass must implement the `writeOut()` and `readIn()` methods, which are declared in interface `edu.rit.io.Streamable`, which is implemented by class `edu.rit.pj2.Tuple`. We’ll see examples of these methods below.

A tuple subclass can have other constructors and methods in addition to the ones above.

Once you’ve defined a tuple subclass, you can create tuples—instances of the tuple subclass—as you would any Java objects. However, the Parallel Java 2 middleware assumes that a tuple is immutable; that is, the tuple’s state will not change once it has been initialized. If you need to change a tuple’s contents, create a *deep copy* of the original tuple and change the copy.

To illustrate how to write a tuple subclass, let’s use the bitcoin mining program from Chapter 13. Suppose that a bitcoin mining task, instead of printing its results, wants to package its results into a tuple. Listing 15.1 shows what the tuple subclass would look like.

The tuple subclass, `MineCoinResult`, extends the `Tuple` base class (line 6). I defined four fields to hold the tuple’s content, namely the bitcoin mining
task’s results—the coin ID, nonce, digest, and running time in milliseconds (lines 9–12). I made the fields public so I could access them directly; making the fields private and adding public accessor methods is overkill for this simple tuple subclass. I defined the required no-argument constructor (lines 15–17) that leaves the fields at their default values. I also defined another constructor (lines 20–30) that initializes the fields to given values. A bitcoin mining task would use the latter constructor to package its results into a tuple. The constructor makes a deep copy of its arguments by cloning the byte arrays rather than assigning the byte array references. If the bitcoin mining task later changes the contents of the byte arrays passed in as the constructor arguments, the tuple’s byte arrays do not change (the tuple must be immutable).

Class MineCoinResult defines the two required methods, writeOut() and readIn() (lines 33–52). These methods are inherited from the superclass Tuple, which implements interface edu.rit.io.Streamable, which declares those methods. A class that implements interface Streamable defines a streamable object. A streamable object can write its contents to an out stream (class edu.rit.io.OutStream). Class OutStream has methods for writing values of primitive types (int, long, and so on), strings, arrays, and objects. Class MineCoinResult’s writeOut() method calls a few of these methods to write its fields. Each of these methods converts its argument to a sequence of bytes and writes the bytes to the given out stream. Likewise, a streamable object can read its contents from an in stream (class edu.rit.io.InStream). Class InStream has methods for reading values of all kinds. Class MineCoinResult’s readIn() method calls a few of these methods to read its fields. Each of these methods reads a sequence of bytes from the given in stream, converts the bytes to a value of the proper kind, and returns the value. The fields must be read back in exactly the same order as they were written out.

Tuples have to be streamable objects because tuples are sent from one process (node) to another process when the parallel program runs on a cluster. But objects, such as tuples, cannot be sent directly from one process to another. Byte streams, though, can be sent between processes. So a process needing to send an object to another process must convert the object to a sequence of bytes and send the byte stream. The process at the other end must then receive the byte stream and convert it back to an object. The Parallel Java 2 Library does most of the work under the hood. You, the programmer, just have to supply the code to write and read a streamable object’s fields, namely the writeOut() and readIn() methods.

What I’ve just described sounds suspiciously like Java’s built-in object serialization capability in package java.io. You’re correct; the Parallel Java 2 Library has its own object serialization capability. Why not use java.io? Mainly because Java Object Serialization produces byte sequences that are far longer than they need to be. As we will discover as we study cluster paral-
import edu.rit.io.InStream;
import edu.rit.io.OutStream;
import edu.rit.pj2.Tuple;
import java.io.IOException;
public class MineCoinResult
extends Tuple
{
    // Content fields.
    public byte[] coinID;
    public long nonce;
    public byte[] digest;
    public long msec;

    // Construct a new uninitialized mine coin result tuple.
    public MineCoinResult()
    {
    }

    // Construct a new mine coin result tuple with the given content.
    public MineCoinResult
    (byte[] coinID,
     long nonce,
     byte[] digest,
     long msec)
    {
        this.coinID = (byte[]) coinID.clone();
        this.nonce = nonce;
        this.digest = (byte[]) digest.clone();
        this.msec = msec;
    }

    // Write this tuple to the given out stream.
    public void writeOut
    (OutStream out)
    throws IOException
    {
        out.writeByteArray (coinID);
        out.writeLong (nonce);
        out.writeByteArray (digest);
        out.writeLong (msec);
    }

    // Read this tuple from the given in stream.
    public void readIn
    (InStream in)
    throws IOException
    {
        coinID = in.readByteArray();
        nonce = in.readLong();
        digest = in.readByteArray();
        msec = in.readLong();
    }
}

Listing 15.1. Tuple subclass MineCoinResult
lel programming, minimizing the number of bytes sent from process to process is extremely important. I designed Parallel Java 2’s streamable object capability to generate as few bytes as possible.

I’m not going to write a cluster parallel bitcoin mining program that uses tuples to report its results. (You can do that yourself if you like.)

**Under the Hood**

Tuple space is implemented as follows. The tuples are stored inside the Job object running in the job process on the cluster’s frontend node. The job maintains a list of all the tuples that have been put into tuple space but have not yet been taken out. The job also maintains a list of pending take and read requests whose templates have not yet matched any tuples.

When a task calls `putTuple()`, the task process sends a message containing the tuple to the job process, which adds the tuple to its list. The task streams the tuple out as it sends the message, and the job streams the tuple in as it receives the message. This is why every tuple subclass needs to be streamable. When streaming in a tuple, the in stream needs to construct an instance of the tuple subclass in which to store the incoming content. This is why every tuple subclass needs to have a no-argument constructor.

When a task calls `takeTuple()`, the task sends a message containing the template to the job, which adds the template to the list of pending requests. The `takeTuple()` method then blocks waiting to receive a response. Whenever the job receives a take-request message or a put-tuple message, the job checks its list of pending requests against its list of tuples. If the job finds a match between a template and a tuple, the job removes the matching tuple from its list, and the job sends a response message with the tuple back to the task that originally made the request. The `takeTuple()` method in the task then unblocks and returns the tuple. The `tryToTakeTuple()` method works the same way, except if there is no tuple that matches the template, the job immediately sends back a response to that effect. The `readTuple()` and `tryToReadTuple()` methods work the same way, except the matching tuple is not removed from the job’s list of tuples.

This design, with the job process acting as a central server of tuples, has a couple of consequences. First, the job process has to have enough memory to hold all the tuples that could ever be in tuple space at the same time. If the Java Virtual Machine’s default heap size is not large enough, you have to specify a larger heap size when you run the “java pj2” command (refer to the java command documentation). Second, all inter-task communication via tuple space has to go through the central job process; the task processes do not communicate with each other directly. Thus, each put or take of a tuple incurs a certain amount of network communication overhead to transfer the tuple from the sending task to the job to the receiving task.
When I designed the cluster parallel programming features of the Parallel Java 2 Library, I decided to orient them primarily towards the loosely coupled and medium coupled region of the coupling spectrum (as described in Chapter 1). The tasks run for minutes or hours, not milliseconds; the tasks communicate with each other infrequently (relative to the speed of the CPU), not every microsecond; and the tasks communicate small to medium amounts of data, not enormous quantities of data. In such an environment, the time the job spends in network communication is small relative to the time spent in computation, so it’s okay if all tuples go through the central job process. Because the tuples do not carry enormous quantities of data, it’s okay to store them all in the job process’s memory.

You can write tightly coupled cluster parallel programs with the Parallel Java 2 Library—programs that do frequent inter-task communication, or that communicate large quantities of data, or both. Just realize that these are outside the design center. As I said before, if you need a very high performance tightly coupled cluster parallel program, a parallel programming library intended for that environment, such as MPI, is probably a better choice.

I also had cloud computing in mind when I designed Parallel Java 2’s cluster parallel programming features. When you run a Parallel Java 2 program on a “cloud cluster,” the frontend process runs on your machine, but the backend processes run on virtual machines in the cloud. It’s fairly easy to set up a connection from a virtual machine in the cloud to your machine (from a backend process to the frontend process). It’s not so easy to set up a connection from one virtual machine to another in the cloud. I therefore decided to put tuple space in the frontend process and have the backend processes talk to the frontend process to transfer tuples, rather than have the backend processes talk directly to each other. (The Parallel Java 2 Library does not support clusters in the cloud yet.)

### Points to Remember

- A Parallel Java 2 job consists of tasks specified by rules.
- When a rule fires, an instance of the rule’s task is created and executed.
- Each task normally runs in its own separate process on one of the cluster’s backend nodes.
- A start rule fires when the job commences execution.
- An on-demand rule fires whenever a target tuple appears in tuple space that matches a given template tuple. The target becomes the task’s matching tuple.
- A finish rule fires at the end of the job, once all other tasks have finished.
- A task can put tuples into tuple space.
- A task can also read tuples in tuple space and take tuples out of tuple space, with or without blocking, by specifying a template to match.
• Normally, a target matches a template if the target is an instance of the same class as the template, or a subclass thereof.
• The tuple matching criterion can be altered by overriding methods in the tuple subclass.
• Every tuple subclass must have a no-argument constructor and must be streamable.
• Every tuple must be immutable—its state must not change once it has been initialized. Make a deep copy of a tuple and change the copy if necessary.