A Tutorial for CC++

First Edition

Caltech CS-TR-94-02

Paul A.G. Sivilotti and Peter A. Carlin
Compositional Systems Research Group
Department of Computer Science
California Institute of Technology
Caltech Mail Stop 256-80
Pasadena, California 91125
cc++@cs.caltech.edu

Acknowledgements

This tutorial owes a great deal to the other members of the Compositional Systems Research Group at Caltech – Mani Chandy, Carl Kesselman, John Garnett, Svetlana Kryukova, Tal Lancaster, Berna Massingill, Adam Rifkin, Mei Su, and John Thornley. Their careful review and criticism helped shape this document into its current form.

This research was supported in part by NSERC. The research on CC++ object libraries for concurrent computation is funded by ARPA under grant N00014-91-J-4014. The research on compositional concurrent notations is funded by the NSF Center for Research on Parallel Computing under grant CCR-9120008.

Contents

List of Figures

Chapter 1

Introduction

1.1 The CC++ Programming Language

Is parallel programming difficult? Many programmers complain that architecture dependencies, confusing notations, difficult correctness verification, and burdensome overhead make parallel programming seem tedious and arduous. These barriers cast doubt on the practicality of parallel programming, despite its many potential benefits.

Compositional C++ (CC++) was designed to alleviate the frustrations of parallel programming by adding a few *simple* extensions to the sequential language C++. It is a strict superset of the C++ language so any valid C or C++ program is a valid CC++ program. Conversly, many classes of parallel CC++ programs can be simply rewritten as equivalent sequential programs. For these classes of programs, the developement path can be very similar to that of the equivalent sequential program. This compatibility with the C and C++ languages facilitates the transition to the task of parallel programming for users knowledgeable in those languages.

CC++ extends C++ with the following eight constructs:

`par` blocks enclose statements that are executed in parallel.

`parfor` denotes a loop whose iterations are executed in parallel.

`spawn` statements create a new thread of control executing in parallel with the spawning thread.

`sync` data items are used for synchronization.

`atomic` functions control the level of interleaving of actions composed in parallel.

processor objects define the distribution of a computation.

`global` pointers link distributed parts of the computation.

data transfer functions describes how information is transmitted between address spaces.

Despite the simplicity and the small number of extensions, the conjunction of these constructs, when combined with C++, results in an extremely rich and powerful parallel programming notation.

The richness of this language is reflected in its suitability to a wide spectrum of applications. In fact, CC++ integrates many seemingly disparate fields:

Sequential and parallel programming Parallel blocks are notationally similar to sequential blocks.

Shared and distributed memory models CC++ can be used on shared or distributed memory architectures as well as across networks which may be heterogenous.

Granularity CC++ can be used in computations involving a variety of granularities, ranging from fine-grain parallelism with frequent synchronization between small threads to coarse-grain parallelism with sparse synchronization between large, distributed processes.

Task and data parallelism Task and data parallel applications can be expressed in CC++, as well as programs that combine the two.

Synchronization techniques The synchronization mechanisms provided by CC++ are powerful enough to express any of the traditional imperative synchronization and communication paradigms.

All of the object-oriented features of C++ are preserved in CC++. These features (especially generic classes and inheritance mechanisms) promote the reuse of code, structure, and verification arguments. Thus, CC++ provides an excellent framework for the developement of libraries and for the use of software templates in program construction. This reuse is especially important and useful in the context of parallel programming.

1.2 This Tutorial

This document is divided into two basic parts. The first three chapters describe the single address space constructs in CC++ (`par`, `parfor`, `spawn`, `atomic`, and `sync`). The last four chapters describe the multiple address space constructs (processor objects, `global` pointers, and data transfer functions).

Each chapter loosely follows this basic outline:

- Introduces one or more constructs, giving motivation and background.
- Introduces semantics and syntax of the construct(s).
- Highlights possible difficulties ("Pitfalls") in using the construct(s).
- Provides examples of the construct(s) in use.

The source code of the examples located at the end of each chapter are at the same ftp site (`cs.caltech.edu` and directory `CC++/docs/tutorial`) as this tutorial. The names of the files mentioned in the tutorial match with the names of the files in that directory.

In order to learn CC++, and in order to read this tutorial, a basic understanding of C is assumed. An understanding of the C++ concept of an object is also assumed. This tutorial uses C and simple C++ syntax. Many programs in CC++, particularly distributed computations, use more detailed C++ features. A good reference for these, and C++ in general, is *The C++ Programming Language* by Bjarne Stroustrup, available from Addison-Wesley.

This tutorial is not a complete definition of the CC++ language. The complete language definition can be found in the `CC++/docs` directory at the anonymous ftp site `cs.caltech.edu`, or as Caltech Technical Report CS-TR-92-02.

This tutorial covers some major restrictions in the current CC++ implementation. The complete set of restrictions is outlined in the release notes, also in the `CC++/docs` directory.

Direct comments, questions and suggestions about CC++ or this tutorial to `cc++@cs.caltech.edu`. Report errors in the implementation to `cc++-bugs@cs.caltech.edu`.

Chapter 2

Creating Parallel Threads of Control

Fundamental to any parallel programming language is the mechanism by which parallel threads of control are created. In this chapter we introduce the three mechanisms used in CC++:

`par`: structured parallel block construct

`parfor`: structured parallel loop construct

`spawn`: unstructured parallelism

It is important to note that these constructs do not actually *distribute* work, or describe *where* work is to be performed. They simply describe *possible* concurrency in a program. If, for example, the program is being executed on one single-processor workstation, at most one thread can be executed at any point in time and hence no performance speed-up will be observed. In this case, the actions of the statements composed in parallel by the above constructs will execute in some arbitrarily interleaved fashion.

2.1 Structured Parallel Blocks: `par`

2.1.1 Introduction

The most basic mechanism for creating parallel threads of control in CC++ is the parallel block. A parallel block looks just like a compound statement in C or C++ with the keyword `par` in front of it:

```
par  {
  statement_1;
  statement_2;
  ...
  statement_N
}
```

Except for a few cases, the statements inside a parallel block can be any legal C, C++, or CC++ statement. The exceptions are variable declarations and statements that result in nonlocal changes in the flow of control. These limitations will be discussed in more detail in the "Pitfalls" section of this chapter (2.1.5).

A parallel block differs from the normal C block in that the order in which the statements in the block execute is not defined: an execution of a parallel block is an interleaved or possibly concurrent execution of the statements within the block. The execution of a parallel block is finished after *all* of the statements within the block have finished executing. Consequently, statements after a parallel block will not start to execute until all the statements within the parallel block terminate.

We do know some things about how the operations in a parallel block are mixed together. We know that the order of operations within any one statement in the parallel block is preserved. We also know that regardless of how long it may take any statement to finish, each statement in the block will eventually get a chance to execute. We call the type of execution that occurs in a parallel block a *fair* interleaving. There are some pragmatic issues having to do with implementing fair interleaving. If you have an application that depends on fairness, you should consult the Appendix for the particular hardware platforms you are using to get the details.

Let us look at some parallel blocks.

```
{
  int a, b, d;
  c = 1;
  par  {
    a = 2;
    b = c+3;
  }
  d = b+1;
}
```

The parallel block in this example has two independent threads of control: the first assigns the value 2 to the integer variable a and the second evaluates the sum c+3 and assigns this value to the integer variable b. The statement assigning the value to `d` does not execute until both assignments to `a` and `b` have completed. This flow of control is illustrated in Figure 2.1.

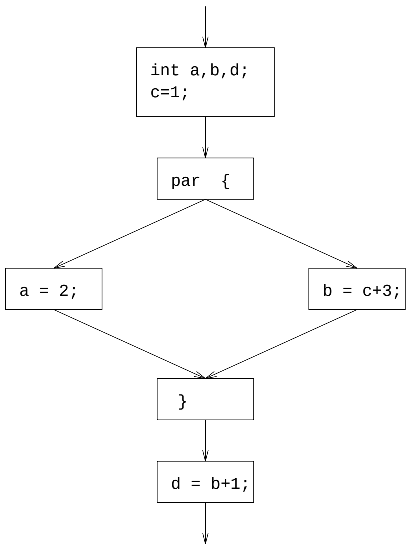

Figure 2.1: Flow of Control in a Simple Parallel Block

The statements contained in a parallel block can be function calls, such as:

```
par  {
  g = gcd(a,b);
  l = lcm(a,b);
  s = sum(a,b);
}
```

2.1.2 Structuring

The key feature of a parallel block is the structuring it provides to parallel code. This structuring is a result of the implicit barrier defined by a parallel block. As discussed above, a parallel block is defined to terminate only when all statements inside the block have terminated. Thus, the closing brace of a parallel block represents a barrier that all parallel threads of control within the block must reach, before the block is terminated.

To illustrate, consider:

```
{
  par  {
    a = f(2);
    b = f(31);
    c = g(2,6.5);
  }
  sum = a+b+c;
}
```

Schematically, the flow of control is represented in Figure 2.2. Notice that the individual threads that assign values to a, b, and c may terminate in any order, at various times. The block, however, is not terminated until all three have completed. Because the outer block is sequential, only when the parallel block terminates can the statement assigning a value to `sum` execute, as you would expect from knowing C.

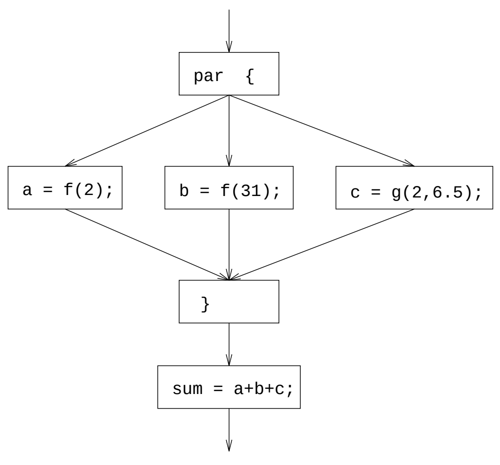

Figure 2.2: Flow of Control in a Parallel Block

Similarly, consider the following example:

```
{
  int min, max;
  par  {
    min = find_min(A);
    max = find_max(A);
  }
  for (i=min; i<=max; i++)
    compute(i);
}
```

Again, at the end of the parallel block, we know that both `findmin()` and `findmax()` functions have completed and their return values have been assigned to `min` and `max` respectively.

The same principal applies when a parallel block is part of an enclosing, perhaps iterating, structure. Consider the following example:

```
{
  int pow[N];
  pow[0] = 1;
  pow[1] = 2;
  for (int i=2; i<N-1; i=i+2)
    par  {
      pow[i]   = pow[i/2]*pow[i/2];
      pow[i+1] = pow[i/2]*pow[i/2]*2;
    }
}
```

This concept of an implicit barrier is an extremely powerful and expressive one. Many sequential C or C++ programs can be conveniently transformed into parallel programs with judicious use of the parallel block. The following example finds the minimum element in a global array. The technique used is that of divide and conquer, where the minimum is found as the smaller of the minimum of the first half and the minimum of the second half of the array.

```
int A[N];
int min_element (int i, int j)
{
  if (i==j) return A[i];
  else  {
    int small1, small2;
    int middle = (i+j)/2;
    par  {
      small1 = min_element(i,middle);
      small2 = min_element(middle+1,j);
    }
    if (small1<small2) return small1;
      else             return small2;
  }
}
```

Notice that if the keyword `par` is removed, the resulting program is a correct solution (albeit a sequential one) to the problem of finding the minimum element. The parallel block allows the independent branches of the recursion to proceed in parallel. The implicit barrier of the parallel block means that the values of `small1` and `small2` have been evaluated and assigned before the comparison between the two is made. The flow of control in this program is illustrated in Figure 2.3.

2.1.3 Sharing Data

Because the actions contained in different threads of control in a parallel block are executed in a parallel, or possibly arbitrarily interleaved, manner, it should be clear that sharing and modifying data between such threads can be dangerous. The CC++ language allows you to do dangerous things. However, you can stay out of trouble by following a simple rule. First a definition:

Definition A *mutable* variable is any non-constant (`const`) variable or structure whose value or contents can be modified.

The above definition may appear redundant. However, in Chapter 4 we will introduce a new type of variable that is non-constant, and yet whose contents cannot be modified.

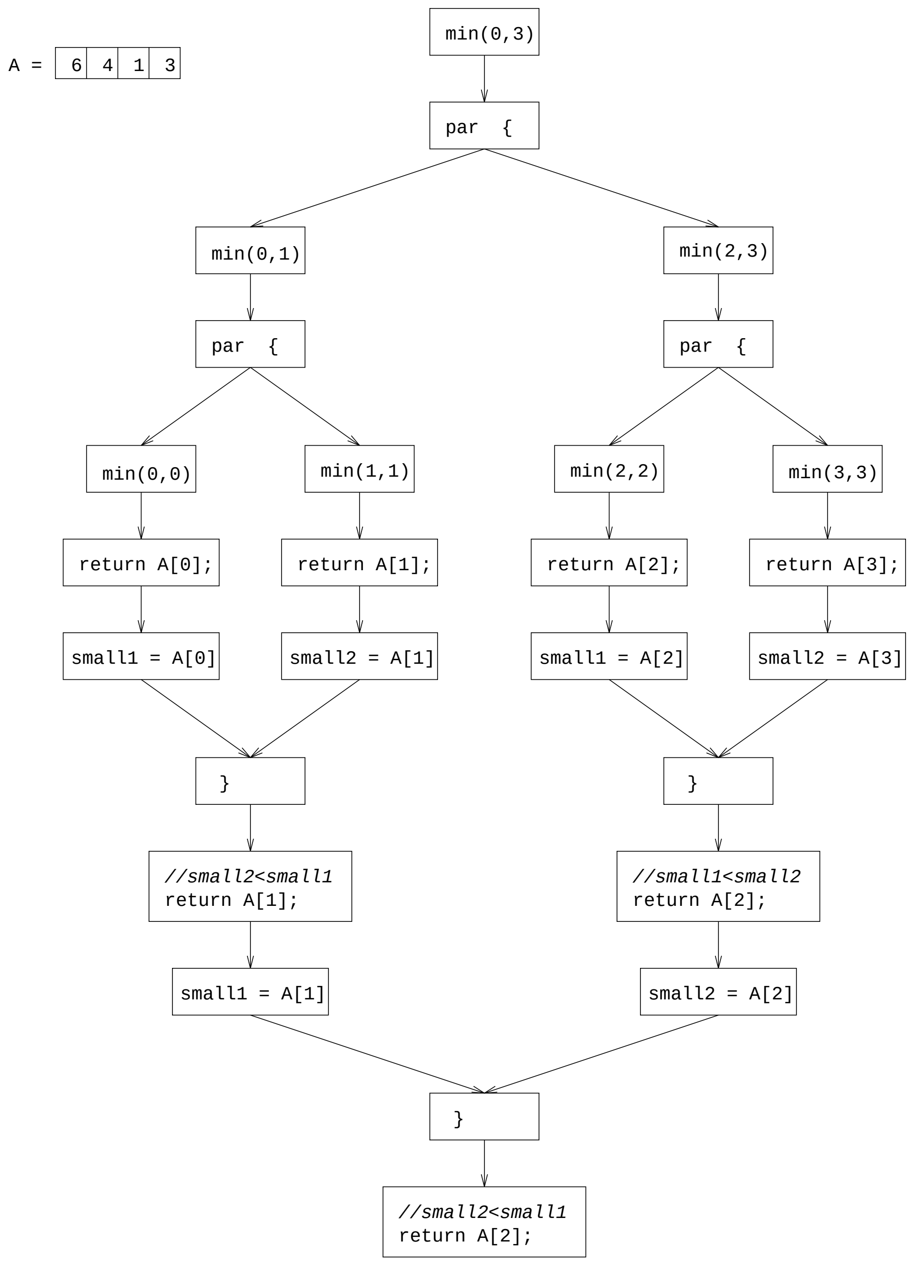

Figure 2.3: Concurrent Execution of Divide and Conquer Algorithm

Now for the rule:

> *If a mutable variable is modified in a thread of a parallel block, then no other thread of control in the parallel block should make use of that variable (either by writing or by reading the variable).*

The following example illustrates the point.

```
{
  int x;
  par  {
    x = 1;     //dangerous sharing
    x = 2;     //dangerous sharing
  }
}
```

This parallel block violates the sharing rule for mutable variables. The mutable integer variable `x` is modified in both statements of the parallel block. Though this code may not result in a compile-time error, or even necessarily a run-time error, it is extremely dangerous. This is because at the end of the parallel block we can say nothing about the value of x. It could be 1, or 2, or something completely different! If mutable variables are shared in this manner between parallel threads of control, it is almost certainly a programming error.

Not only should at most one thread of control modify the value of a mutable variable, but if a mutable variable is modified in one thread, then no other thread should access the value of that variable. For example:

```
{
  int n,s;
  par  {
    s = 3;      //dangerous sharing
    n = 2*s;    //dangerous sharing
  }
}
```

Here the first statement in the parallel block initializes the value of the mutable variable `s`, and the second statement uses the value of `s`. Again, this does not necessarily result in a run-time error, but it is almost certainly a programming error, since we can say nothing about the value of `n` (or even `s` for that matter) at the end of the parallel block. Notice that both examples above are correct sequential programs when the keyword `par` is removed. The programmer should therefore take care when parallelizing sequential code that no inadvertent sharing of mutable data is created by the creation of parallel blocks.

So the only safe kind of sharing of mutable data between parallel threads of control is multiple reading of the variable. Examples of this kind are given in Sections 2.1.1 and 2.1.2. Make sure you understand why these examples do not involve dangerous sharing of mutable variables and are indeed correct parallel programs. In Chapter 3 we introduce a mechanism for controlling access to shared mutables, and in Chapter 4 we introduce a new kind of variable that can be shared between parallel threads of control in a different manner.

2.1.4 Nesting

Parallel blocks can contain simple statements, sequential blocks, or even other parallel blocks. The behavior of such nesting is precisely what one would expect. The statements within the sequential block are executed sequentially with respect to each other, but are composed in parallel with the other threads of control of the parallel block. For example:

```
par  {
  {
    result1 = trial(params1);
    stats1 = generate_stats(result1);
  }
  {
    result2 = trial(params2);
    stats1 = generate_stats(result2);
  }
}
```

Here the function `trial()` is performed on the argument `params1` and the assignment to `result1` is completed *before* the function `generate_stats()` begins. Between the two threads of control, however, there is no ordering of actions. The statistics could be generated for the first trial before the second trial even begins, or vice versa, or some interleaving of the two could occur. But within each thread, the order of execution is strictly sequential.

Similarly, one of the statements of a parallel block could be another parallel block. Consider the following general example, where `si` represents a generic statement.

```
par {
  par {
    s1;
    s2;
    s3;
  }
  {
    s4;
    par {
      s5;
      s6;
    }
    s7;
  }
  s8;
}
```

The flow of control for this program is represented in Figure 2.4.

2.1.5 Pitfalls

In this section we describe several common errors to be careful to avoid.

1. The interleaving of actions composed in parallel is *arbitrary.* The language makes no guarantees about how often, or even how soon, instructions from a particular thread will be executed. For example:

   ```
   par {
     while (1) g();
     f();
   }
   ```

   The first statement in this case is an infinite loop. Instructions from this loop could be executed for a very, very long time before a single instruction from the second thread is chosen for execution. Thus, we cannot expect to observe function `f()` even begin execution.

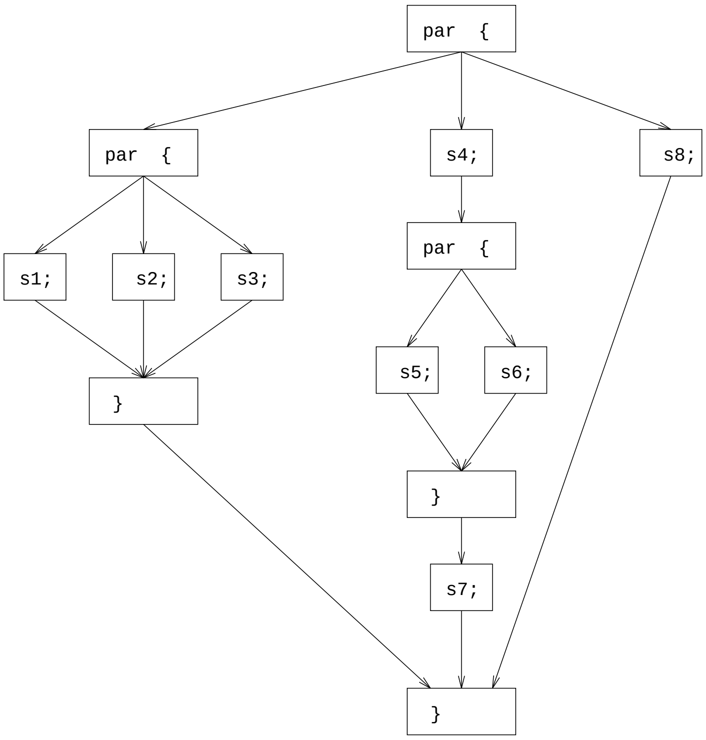

Figure 2.4: Nesting Blocks Within Parallel Blocks

2. Declarations are not permitted at the level of scope of a parallel block. This is consistent with the rules of variable sharing for parallel blocks. For example:

   ```
   par {
     int x; //ERROR
     x = 2;
   }
   ```

   Within nested levels of scope, however, declarations are permitted:

   ```
   par  {
     {
       int s;
       s = f();
       g(s);
     }
     for (int i=0; i<3; i++)
       k(i);
   }
   ```

3. Gotos into, out of, or between statements at the level of scope of a parallel block are not permitted. In particular, no `break`, `continue`, `goto`, or `return` statements are permitted.

In addition, the current implementation of the language places the following restrictions on CC++ programs.

1. No exceptions can be thrown inside parallel blocks.

2. A file or stream on which I/O is performed should be seen as a mutable variable. Thus, composing I/O operations on the same file in parallel is dangerous and should be avoided. We will see a mechanism in Chapter 3 that permits safe parallel composition of such operations. I/O operations on different files or streams can safely be composed in parallel with each other.

2.1.6 Examples

In this section we give some complete examples that can be compiled and executed. These examples illustrate the use of the parallel block.

Hello World This program displays the traditional greeting `"hello, world"`.

```
#include <iostream.h>

int main()
{
  char *s1, *s2;
  par  {
    s1 = "hello, ";
    s2 = "world\n";
  }
  cout << s1 << s2 << endl;
  return(0);
}
```

The assignment of `"hello, "` to `s1` and of `"world"` to `s2` can occur concurrently or in some arbitrarily interleaved manner. Perhaps the operations required for assignment to `s1` are executed, and then the operations required for the assignment to `s2` are executed. (Such a sequence of operation is identical to the execution of the sequential program created by removing the keyword `par`.) Perhaps `s2` is assigned to the string `"world"` first, and then the assignment of `s1` occurs. Perhaps the operations for these two assignments are interleaved in some manner or perhaps they occur in parallel. Regardless, because `s1` and `s2` are distinct mutable variables, these operations are guaranteed to be noninterfering. Hence, at termination of the parallel block, we know that `s1` is the string `"hello, "` and `s2` is the string `"world"`.

The program therefore results in the message "hello, world" being displayed every time, regardless of the actual order of operations.

Finding a Minimum Element This program finds the minimum element of a statically defined integer array. The value of this minimum element is displayed.

```
#include <iostream.h>

const int N = 8;
int A[N] = {3, 4, 5, 1, 2, 5, 9, 6};
```

```
int find_min (int i, int j)
{
  if (i==j)
    return A[i];
  else  {
    int small1, small2;
    par  {
      small1 = find_min (i, (j+i)/2);
      small2 = find_min ((j+i)/2+1, j);
    }
    if (small1<small2)  return small1;
      else              return small2;
  }
}

int main()
{
  int min = find_min(0,N-1);
  cout << "Minimum element is " << min << endl;
  return 0;
}
```

As explained in Section 2.1.2, the minimum is found recursively as the smaller of the minimum of the first half of the array and the minimum of the second half of the array. Because the recursive calls operate on different parts of the array, they are completely independent and can be composed in parallel.

2.2 Structured Parallel Loops: `parfor`

2.2.1 Introduction

The construct for parallel composition of a variable number of statements is `parfor`. With the exception of the keyword, the syntax of a `parfor` statement is the same as the usual C++ `for` statement:

```
parfor (int i=0; i<N; i++)  {
  statement_1;
  statement_2;
  ...
  statement_N
}
```

This is a parallel loop construct in which the iterations are executed in parallel with each other. As with the usual C or C++ for loop, the body of each iteration is executed sequentially. Similar to the parallel block discussed in Section 2.1, there is an implicit barrier at the end of a `parfor`. The `parfor` statement completes only when all the iterations have completed.

As a simple example, consider:

```
{
  int A[N];
  parfor (int i=0; i<N; i++)
    A[i] = i;
}
```

Here there are `N` parallel threads of control. Each element of the array is assigned (by a different thread of control) to the value of its index. The `parfor` statement terminates only when all elements of the array have been assigned their values. The flow of control for this example is illustrated in Figure 2.5.

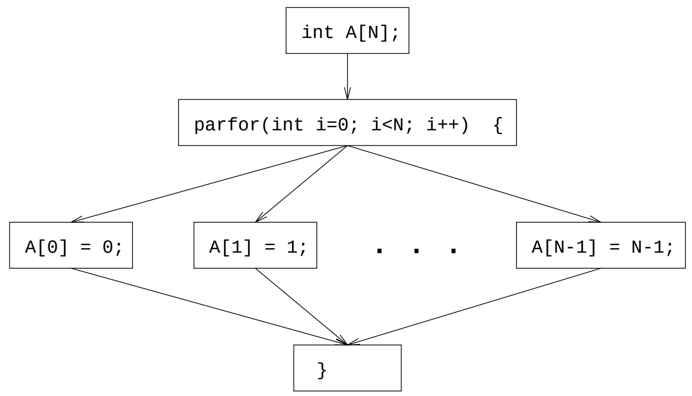

Figure 2.5: Flow of Control in a Basic Parallel Loop

2.2.2 Sharing Data

The rule concerning sharing data that applies to parallel blocks applies to `parfor` statements as well. The parallel threads of control in a `parfor` statement (that is, the individual iterations of the loop) should not share mutable data. For example:

```
{
  int sum = 0;
  parfor (int i=0; i<N; i++)
    sum += A[i]; //dangerous sharing
}
```

In this example, the mutable variable `sum` is modified in all `N` iterations. Recall from Section 2.1.3 that such sharing is dangerous.

The loop control variable used in a `parfor` statement is a special case. This variable must be declared in the `parfor` statement itself, as is seen in the preceding two examples. Each iteration is then considered to have its own `const` copy of this loop control variable. The loop control variable *cannot* be modified within the body of a `parfor`.

2.2.3 Loop Unraveling

The conversion of the loop control variable to a constant value within each iteration permits unraveling of the loop without executing the body of any iteration. After the initialization of the loop control variable and the test of the loop condition, execution of the body of the first iteration can begin, but so can the increment of the loop control variable, followed by the test of the loop condition. The flow of control for a generic `parfor` statement is represented in Figure 2.6.

Notice that, as in a parallel block, nothing can be said about the order of execution of the individual iterations. There is no guarantee that the first iteration will even *begin* before any other iteration.

The semantics of loop unraveling are therefore defined in terms of a *sequential* repetition of condition test evaluation then loop control variable increment. Though the generality of C and C++ permit these operations to be arbitrarily complicated and hence require sequential evaluation, a particular implementation of CC++ may do better in certain instances. For example, it is not difficult to envision a compiler that detects the common loop format

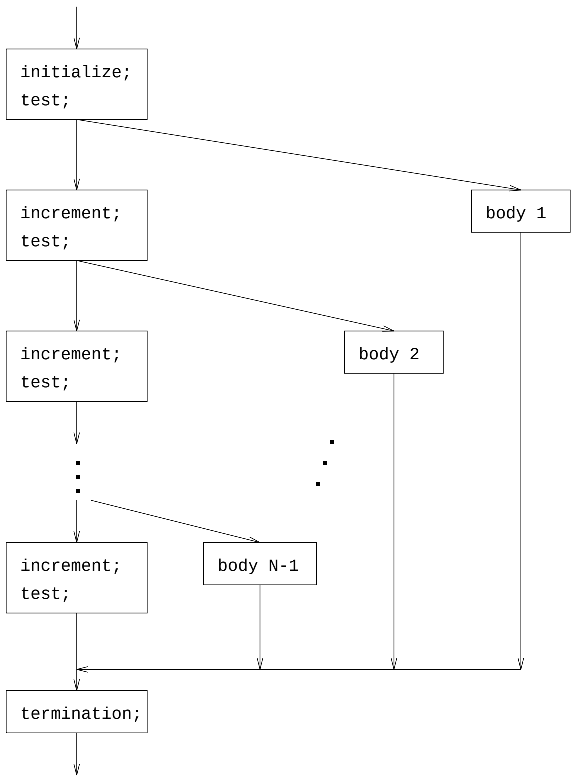

Figure 2.6: Flow of Control in a Parallel Loop

```
parfor (int i=0; i<N; i++)
```

and is able to flatten the creation of the N parallel threads of control. Thus, the linear complexity of the semantic definition of `parfor` does not necessarily imply a linear component in the performance of `parfor` in all cases.

2.2.4 Pitfalls

1. The loop control variable must be declared in the `parfor` statement. Variables that come from a higher scope cannot be used as loop control variables in a parallel loop. The following example is a compile-time error:

   ```
   {
     int i;
     parfor (i=0; i<N; i++)  //ERROR
       { ... }
   }
   ```

2. The (sequential) body of a `parfor` represents a nested level of scoping within the parallel composition. Declarations are therefore permitted at this level.

   ```
   parfor (int i=0; i<N; i++)  {
     int x, y = i*2;
     x = f();
     g(x,y);
   }
   ```

3. Nesting sequential loops within parallel loops must be done with care. Consider the following sequential code:

   ```
   {
     int i,j;
     for (i=0; i<N; i++)
       for (j=0; j<N; j++)
         { ... }
   }
   ```

The following parallelization of this code is incorrect:

```
{
  int j;
  parfor (int i=0; i<N; i++)
    for (j=0; j<N; j++)          //Error: j is a shared mutable
      { ...  }
}
```

This code is incorrect because the mutable variable `j` is shared between the concurrently executing iterations of the parallel loop. All sequential loops nested within a parallel loop should declare their loop control variables.

4. In general, the creation and deletion of parallel threads of control can be a relatively expensive operation. If the amount of computation performed by each iteration is small, such a program could exhibit significant performance degradation. This cost puts a practical limitation on the number of iterations composed in parallel by a `parfor` statement.

5. No gotos into, out of, or between iterations of a `parfor` are permitted. No `break`, `continue`, `goto`, or `return` statements are permitted at the `parfor` level of scope.

The current implementation restrictions are the same as those for parallel blocks (see Section 2.1.5).

2.2.5 Examples

In this section we give some complete examples that can be compiled and executed. These examples illustrate the use of the `parfor` statement.

Array Initialization This program initializes the entries of a float-valued array.

```
#include <iostream.h>
const int N = 10;
```

```
float f (float i)
{
  //evaluate polynomial 3xxx-7xx+2x+4 at x=i
  return 3*i*i*i - 7*i*i + 2*i + 4;
}

int main()
{
  //makes A[i] = f(i) and then outputs A[i]
  float A[N];

  parfor (int i=0; i<N; i++)
    A[i] = f((float)i);
  //implicit barrier

  for (int j=0; j<N; j++)
    cout << "A[" << j << "] is " << A[j] << endl;
  return 0;
}
```

Each element of the array `A[]` is evaluated and assigned a value by a different thread of control. This is an instance of safe sharing of mutable variables between concurrent threads of execution because each individual element of the array `A[]` is a different mutable variable, and single elements are not shared between iterations of the `parfor` loop.

Scientific Mesh Computation This is a solution to the cellular automaton grid computation problem. A gradient function is evaluated iteratively over a two dimensional grid of points. The initial boundary conditions are given and each interior point computes its new value as a weighted average of its old value and its neighbors' old values. This process is repeated until convergence (or, in this case, until a fixed number of iterations have been processed).

```
#include <iostream.h>
const int N = 10;

float Mesh[N][N];

float compute_cell (int r, int c)
{
  return (4*Mesh[r][c] + Mesh[r+1][c] + Mesh[r-1][c]
          + Mesh[r][c+1] + Mesh[r][c-1])/8.0;
}
```

```
void calculate (void)
{
  float New_Mesh[N][N];
  for (int iterate=0; iterate<300; iterate++)  {
    parfor (int row=1; row<N-1; row++)              //compute new mesh
      parfor (int col=1; col<N-1; col++)
        New_Mesh[row][col] = compute_cell(row,col);
    parfor (int newrow=1; newrow<N-1; newrow++)     //update old mesh
      parfor (int newcol=1; newcol<N-1; newcol++)
        Mesh[newrow][newcol] = New_Mesh[newrow][newcol];
  }
}

int main()
{
  for (int i=1; i<N-1; i++)  {        //initialize boundary of Mesh
    Mesh[0][i] = i;
    Mesh[i][0] = i;
    Mesh[N-1][i] = N-1+i;
    Mesh[i][N-1] = N-1+i;
  }
  for (i=1; i<N-1; i++)               //initialize interior of Mesh
    for (int j=1; j<N-1; j++)
      Mesh[i][j] = 0;

  calculate();

  for (i=0; i<N; i++)  {              //display outcome
    for (int j=0; j<N; j++)
      cout << Mesh[i][j] << "\t";
    cout << endl;
  }
  return 0;
}
```

The computation is done in parallel for all $\mathtt{N}^2$ points. Note that a particular point's old value may be used in as many as 4 computations (i.e. for each of its neighbors). Because each computation requiring this old value performs only a read operation, this is not an instance of dangerous sharing. The value that is computed is written to a new mesh. At the end of the first two `parfor` statements, therefore, we know that this new mesh of values has been completely filled in with the new values. The second group of `parfor` statements can then safely copy these values to the original mesh. It is important to understand why no mutable variable is being both written and read by parallel threads of control.

Clearly this is not a very efficient solution to this problem. The cost of copying the mesh of values at every iteration is high. One way to avoid this cost is to maintain two arrays, M1 and M2. On even iterations, the M1 stores the old value and the new values are written into M2, and vice versa on the odd iterations. Also, the number of parallel threads of control is excessive (as discussed in the "Pitfalls" section, 2.2.4), considering the small amount of work to be performed by each one. It is reasonable to expect a program in which each thread of control computes the values for a collection of cells to be more efficient. The following code incorporates this optimization.

```
#include <iostream.h>

//N...........size of grid (N by N)
//T...........number of concurrent processes, each working
//                  on an N by ((N-2)/T+2) slice  (T must divide N-2)
//                  and (N-2)/T >= 2
//HORIZON.....the event horizon for terminating iteration

#define N  22
#define T  5
#define HORIZON 300

float Grid[2][T][N][(N-2)/T+2];

void initialize (void)
{
  int i,j,k;
  for (i=0; i<T; i++)                //initialize interior of Grids
    for (j=1; j<N-1; j++)
      for (k=0; k<(N-2)/T-1; k++)
        Grid[0][i][j][k] = 0;
  for (i=0; i<T; i++)                  //initialize boundary of Grids
    for (k=0; k<(N-2)/T+2; k++)  {
      Grid[0][i][0][k] = i*(N-2)/T+k;
      Grid[0][i][N-1][k] = i*(N-2)/T+k+N-1+N-1;
    }
  for (i=0; i<N; i++)  {
    Grid[0][0][i][0] = i;
    Grid[0][T-1][i][(N-2)/T+1] = i+N-1;
  }
}
```

```
float compute (int l, int s, int r, int c)
{
  return (4*Grid[l][s][r][c] + Grid[l][s][r+1][c] + Grid[l][s][r-1][c]
          + Grid[l][s][r][c+1] + Grid[l][s][r][c-1])/8.0;
}

void exchange_boundaries (int l, int s)
{
  int i;
  if (s<T-1)
    for (i=0; i<N; i++)
      Grid[l][s+1][i][0] = Grid[l][s][i][(N-2)/T];
  if (s>0)
    for (i=0; i<N; i++)
      Grid[l][s-1][i][(N-2)/T+1] = Grid[l][s][i][1];
}

int main()
{
  initialize();

  for (int iterate=0; iterate<HORIZON; iterate++)  {
    parfor (int slice=0; slice<T; slice++)  {       //for each slice
      for (int row=1; row<N-1; row++)                  //compute new Grid
        for (int col=1; col<(N-2)/T+1; col++)
          Grid[(iterate+1)%2][slice][row][col] = compute(iterate%2,slice,row,col);
    }
    parfor (int slice2=0; slice2<T; slice2++)  {    //exchange boundaries
      exchange_boundaries((iterate+1)%2,slice2);       //between neighbours
    }
    cout << "exchange " << iterate << endl;
  }

  for (int i=0; i<T; i++)  {            //display outcome
    cout << "Slice " << i << endl;
    cout << "=========" << endl;
    for (int j=0; j<N; j++)  {
      for (int k=0; k<(N-2)/T+2; k++)
        cout << Grid[HORIZON%2][i][j][k] << "\t";
      cout << endl;
    }
    cout << endl;
  }

  return 0;
}
```

Also, the synchronization at the end of each iteration is excessive. The value of a particular cell in the mesh can affect the next values of only its neighbors. Thus, there is no need for a cell to synchronize with any cells apart from its immediate neighbors. We will discuss how such synchronization schemes can be constructed in Chapter 4.

2.3 Unstructured Parallelism: `spawn`

2.3.1 Introduction

A final construct for creating parallel threads of execution is `spawn`. Parallel blocks and `parfor` statements have the nice property that a block terminates only when all their components terminate. They are the parallel equivalent of structured control flow statements in C and C++. The `spawn` statement is used to create a completely independent thread of control that executes in a concurrent (or possibly a fairly interleaved) manner with the thread that executes the `spawn`. Unlike the structured parallel statements, no parent-child relationship exists between the spawned thread and the spawning thread. There is no barrier or any form of implicit synchronization between the two, either at their beginning or at their termination.

Only functions can be spawned. A spawned function cannot return a value. Thus, `spawn` is similar in functionality to the thread creation facilities provided in many thread libraries.

The syntax for this statement is:

```
spawn f();
```

This unstructured parallelism is analogous to unstructured sequential code, with jumps and breaks in execution. Structured concurrency can be built on top of `spawn`, but this requires care and effort on the part of the programmer. The `spawn` statement should be used with care.

2.3.2 Argument Copying

Spawn guarantees that the arguments to the function being spawned are copied before the spawning thread continues to the next instruction. Thus, the following code has the expected effect:

```
for (i=0; i<N; i++)
  spawn f(i);
```

The argument to `f()` is copied before the next instruction executes (which will increment i). Thus, at the end of this sequential `for` loop, there are possibly $N+1$ concurrent threads of control: the original thread and the `N` threads spawned in the loop. Each of the N instances of the function `f()` has a distinct value for its integer argument. We can say nothing, however, about when each of the instances of `f()` will begin or terminate execution, either with respect to each other or with respect to the spawning thread.

For consistency with the C and C++ language definitions, the order of argument evaluation for the spawned function is not defined, but all side-effects are guaranteed to occur *before* the spawned function begins execution. Consider the following example:

```
spawn f(i++, &i)
```

We do not know when `f()` will begin execution, but when it does it will have a pointer to the incremented value of `i` (unless of course the value of `i` has been modified; see Section 2.3.4).

2.3.3 Unstructured Termination

Because the termination of a spawned thread is not synchronized with the spawning thread, it is an error (compile-time checked) to spawn a function that returns a value. All spawned functions must be void functions.

Again, because there is no synchronization between spawned and spawning threads, care must be taken that `main()` does not terminate before any of the spawned threads. The following example illustrates the problem:

```
void f(int i) {...}
int main()
{
  spawn f(2);
  return 0;
}
```

The end of a CC++ program is defined to occur (as with C and C++) at the termination of `main()`. Thus, the program in the above example could terminate before `f()` begins execution. Such behavior is almost certainly a programming error.

The lack of implicit synchronization with `spawn` transfers responsibility for synchronization to the programmer. Barriers, or any other form of synchronized behavior, must be explicitly programmed. We will return to this question once we have introduced the synchronization mechanism provided by CC++ (Chapter 4).

2.3.4 Sharing Data

The same rules that apply to sharing mutable variables in parallel blocks and `parfor` apply to `spawn` as well. Usually, the pass-by-value semantics of function calls in C and C++ prevents such sharing.

```
{
  int i = 1;
  spawn incr(i);
  i++;
  spawn incr(i);
}
```

There is no dangerous sharing of variables here because each instance of `incr()` has its own *copy* of the value of `i`.

However, care must be taken when pointers (or C++ references) are used in function arguments. This can lead to inadvertent, dangerous sharing of mutable variables.

```
{
  int a = 1;
  spawn f(&a); //DANGER: possible sharing of mutable a
  if (a==1) {...}
}
//end of a's scope, so spawned thread could reference garbage
```

Even if `a` is not explicitly modified by either `f()` or the spawning thread, this example illustrates another possible danger. The scope of the variable `a` is defined in the spawning thread. Without an explicitly programmed barrier, the variable `a` could reach the end of its scope and be implicitly destroyed in the spawning thread, leaving `f()` with an invalid pointer.

2.3.5 Pitfalls

Some potential problems to keep in mind when using `spawn`:

1. Spawned functions cannot return a value.

2. Explicit synchronization points must be programmed when using spawned functions; otherwise there is no guarantee they will begin execution before the end of `main()` is encountered. How to construct such synchronization points is discussed in Chapter 4.

3. Great care must be exercised when passing pointers (or C++ references) to spawned functions, as this often leads to dangerous sharing of mutable variables.

4. To understand complicated spawning expressions, the precedence in the order of evaluation in a function call must be understood. The function call is spawned, not the evaluation of any prefix operators. For example:

   ```
   spawn f->g()->h();
   ```

   First the pointer `f` is evaluated, then the function `g()` is executed, and then the result is used to determine which function `h()` is spawned. The spawning occurs only at the highest level function call.

2.3.6 Examples

Because no mechanism for synchronization with spawned threads of control has yet been introduced, we postpone the presentation of any examples of this construct until Chapter 4.

Chapter 3

Atomicity

3.1 Introduction

In this chapter we introduce the concept of atomicity. Because this construct is related to the notion of classes and member functions, some familiarity with the object-oriented aspects of C++is assumed.

In Chapter 2 the concept of threads of control executing in a parallel manner was introduced. A rule was presented for avoiding dangerous behavior by not sharing mutable variables between concurrent threads of execution. Sometimes, however, this sharing is necessary. Consider, for example, an implementation of a queue class. The following implementation is typical:

```
class Node  {
  public:
    int item;
    Node* next;
    Node (int i) { item = i; }
};
```

```
class Queue  {
  private:
    Node* head;
    Node* tail;
  public:
    Queue (void)  {
      head = NULL;
      tail = NULL;
    }

    void enqueue (int i)  {
      Node* add = new Node(i);
      if (head==NULL)  {
        head = add;
        tail = add;
      }
      else  {
        tail->next = add;
        tail = add;
      }
    }

    int dequeue (void)  {
      int ret_val = 0;
      if (head != NULL)  {
        ret_val = head->item;
        old_head = head;
        head = head->next;
        if (head == NULL)
          tail = NULL;
        delete old_head;
      }
      return ret_val;
    }
};
```

Now consider a Queue that can be used by an arbitrary and varying number of threads of control, all executing in parallel. Obviously this can lead to trouble if one thread of control accesses the queue by interrupting another thread that was already accessing the queue. We would like a mechanism to specify that once a particular member function has begun executing, no other member functions (from a particular set) of that object will begin executing. This mechanism is provided in CC++ by the keyword `atomic`. Atomicity is a mechanism for controlling the granularity of permitted interleavings of parallel threads of control.

Member functions (private, public, or protected) of an object can be declared atomic. This declaration specifies that the actions of such a function will not be interleaved with the actions of any other atomic function of the same object. In our queue example, both the `enqueue()` and the `dequeue()` operations would be declared atomic.

```
class Queue  {
  ...
  atomic void enqueue (int i) {...}
  atomic int dequeue (void) {...}
};
```

As a simpler example, consider the following program:

```
class Value  {
  private:
    int x;
  public:
    atomic void assign (int i)
      { x = i; }
};

void f(void)
{
  Value v;
  par  {
    v.assign(1);
    v.assign(2);
  }
  //v.x is now either 1 or 2
}
```

Two threads of control are created in the parallel block, each executing an atomic function of the object `v`. Because atomic functions that are members of the same object cannot execute concurrently, one atomic function executes first and is then sequentially followed by the execution of the second atomic function. The nondeterminism of the interleaving of actions within a parallel block is reflected in the fact that we do not know which atomic function will execute first. But once one atomic function begins execution, it will not be interrupted by the other atomic function. The two possibilities for the flow of control in this example are illustrated in Figure 3.1.

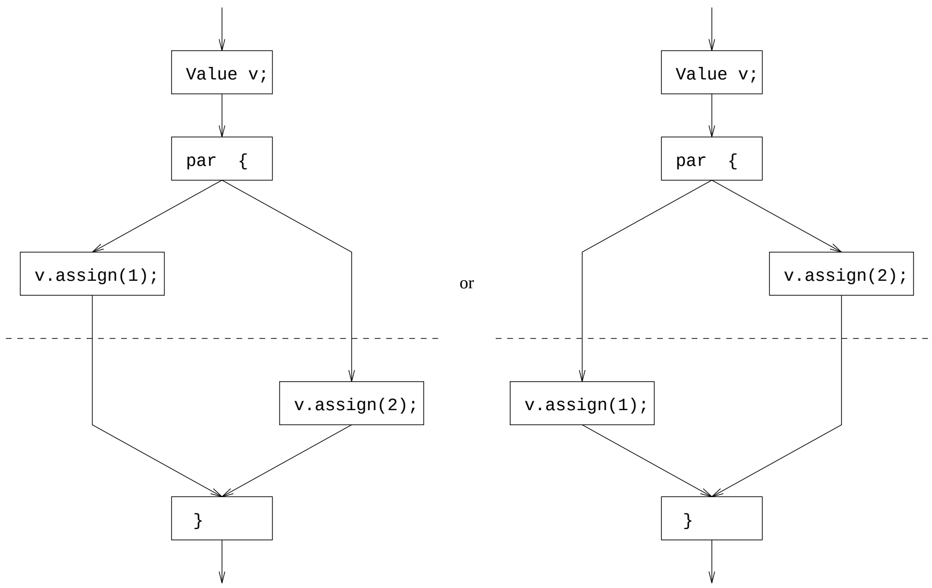

Figure 3.1: Flow of Control with Atomic Functions

Atomic functions should always be used to access mutable variables that are shared between concurrent threads of control.

3.2 Controlled Nondeterminism

In Chapter 2 it was stressed that arbitrary sharing of mutable variables between concurrent threads of control is a dangerous practice. This is because the manner in which the operations on these shared mutables are interleaved is unknown. Therefore nothing can be said about the outcome of such a program. Atomicity gives us a way to control this interleaving, and hence to control the nondeterminism of parallel composition. The example presented in Section 3.1, for instance, results in `v.x` having the value 1 or the value 2. Without an atomic access to `v.x`, however, this program would result in `v.x` having an arbitrary value.

Atomic functions can be used to write deterministic programs, despite the nondeterminism inherent in concurrent access to shared mutables. For example, consider finding the minimum element of an array:

```
class Min {
  private:
    int current_min;
  public:
    Min(int i)
      { current_min = i; }
    atomic void check (int i)
     { if (i<current_min) current_min = i; }
};

int main()
{
  int A[N];
  ...
  Min m(A[0]);
  parfor (int i=1; i<N; i++)
    m.check(A[i]);
  ...
}
```

Because of the nature of `parfor`, we do not know the order in which each of the `N`-1 parallel threads of control initiates execution of `m.check(A[i])`. However, once one thread begins execution of this `m.check()` function, no other thread is permitted to begin execution of any other atomic member functions of object `m` (including, of course, other instances of `m.check()`). Figure 3.2 represents one possible sequence of execution for this program.

Thus, the order in which `m.current_min` is updated is nondeterministic. However, the nature of the application guarantees that at the end of the parfor statement, `m.current_min` will have been compared (and updated)

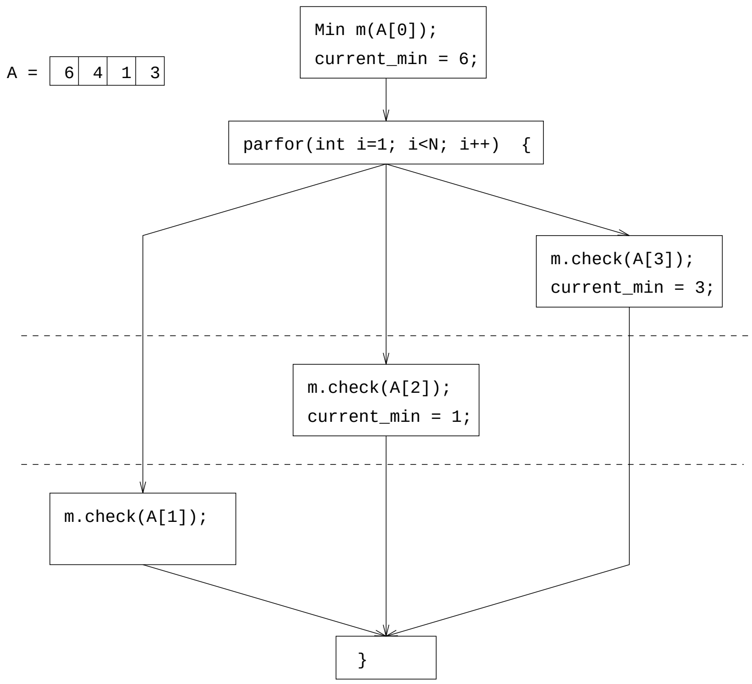

Figure 3.2: Possible Sequence of Execution for Finding Minimum Element

to the minimum element of array `A[]`. Hence, given the input array `A[]`, the intermediate values taken on by `m.current_min` are unknown, but the final value is fixed.

It is important to notice how this atomic function represents a significant bottleneck in the computation of this minimum element. Because only a single thread of control is allowed to be executing an atomic member function of `m` at any given time, the execution is essentially sequential. This suggests that atomic functions should be kept very small.

3.3 Deadlock

The execution of an atomic function represents a significant control over the rest of the computation. No other threads of control will be permitted to begin execution of an atomic function on the same object as the executing atomic function, until that executing atomic function terminates. Thus, it is possible to write an atomic function that prevents the rest of the computation from proceeding by preventing any other atomic functions from executing. This is an example of deadlock.

Fortunately, there is a small collection of simple rules for avoiding deadlock. The following rules guarantee that an atomic function will not cause a computation to deadlock:

1. atomic functions must terminate.
2. atomic functions must not suspend (suspension is discussed in Chapter 4).
3. atomic functions must not contain parallel blocks or parfor statements.
4. atomic functions must not call other functions.

Notice that the above collection of rules is a stronger set of requirements than strictly required. It is possible to write a program that violates one or more of these rules and yet will not deadlock. Following these rules, however, *guarantees* that no atomic function will cause a deadlock.

Let us examine each of these requirements in turn.

The first two rules prevent a single atomic function from monopolizing the computation by preventing any other atomic function from executing. It is possible, however, to write a program with a nonterminating atomic function which is deadlock-free. For example, if no other threads of control require atomic access to the same object as the nonterminating atomic function, no deadlock will occur. Of course, in this case, declaring the function atomic has no effect.

The third rule prevents deadlock at a nested level of scope within an atomic function. Certainly if an atomic function does not contain a parallel block or a parfor, no deadlock between concurrent threads of control within the atomic function is possible. Again, however, it is possible to write programs that violate this rule and yet will not result in deadlock. For example, an atomic function that contains a parallel block that can be guaranteed not to deadlock is perfectly safe.

The last rule prevents an atomic function from not terminating due to a deadlock in a function called by the atomic function. Clearly if no functions are called from within an atomic function, such deadlock cannot occur. Again this requirement is too strong, and it is possible to write atomic functions that do call member functions and yet will never deadlock.

Though it is possible to write deadlock-free programs that violate the rules given above, this programming style is strongly discouraged. Such programs can easily contain subtle errors that, because of various timing or dependency issues, may go undetected for a long time.

As a general rule, atomic functions should be used sparingly, and then only to do the most fundamental operations.

3.4 Inheritance

Atomicity of member functions is preserved under inheritance. Base classes and derived classes can contain atomic member functions. The atomic members of an object of a derived class behave the same as for a simple class without inheritance. That is, regardless of whether the atomic member is declared in the base class or the derived class, it is an atomic member of the derived class. For example:

```
class Base {
  protected:
    atomic void f(int) {...}
};
```

```
class Derived :  private Base {
  public:
    atomic int g(void) {...}
    void h(int i)
      {
        f(i); //executes atomically with respect to g()
        ...
      }
};
```

In an object of type `Derived`, instances of `g()` and `f()` execute atomically.

3.5 Pitfalls

The following issues should be kept in mind when using atomic functions.

1. Atomic functions should be used sparingly. This is because of the high performance cost associated with the decrease in parallelism they represent. Many applications will have no need for atomic functions.

2. When it is necessary to use an atomic function, encapsulate only what is absolutely necessary within the atomic function. These functions should be small and simple.

3. Because atomic functions provide a means to manipulate shared mutable variables, it is easy to fall into the trap of a busy wait similar to the fair interleaving pitfall described in Section 2.1.5. For example, consider:

   ```
   class Trouble {
     private:
       int x;
     public:
       atomic void assign (int i)
         { x = i; }
       atomic int check (void)
         { return x; }
   };
   ```

```
int main()
{
  Trouble t;
  t.assign(0);
  par  {
    t.assign(1);
    while (t.check()!=1) {}
  }
  ...
}
```

Though the language guarantees that *eventually* operations from both threads of control in the parallel block will get a chance to execute, there is no guarantee about how soon an operation from a particular thread (say the first one, which assigns 1 to `t.x`) will be chosen. Thus, we cannot be guaranteed to observe the termination of this parallel block.

4. Because atomic functions are associated with an object, static members cannot be declared atomic. Similarly, functions at global scope (i.e. non-member functions) cannot be declared atomic.

5. The actions within an atomic function *can* be interleaved with actions from nonatomic members of the same object. Thus, it is important to protect not only the write operations on mutable variables inside atomic functions, but also the read operations. The following class, for example, permits dangerous sharing.

```
class Protect {
  private:
    int x;
  public:
    atomic void write (int i) { x = i; }
    int read (void)           { return x; }
};
```

To rectify the problem, the member function `read()` should be declared atomic as well. (Of course, there is no problem if the class is used in a manner that guarantees that no instance of `write()` is composed in parallel with any instances of `read()`.)

3.6 Examples

In this section we give some complete examples that can be compiled and executed. These examples illustrate the use of atomic member functions.

Hello World This program is a variation on the traditional greeting program presented as the first example of Chapter 2.

```
#include <iostream.h>
#include <string.h>

class Greeting  {
  private:
    char* s;

  public:
    Greeting (void)               { s = new char[20];  }
    atomic void append (char* add) { strcat(s,add);  }
    void display (void)           { cout << s << endl;  }
};

int main()
{
  Greeting g;
  par  {
    g.append("hello, ");
    g.append("world");
  }
  g.display();

  return 0;
}
```

This program displays one of the two following messages: `"hello, world"` or `"worldhello, "`. This message is built up by appending strings to the private mutable variable `g.s`. These appends can be safely done in parallel because the function is atomic. We do not know, however, which append will be performed first. The two possible interleavings of these append operations result in two different messages which can be displayed. Notice that `display()` is not atomic. This is not a problem because the program does not compose any operations in parallel with `display()`.

Finding the Minimum Element This program finds the minimum element of a statically defined integer array.

```
#include <iostream.h>
const int N = 8;
int A[N] = {3, 4, 5, 1, 2, 5, 9, 6};

class Min {
  private:
    int current_min;

  public:
    Min(int i)
    {
      current_min = i;
      cout << "minimum initialized at " << current_min << endl;
    }

    atomic void check (int i)
    {
      cout << "comparing " << i << "...";
      if (i<current_min) current_min = i;
      cout << "minimum so far is " << current_min << endl;
    }

    int value (void)
    {
      return current_min;
    }
};

int main()
{
  Min m(A[0]);
  parfor (int i=1; i<N; i++)
    m.check(A[i]);

  return m.value();
}
```

Each element of the array is compared to the smallest value seen so far. If the element is smaller, then the smallest value seen so far is updated. By the nature of a `parfor` loop, we do not know the order in which elements will be compared using class `m`. The atomicity of the member `check()`, however, guarantees that there will be no interference between concurrent threads operating on `m`. Thus, the mutable variable `m.current_min` is protected, and the result of the program is deterministically the smallest element of the array.

Multiple-Reader Multiple-Writer Linked List This example defines a class that implements a linked list of integers. This class can be shared by multiple processes adding elements to the list (writers) and multiple processes making removals from the list (readers).

```
#include <iostream.h>

class List;

class ListNode  {
  private:
    int data;
    ListNode* next;

    ListNode (int d)
    {
      data = d;
      next = NULL;
    }
  friend class List;
};

class List  {
  private:
    ListNode* head;
    ListNode* tail;

  public:
    List (void)
    {
      head = new ListNode(0);
      tail = head;
    }

    atomic void append (int a)
    {
      ListNode* addition = new ListNode(a);
      tail->next = addition;
      tail = addition;
    }
```

```
    atomic int remove (int& item)
    {
      if (head==tail)
        return 0;
      else  {
        ListNode* old_head = head;
        head = head->next;
        delete old_head;
        item = head->data;
        return 1;
      }
    }
};

List L;

void producer (int id, int n)
{
  for (int i=0; i<n; i++)
    L.append(id*n+i);
}

int consumer (int id, int n)
{
  int item;
  int sum = 0;
  for (int i=0; i<n; i++)
    if (L.remove(item) == 1)
      sum += item;
  return sum;
}
```

```
int main()
{
  par  {
    producer(0,10);
    producer(1,10);
  }
  int sum0, sum1;
  par  {
    sum0 = consumer(0,10);
    sum1 = consumer(1,10);
  }
  cout << "Sum of list received by consumer 0:  " << sum0 << endl;
  cout << "Sum of list received by consumer 1:  " << sum1 << endl;
  return 0;
}
```

Because modifications (appends and removals) to this list are atomic, an object of this list class can be shared between multiple threads of control. Instances of an `append()` operation and a `remove()` operation are guaranteed not to interfere with each other, or with other instances of the same operation. Thus, in the examples, two producers can safely be composed in parallel, as can two consumers. The result of the composition of these two producers is a linked list containing the integers 0 to 9 (in that order) interleaved with the integers 10 to 19. The language definition says nothing about how these two sequences will be interleaved (though a particular implementation may have a specific strategy).

Chapter 4

Synchronization

4.1 Introduction

Until now, we have discussed how parallel threads of execution can be created (with `par`, `parfor`, and `spawn`) and how the granularity of the interleaving of actions in different threads of execution can be controlled (with `atomic`). The only mechanism for synchronization between concurrently executing threads of control has been the implicit barrier at the end of a parallel block and at the end of a `parfor` statement. In this chapter, we introduce a mechanism for programming arbitrary synchronization behavior between concurrent threads of control.

The sharing of a mutable variable (unprotected by atomic access) between actions composed in parallel is dangerous when at least one of the actions *modifies* the value of this variable. By contrast, it is always safe to share constants (i.e. C or C++ `const` variables). In this spirit, CC++ defines a new type of variable, a single-assignment variable (or delayed initialization constant), denoted by the keyword `sync`. Like a constant, the value of a defined single-assignment variable cannot be modified. Attempting to modify the value of a defined single-assignment variable is a run-time error. Unlike a constant, however, a single-assignment variable need not be defined when it is declared. The definition can be postponed until some later point. Thus, a single-assignment variable can be in one of two states: undefined (as it is initially) or defined. Once defined, there is no difference between a single-assignment variable and a constant.

Here are some examples of declarations of single-assignment variables:

```
sync int a;            //sync integer
char *sync b;          //sync pointer to a mutable character
sync char* c;          //mutable pointer to a sync character
sync float D[N];       //array of sync floats
sync int *sync e;      //sync pointer to a sync integer
```

The keyword `sync` is analogous to the keyword `const`. It can be used anywhere that `const` can be used. Any regular C or C++ type can be declared to be single-assignment (see Section 4.7 for an exception).

Single-assignment variables provide a means for synchronization because of the following rule:

> *If a thread of control attempts to read a single-assignment variable that has not yet been defined, that thread suspends execution until that single-assignment variable has been defined.*

Thus, threads of control that share access to a single-assignment variable can use that variable as a synchronization element.

4.2 Data Dependencies and Flow of Control

Consider the following code:

```
{
  sync int a,b,c,d;
  par  {
    a = b+c;
    b = 2;
    c = 3;
    d = a+c;
  }
  //at this point, a=5, b=2, c=3, d=8
}
```

The data dependencies in this calculation control the flow of control. The first thread of execution, that defines the single-assignment integer `a`, cannot proceed until both `b` and `c` have been defined. The second and third threads of control can proceed immediately with the definition of `b` and `c` respectively. Once `a` and `c` have been defined, then the fourth thread of control can define `d`. Notice that in this example, `d` could have been a mutable integer, since it is not shared between any threads. The flow of control for this code is schematically represented in Figure 4.1.

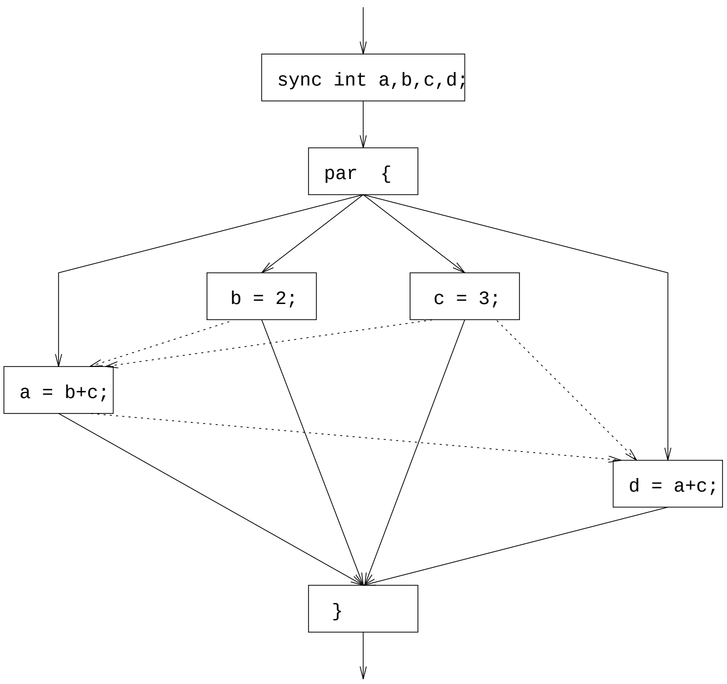

Figure 4.1: Effect of Single-Assignment Variables on Flow of Control

As another example, consider the problem of calculating all the powers of 2 from 0 to N-1. We use the fact that the i^{th} power of 2 can be calculated as $2 * 2^{i-1}$.

```
{
  sync int P[N];
  P[0] = 1;
  parfor (int i=1; i<N; i++)
    P[i] = 2*P[i-1];
}
```

Recall that we do not know in what order or in what interleaving the threads of execution created by a `parfor` statement will be executed. The semantics of single-assignment variables, however, guarantee that a value will not be assigned to `P[i]` until a value has been assigned to `P[i-1]`. The data dependencies for this program are represented in Figure 4.2. Notice how the strict linear data dependency of this example constrains the execution to essentially a sequential one.

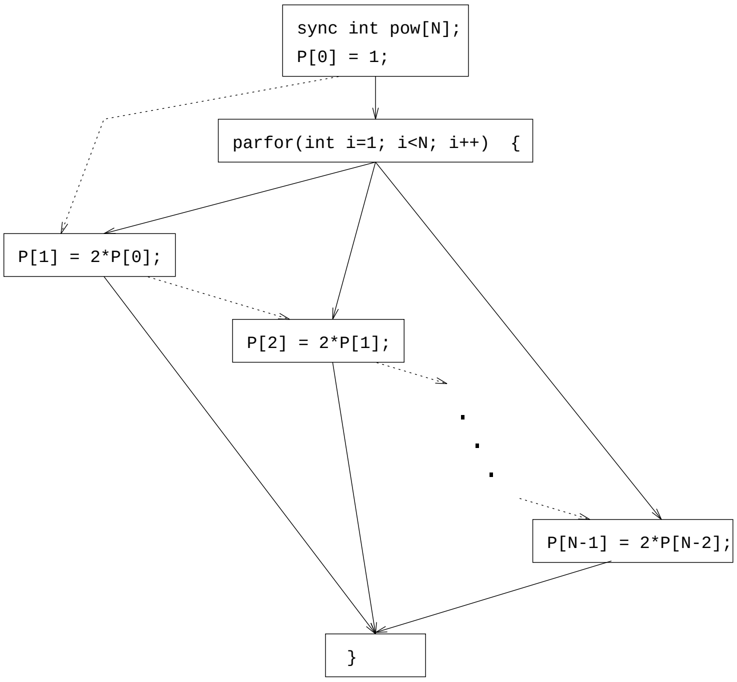

Figure 4.2: Linear Data Dependency for Calculating Powers of 2

It is also worthwhile mentioning that this example can be simply rewritten as a correct sequential program by replacing the `parfor` statement with a `for` statement. Of course, this need not be the case in general. For example, if the bounds for the loop are reversed, so that `i` begins with a value `N-1` and is decremented to 1, the corresponding sequential program would no longer be correct. This correspondence between sequential and parallel programs suggests methods of systematic parallelization of certain kinds of sequential code structures. It also suggests a deterministic debugging methodology for parallel CC++ programs.

With a slight modification to the previous example, the amount of parallelism possible can be dramatically improved. Consider:

```
sync int P[N];
P[0] = 1;
P[1] = 2;
parfor (int i=2; i<N; i=i+2)
  par  {
    P[i] = P[i/2] * P[i/2];
    P[i+1] = P[i/2] * P[i/2] * 2;
  }
```

This program makes use of the fact that the i^{th} power of 2 can be calculated as $2^{i/2}*2^{i/2}$ when i is even, and $2^{(i-1)/2}*2^{(i-1)/2}*2$ when i is odd. This modifies the data dependencies in the computation from a linear structure (as seen in Figure 4.2) to a tree structure, represented in Figure 4.3.

Again notice that this program can be simply rewritten as a correct sequential program by replacing the `parfor` statement with a `for` statement, and replacing the parallel block with a sequential one.

4.3 Single-Assignment Arguments and Return Values

Single-assignment variables can be used as function arguments (again, in exactly the same way that constant variables can be used). The pass-by-value semantics of function invocation in C and C++ guarantees that the single-assignment variable can be copied (and hence has been defined) before the function begins execution. For example:

```
void f(sync int i)  //Suspends here until i is defined
{
  // At this point, we can assert that i has been defined
  ...
}
```

Similarly, a function can return a single-assignment type. This is not generally a useful thing, however, since an individual statement is evaluated sequentially in CC++. For example, to evaluate the expression `a()+b()`, first the function `a()` is executed, then the function `b()` is executed, then

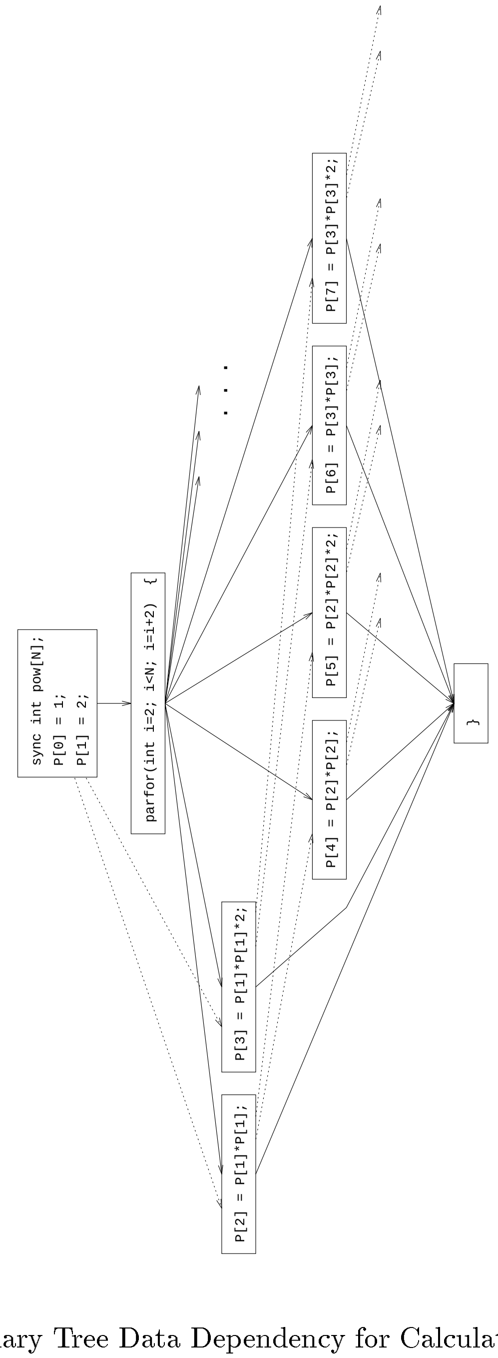

Figure 4.3: Binary Tree Data Dependency for Calculating Powers of 2

their result is summed. No potential parallelism between these two functions is exploited, so no synchronization in the form of single-assignment variables is required. Because functions that are spawned must be `void` functions, a single-assignment return type is not useful in that case either. Thus, it is always possible to replace a function that returns a single-assignment type with one that returns a mutable type, without altering the semantic meaning of the program.

4.4 Type Conversions

The single-assignment nature of a `sync` variable cannot be cast away, neither implicitly nor explicitly. This guarantees that a single-assignment variable cannot be misused (that is, modified) in a thread of control. For example, consider the following examples:

```
{
  void f (int *);
  sync int a;
  int b;

  b = a;         //OK
  b = (int)a;    //OK

  (int)a = 3;    //ERROR
  f((int *)&a); //ERROR
}
```

4.5 Synchronizing Spawned Functions

Synchronization for threads of control created with the `spawn` command must be explicitly programmed. This can be done using single-assignment variables (in conjunction with pointers or C++ reference arguments). For example, a barrier between a spawning thread of control and the function that it spawns might be programmed as follows:

```
void independent (sync int* b)  {
  ...
  *b = 1;
}

int main()
{
  sync int Barrier;
  spawn independent(&Barrier);
  ...
  if (Barrier == 1)          //spawning thread waits here until
                              //function independent() has set sync value
    {;}
  ...
}
```

It is important to recall that the semantics of the **spawn** statement requires that all the function arguments be evaluated prior to the function beginning execution. This means that if one of the arguments is a single-assignment variable, the spawned function will not begin execution until that variable has been defined. Consider the following code:

```
void f (sync int* p, sync int n)
  { *p = n; }

int main()
{
  sync int A[3], B[3];
  A[2] = 1;
  B[2] = 1;
  par  {
    f(&A[0],A[1]);              //OK
    f(&A[1],A[2]);
  }
  spawn f(&B[0],B[1]);          //program suspends here forever
  spawn f(&B[1],B[2]);
  ...
}
```

The parallel block executes correctly because initialization (e.g. argument evaluation) and execution of both instances of the function `f()` are composed in parallel with each other. The `spawn` statements following this parallel block, however, must be executed in strict sequential order. Completing the first `spawn` statement means evaluating the arguments to the function `f()` – that is, the address of `B[0]` and the value of `B[1]`. Because `B[1]` is an undefined single-assignment variable, the spawn statement itself suspends. Execution does not proceed to the next statement.

The sharing of references and pointers to single-assignment variables by concurrent threads of execution can be extremely useful. By contrast, the sharing of references and pointers to mutables by concurrent threads of execution can be dangerous. Because the spawned function is composed in parallel with the spawning function, pass-by-reference semantics for mutable variables can lead to dangerous sharing between concurrent threads of control (see Chapter 2, Section 2.3.4).

4.6 Memory Management

The lifetime of a single-assignment object obeys the usual C++ scoping conventions. At the end of the block in which it is declared, a single-assignment variable goes out of scope, and is destroyed. In this case, the memory management is handled implicitly.

To create a single-assignment variable whose lifetime extends beyond the scope of its declaration, dynamic memory allocation can be used. Again, such allocation is completely consistent with how the allocation would be done for a constant value in C++, using the `new` operator. For example:

```
{
  sync int* a = new sync int;
  ...
  *a = 3;
}
```

The declaration above creates a pointer to an undefined single-assignment integer. The assignment later defines the single-assignment integer referenced by `a` to be the value 3.

When dynamic memory allocation is used, C++ makes it the programmer's responsibility to deallocate this memory, freeing it for future use. This applies to single-assignment variables as well. Thus, corresponding to the declarations above, we might expect to see:

```
delete a;
```

4.7 Pitfalls

Keeping the following points in mind when using single-assignment variables will help to avoid many common mistakes.

1. Single-assignment variables can be read by other threads of control *immediately* after their definition has terminated. For example, a structure that contains some single-assignment fields and some mutable fields must usually be initialized such that the mutable fields are defined before the single-assignment fields. For an example of the subtlety of this pitfall, see the second example in Section 4.8 of this chapter.

In addition, the present implementation of the compiler places the following restrictions on the single-assignment construct:

1. A user-defined class cannot be declared to be single-assignment. The `sync` construct can only be applied to fundamental types (that could, in turn, be part of a user-defined class).

4.8 Examples

In this section we present several examples that can be compiled and executed and that illustrate the use of single-assignment variables (in conjunction with some of the constructs seen in previous chapters).

All-Pairs Shortest Paths This program calculates the length of the shortest path between all pairs of vertices in a directed, acyclic graph. The graph is defined statically by its adjacency matrix.

```
#include <iostream.h>
const int N = 4;

int min (int a, int b)
{
  if (a < b)  return a;
    else      return b;
}

sync int path_lengths[N+1][N][N];
int edges[N][N] = { 0,    1,    8,   4,
                    1,    0, 1000,   2,
                    8, 1000,    0,   4,
                    4,    2,    4,   0 };

void initialize_paths (void)
{
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      path_lengths[0][i][j] = edges[i][j];
}

void solve (void)
{
  for (int k=1; k<=N; k++)
    parfor (int i=0; i<N; i++)
      parfor (int j=0; j<N; j++)
        path_lengths[k][i][j] = min (path_lengths[k-1][i][j],
                   path_lengths[k-1][i][k-1] + path_lengths[k-1][k-1][j]);
}

void display_result (void)
{
  cout << "All-pairs distance matrix is:" << endl;
  for (int i=0; i<N; i++)  {
    for (int j=0; j<N; j++)  {
      cout << path_lengths[N][i][j] << "\t";
    }
    cout << endl;
  }
}
```

```
int main()
{
  initialize_paths();
  solve();
  display_result();

  return 0;
}
```

This example uses the dynamic programming recurrence relation

$$\forall i, j, k : (0 \le i, j \le N-1) \wedge (1 \le k \le N) :$$
$$D^k_{i,j} = \min(D^{k-1}_{i,j},\ D^{k-1}_{i,k-1} + D^{k-1}_{k-1,j})$$

where $D^k_{i,j}$ is the minimum distance between vertices i and j, using only vertices numbered strictly less than k as intermediate vertices.

In this example, all $\mathtt{N}^3$ calculations are composed in parallel with each other. If any of the three values required to calculate `path_lengths[k][i][j]` are not yet defined, that thread of control suspends. Thus, the order of evaluation of this three-dimensional array is controlled by the data dependencies of each element of the array on the previous elements. Again, it is important to note that the small size of each task to be performed in parallel relative to the number of these tasks makes this program extremely inefficient. The dominant cost here is the thread creation and termination time (as opposed to the calculations performed) and thus we would expect to observe a degradation in performance in any practical implementation. This example is meant only to illustrate the semantic meaning of single-assignment variables and the functional style of programming they induce.

Synchronizing Single-Reader Single-Writer Stream This example implements a single-reader, single-writer stream. Two operations are defined on such a stream: an append and a removal. A removal from an empty stream suspends until the stream is non-empty. Appends to the stream never suspend.

```
#include <iostream.h>

class Stream;
```

```
class StreamNode  {
  private:
    int data;
    StreamNode *sync next;
    StreamNode (int d) { data = d; }
  friend class Stream;
};

class Stream  {
  private:
    StreamNode* head;
    StreamNode* tail;

  public:
    Stream (void)
    {
      head = new StreamNode(0);
      tail = head;
    }

    void append (int a)
    {
      StreamNode* addition = new StreamNode(a);
      tail->next = addition;
      tail = addition;
    }

    int remove (void)
    {
      StreamNode* old_head = head;
      head = head->next;
      delete old_head;
      return head->data;
    }
};

Stream S;

void producer (int n)
{
  for (int i=0; i<n; i++)  {
    cout << "[appending " << i << "]";
    S.append(i);
  }
}
```

```
void consumer (int n)
{
  for (int i=0; i<n; i++)
    cout << "Consumer removes :  " << S.remove() << endl;
}

int main()
{
  par  {
    producer(10);
    consumer(10);
  }
  return 0;
}
```

The last `StreamNode` of a stream always has an undefined `next` field. An empty stream is represented by a single `StreamNode` with an undefined `next` pointer. See Figure 4.4.

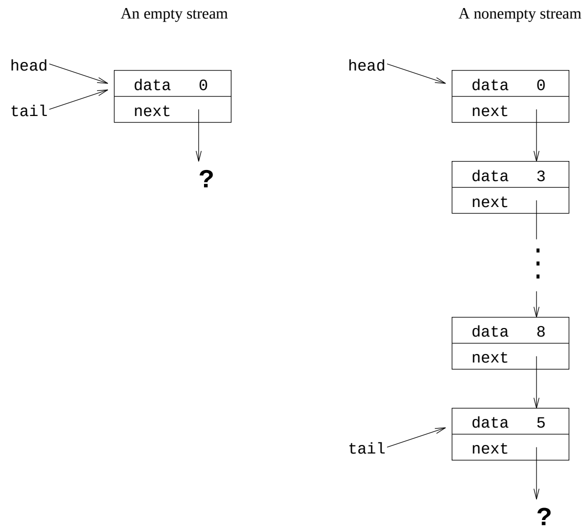

Figure 4.4: A Stream Implemented as a Linked List with Single-Assignment Links

Appending an item means creating a new node with the appropriate data, defining the `next` field of the last `StreamNode`, and modifying the mutable `tail` member to point to this new node. Notice that because this action is not atomic, this modification of `tail` is unprotected. Hence concurrent `append()` operations are dangerous, and so this is a single-writer stream. A multiple-writer class can be created by simply making `append()` an atomic operation.

Removing an item requires reading the `next` field of the `StreamNode` referenced by the mutable member `head`. If this field is not defined, the removing thread of control suspends here. Once this field is defined, the data contained in the `StreamNode` referenced by this `next` field is returned, the first node is deleted, and the mutable `head` member is modified to point to this second node. Again, this modification is unprotected, and so concurrent `remove()` operations are dangerous. Unlike the `append()` operation, however, this member cannot simply be made atomic to permit multiple readers. This is because atomic functions must not suspend (recall Chapter 3, Section 3.3).

This example illustrates some of the complexity involved in implementing such synchronization classes that contain mutable members and will be shared between concurrently executing threads of control. There are at least two common errors that are avoided in the above implementation. Both stem from the fact that if a thread of control is suspended on an undefined single-assignment variable, that thread may resume execution *immediately* once that single-assignment variable has been defined:

1. When adding a new `StreamNode` to the stream, the node must be created and the data field initialized *before* the new node is linked on to the stream. If the node is linked first, and then the data filled in, a suspended `remove()` operation could resume execution immediately when the `next` field of the previous node is defined and attempt to access a garbage data field.

2. The first node can be deleted by a `remove()` operation as soon as the stream becomes non-empty, which occurs in the second line of the `append()` function. Thus, we must be careful not to access the contents of this node in the last line of `append()`. The following code, for example, is incorrect:

```
void append (int a)  {
  StreamNode* addition = new StreamNode(a);
  tail->next = addition;
  tail = tail->next;            //ERROR: tail may point to a deleted node
}
```

If the stream was initially empty (so `head` and `tail`) point to the same node), then using the value `tail->next` may dereference deleted memory, since at this point a `remove()` operation may have deleted the node referenced by `head`.

Fortunately, these issues of synchronization and interaction based on shared objects can usually be encapsulated in a small collection of classes. These classes can be rigorously analyzed and verified and used whenever appropriate. For example, libraries that implement semaphores, monitors, and a variety of message-passing channels have been implemented and verified here at Caltech.

Chapter 5

Distributed Hello World

5.1 Introduction to Distributed Computing

An address space is the set of memory that can be accessed from a thread of control. So far, all the concurrency we have created using `par`, `parfor`, and `spawn` has been inside a single address space. Modulo the scoping boundaries imposed by the C++ language, each thread of control has access to the same memory. This means that communicating data from one thread to another simply requires agreeing on which location in memory to place the information. However, simultaneous access to data by multiple threads is nondeterministic. We introduced `atomic` and `sync` to control this nondeterminism.

We are now going to talk about distributing a computation over several address spaces. Threads on separate address spaces no longer have access to the same memory. Thus, communication of data from one address space to another is required for two such threads to share data. This communication is often quite time-consuming. However, we now need to be concerned only with nondeterminism caused by interaction with other threads on the same address space, rather than with all threads in the entire computation.

Because communication is now more expensive, deciding in which address space to place which pieces of data becomes important. Each thread would like access to pieces of data it frequently uses to be inexpensive, i.e., in the same address space. We would like to distribute the computation to the available address spaces in such a way that each piece can inexpensively access most of the data on which it depends.

In C++ objects, we group the data related to pieces of computation (member functions) together. CC++ extends this idea with *processor objects*. Each processor object is a separate address space. We group related pieces of data, and the parts of the computation that go with them, into one processor object.

Naturally, we cannot always break the computation up such that each piece can inexpensively access all the data on which it depends. In CC++, data that is expensive to access is distinguished from data that is inexpensive to access. Pointers that reference data that is expensive to access (i.e., on another address space/processor object) are *global pointers*, while those that reference inexpensively accessible data (i.e., on the same processor object) are local pointers.

Dereferencing a global pointer creates a communication to another processor object to fetch the value referenced. The specifics of this communication are controlled through the CC++ construct of *data transfer functions.*

We will see processor objects, global pointers, and data transfer functions in detail in the next 3 chapters. First we present a simple example of their use.

5.2 Distributed Hello World

Let us modify the 'Hello world' example presented in Chapter 2 to say hello from a set of processor objects. We use three files:

```
// dist_Greeter.h
#include <iostream.h>

global class Greeter { // global identifies a processor object type
  public:
    Greeter() {}
    void say_hi (int id);
};
```

```
// dist_Hello.cc++
#include <stdlib.h>
#include "dist_Greeter.h"
```

```
// argv[1] - # of Greetings desired (integer)
// argv[2]..argv[1+argv[1]] -
//      machines on which processor objects should be located (strings)
int main (int argc, char** argv)
{
  int P = atoi(argv[1]); // P becomes # of processor objects to be created
  parfor(int p=0; p<P; p++) {
    Greeter *global G;
    proc_t placement = proc_t("dist_Greeter.out",argv[2+p]);
    // placement of processor object is specified by a (definition,location) pair
    G = new (placement) Greeter();
    G->say_hi(p);
    delete G;
  }
  return 0;
}


// dist_Greeter.cc++
#include "dist_Greeter.h"

void Greeter::say_hi (int id)
{
  cout << "Hello World from Processor Object#" << id << endl;
}
```

From our shell we compile two executables:

```
>cc++ dist_Greeter.cc++ -ptype=Greeter -o dist_Greeter.out
>cc++ dist_Hello.cc++ -o dist_Hello.out
```

We must now start PVM, a communication library that CC++ uses. Basically this means invoking `pvmd` with a file (the hostfile) that lists the machines we plan to use in our computation. The release notes for the CC++ compiler provide more detailed instructions for executing distributed CC++ computations than given here.

We execute

```
>pvmd hostfile
```

and then place `pvmd` in the background.

Finally we are ready! We run `dist_Hello.out`, telling it how many greetings we want, and a corresponding list of locations from which we want greetings. For instance, here at Caltech we might write

```
>dist_Hello.out 3 fides hebe fides
```

As described in the section of the release notes entitled "Running a Distributed CC++ Program", the standard output from `dist_Hello.out` will be piped to a file in the `tmp` directory of the machine you started `pvmd` from.

Let us examine the parts of this program that are not standard C++.

1. The keyword `global` qualifying the class declaration on line 2. This identifies `Greeter` as a processor object type. Each object of type `Greeter` will be a separate address space. Note that other than the word `global`, `class Greeter` looks like any other class: processor objects have constructors, destructors, private, public and protected members, and can be inherited. Processor objects are explained in Chapter 7.

2. The keyword `global` qualifying the pointer declaration on line 16. This identifies `G` as a global pointer. A global pointer can reference memory in other processor objects, and thus is the basic mechanism for communication in CC++. Global pointers are explained in Chapter 6.

3. The object `placement` of type `proc_t` on line 17. `proc_t` is an implementation-defined type that specifies placement of a processor object. In our implementation of CC++, `proc_t` contains two fields: an executable name and a machine name. The executable name states where the definition of the processor object can be found, and the machine name states on what machine that processor object should be created. We compiled the definition of type `Greeter` into `dist_Greeter.out`, and we take the $2+p$th argument in array `argv` as the machine name.

4. The allocation `G=new (placement) Greeter()` on line 19. This creates an object of type `Greeter`, placed according to the `proc_t placement`. Like all calls to `new`, a pointer to the newly created object is returned. Since that object is a processor object, by definition it resides in another address space, and therefore `G` must be a global pointer.

5. The function call `G->say_hi(p)` on line 20. This invokes the member function `say_hi` on the object referenced by `G`. Since `G` references another processor object, the function will be executed on that processor object. This is known as a remote procedure call, or RPC. `say_hi` takes an argument, which is transferred to the processor object where the function is to execute. If `say_hi` had a return type, the value returned would be transferred back. The mechanism for controlling how data is transferred is explained in Chapter 8.

6. The deallocation `delete G` on line 21. This destroys the processor object referenced by `G`. All variables inside the processor object are destroyed, and any member functions of the processor object currently executing are halted. The execution of `dist_Greeter.out` on machine `argv[2+p]` is terminated. Deallocation of a processor object is trickier than that of other objects: a processor object might have other member functions executing when the destructor is run. This will be discussed in Chapter 7.

7. The compilation `cc++ dist_Greeter.cc++ -ptype=Greeter -o dist_Greeter.out`. In CC++, a processor object type is defined by an executable. The `-ptype=` linker option names the processor object type that this executable defines. When a processor object is created, CC++ checks to insure that the type specified in the `new` statement and the type assigned to the executable match. Here we define the processor object type `Greeter` by the executable created from compiling `dist_Greeter.cc++`. The actual name of the executable doesn't matter.

The next few chapters will explore the concepts presented above in greater detail. Global pointers, data transfer functions, and processor objects will be examined.

Chapter 6

Global Pointers

6.1 Introduction

We have introduced distributed computations as those using several address spaces, and defined a processor object to be an address space. We postpone a detailed explanation of how these processor objects are defined and created until Chapter 7. In this chapter we discuss how global pointers are used to communicate data between processor objects, assuming the processor objects have been created.

In CC++ there are two types of pointers: global pointers and local pointers. Global pointers can reference addresses in any processor object in the computation. Local pointers can only reference addresses in the processor object in which they are created. Global pointers identify data that is expensive to access, while local pointers identify data that is inexpensive to access.

Global pointers are used much like local pointers. When a global pointer is dereferenced, the value it references is returned. If the global pointer references an object, member functions can be invoked through the pointer. In both these cases, since the object resides on another processor object, an implicit communication is performed to fetch the value or call the function.

Global pointers are declared by using the keyword `global` to modify a pointer declaration. Here are some examples of declarations of global pointers:

```
int *global gpint;     // global pointer to an integer
int * *global gppint; // global pointer to a local pointer to an integer
C *global gpC;         // global pointer to an object of type C
```

Global pointers can reference basic types and user-defined structures, but in the current implementation they may not reference functions. Thus we may not declare

```
int (*global gpf)(); // ERROR - global pointer to function returning int
```

6.2 Dereferencing Global Pointers

Because the communication needed to fetch the value referenced by a global pointer is implicit, we write expressions involving global pointers as if they were local pointers. For example:

```
{
  int *global gpint;
  int x = *gpint+1;
}
```

Here `x` is assigned the sum of 1 and the value referenced by `gpint`.

We can use `gpint` without knowing in which processor object the integer referenced by it resides. The integer might even be in the processor object where this statement is executed. If this is the case, then the expression is equivalent to the same expression using a local pointer:

```
{
  int* lpint;
  int x = *lpint+1;
}
```

The current implementation of CC++ does not take full advantage of global pointers that reference local memory. Dereferencing such global pointers will take longer than if they were local pointers, but not as long as if the value resided on another processor object.

6.3 Invoking Functions Through Global Pointers

The communication needed to invoke a member function of an object referenced by a global pointer is implicit, as is the transfer of arguments to the processor object in which the object resides. For example, we write

```
{
  C *global gpC;
  gpC->a_function();
}
```

to invoke the function `a_function()` on the object referenced by `gpC`. This mechanism of function invocation through a global pointer is known as a remote procedure call, or RPC.

Each RPC creates a separate thread of control on the remote processor object. Thus, several RPCs can execute concurrently. `atomic` and `sync` should be used to avoid the dangerous sharing of mutables that might result.

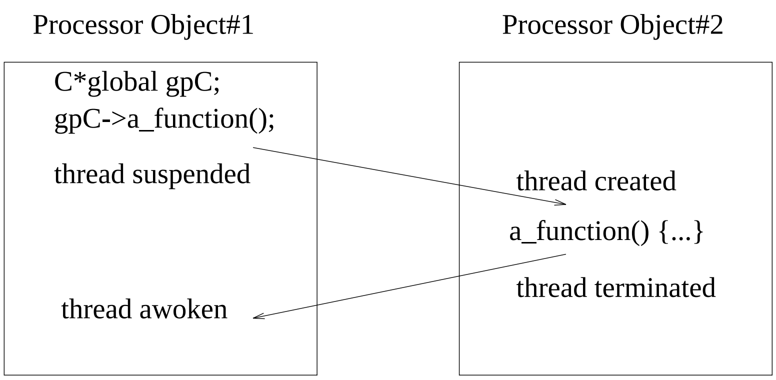

Figure 6.1: Flow of Control in an RPC

The semantics of function call are preserved by an RPC. That is, the function call statement does not terminate until the function has terminated on the remote processor object. The flow of control in an RPC is illustrated in Figure 6.1.

If the function has a return value, it is returned to the processor object that made the call. Thus, we can write the following:

```
{
  C *global gpC;
  int x = gpC->a_function_returning_int()+10;
}
```

CC++ has a mechanism for controlling how the arguments and return values of functions called remotely are transfered between processor objects. This mechanism is described in Chapter 8.

Again, a global pointer does not have to reference an address in another processor object. If the address referenced is on the processor object from which the function is invoked, the effect is the same as a function invocation through a local pointer.

Functions that are called remotely may not have arguments that are local pointers, references, or arrays. This is because these types cannot be copied from one processor object to another. This topic is also covered in Chapter 8. For the same reason, remote functions may not return local pointers, references, or arrays. Violating either restriction will result in a compile-time error.

6.4 Casting Global Pointers

Casting a global pointer to a local pointer when the global pointer does not reference an address in that processor object is an error. After such a cast, the address in the local pointer does not reference the same memory that the global pointer did.

Because of this danger, CC++ will not implicitly cast a global pointer to a local pointer. If the global pointer really references memory on the current processor object, then an explicit cast can be used.

```
{
  int *global gpi;
  int* pi= (int *)gpi;  // Think Carefully Before Using!
}
```

An explicit cast must only be used when it is certain that the global pointer references memory in the local processor object. A run-time error results if this is not the case.

Local pointers are implicitly, and can also be explicitly, cast into global pointers. Here are the four possibilities:

```
{
  int* pi;
  int *global gpi = pi;   // Implicit local-to-global cast OK
  int *global gpi2 = (int *global)pi;   // Explicit local-to-global cast OK
  pi = gpi;   // Implicit global-to-local cast COMPILE-TIME ERROR
  pi = (int *)gpi;   // Explicit global-to-local cast POSSIBLE RUN-TIME ERROR
}
```

In this example, the explicit global-to-local cast would not be an error, since we initialized `gpi` using the local pointer `pi`!

6.5 Pitfalls

1. A global pointer takes more memory than a local pointer, and it takes more time to dereference. Global pointers should be used to indicate that the data referenced is expensive to obtain.

2. When creating objects whose member functions will be invoked through RPCs, keep in mind that each RPC creates a separate thread of control, and that concurrently executing RPCs on the same object might dangerously share mutable variables.

3. Global pointers cannot be ordered, i.e. the relational operators `>`,`<`,`<=`, and `>=` cannot be used with global pointer operands. They may be compared for equality or inequality using the operators `!=` and `==`. An expression comparing a global pointer to 0 (NULL) will evaluate to true if the global pointer points to 0 in that processor object.

   ```
   {
     int *global gpint1;
     int *global gpint2;
     if (gpint1>gpint2) {...} // COMPILE-TIME ERROR
     if (gpint1==gpint2) {...} // OK
     if (gpint1==0) {...} // OK
   }
   ```

6.6 Examples

A Distributed List Let's modify the synchronizing single-reader single-writer linked list presented in Chapter 4 to append items on one processor object and remove them on another.

We need to separate the list into two objects:

- `DList_appending`, which will exist in the appending processor object.
- `DList_removing`, which will exist in the removing processor object.

The data that has been appended but not removed will be stored in `DList_removing`. Thus, we need to transfer an append request to this object. To do this, we need a global pointer, as `DList_removing` is in a different processor object than `DList_appending`.

`DList_appending` contains a member `removing_side`, which is a global pointer to an object of class `DList_removing`. The member function `append` just forwards the request through this global pointer, calling the member function of `DList_removing` named `real_append`.

The declaration in the header file `gptr_dlist.h` is as follows:

```
class DList_removing;

class DListNode  {
  private:
    int data;
    DListNode *sync next;
    DListNode (int d) { data = d; }
  friend class DList_removing;
};

class DList_removing  {
  private:
    DListNode* head;
    DListNode* tail;
    atomic void real_append (int a); // Called by DList_appending
  public:
    DList_removing (void);
    int remove (void);
  friend class DList_appending;
};
```

```
class DList_appending  {
  private:
    DList_removing *global removing_side;
  public:
    DList_appending(DList_removing *global lr) :  removing_side(lr) {}
    void append (int a);
};
```

The definition in `gptr_dlist.cc++` is as follows:

```
#include "gptr_dlist.h"

DList_removing::DList_removing (void)
{
  head = new DListNode(0);
  tail = head;
}

atomic void DList_removing::real_append (int a)
{
  DListNode* addition = new DListNode(a);
  tail->next = addition;
  tail = addition;
}

int DList_removing::remove (void)
{
  DListNode* old_head = head;
  head = head->next;
  delete old_head;
  return head->data;
}

void DList_appending::append (int a)
{
  // Use RPC to add item to remote list
  removing_side->real_append(a);
}
```

The member function `real_append` of object `DList_removing` is atomic so that several RPCs to `DList_removing` can be executing concurrently but not be dangerously sharing mutables. This makes this list a single-reader multiple-writer list. The `sync next` field of `ListNode` ensures that a `real_append` and a `remove` will not be sharing mutable data.

Producer and Consumer Even though the global pointer is not necessary for the list presented above if the appender and remover are in the same processor object, it will still function correctly. We can see it working with this program, `gptr_prod_cons.cc++`

```
#include <iostream.h>
#include "gptr_dlist.h"

class Consumer {
  public:
    DList_removing* remover;
    Consumer()  { remover = new DList_removing(); }
    ~Consumer() { delete remover; }

    void consume (int n)
    {
      for (int i=0; i<n; i++)
        cout << "Consumer removes:  " << remover->remove() << endl;
    }
};

class Producer {
  public:
    DList_appending* appender;

    Producer(DList_removing *global remover)
    {
      appender = new DList_appending(remover);
    }

    ~Producer() { delete appender; }

    void produce(int n)
    {
      for (int i=0; i<n; i++)  {
        cout << "[appending " << i << "]";
        appender->append(i);
      }
    }
};
```

```
int main (int argc, char**argv)
{
  Consumer C;
  Producer P(C.remover);
  par {
    P.produce(10);
    C.consume(10);
  }
  return 0;
}
```

We compile and run as follows:

```
>cc++ gptr_dlist.cc++ -c
>cc++ gptr_prod_cons.cc++ -o gptr_prod_cons.out gptr_dlist.o
>pvmd &
>dpc.out
```

In Chapter 7, after we have seen how to create processor objects, we use this distributed list class across processor objects.

Chapter 7

Processor Objects

7.1 Introduction

A processor object is a collection of data and computation that defines a single address space. Although each processor object is a separate address space in a CC++ computation, each processor object does not have to be located on a physically distinct address space.

This distinction between the virtual address spaces (processor objects) used in specifying the computation and the physical address spaces used to implement it is important. It allows us to separate the problem of defining the computation from the problem of distributing that computation to the available resources. We specify the computation in terms of abstract objects, and then define the mapping from abstract objects to available resources. If the available resources change, we do not have to change the definition of the computation, only the mapping.

As a trivial example, we saw this in the example in Chapter 5, where we executed

```
>dist_Hello.out 3 fides hebe fides
```

This created three `Greeter` processor objects, two on a machine at Caltech named fides and one on a machine named hebe. If we get another machine, say named rhea, we can execute

```
>dist_Hello.out 4 fides hebe rhea fides
```

without redefining what a `Greeter` does.

In this chapter we will go through a more complex example: a distributed mergesort. We will explain the syntax behind declaring, defining, allocating, using, and destroying processor objects.

Mergesort can be thought of as a tree of processes, each leaf of which sorts a segment of the array, and each interior node of which merges two branches, eventually resulting in a completely sorted list at the root. Figure 7.1 shows the interaction of these two types of objects, `Merger` and `Sorter`.

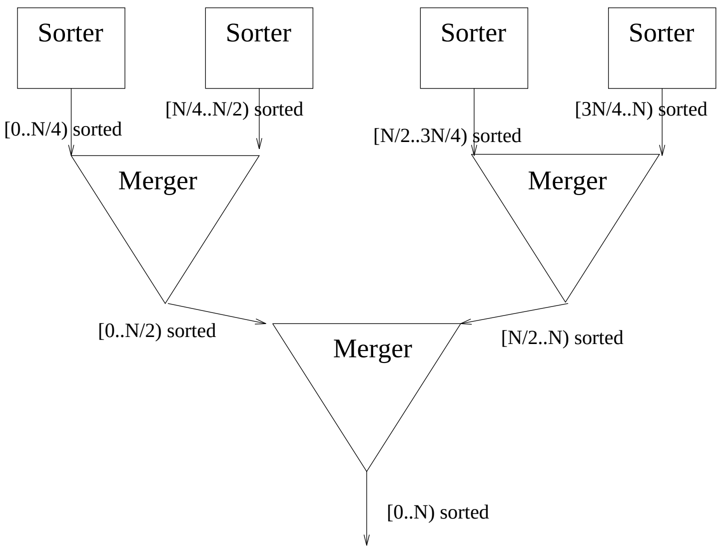

Figure 7.1: MergeSort

After we have defined `Merger` and `Sorter`, we can write a mergesort that uses these objects.

7.2 Declaring Processor Object Types

A processor object type is declared when a class or structure declaration is modified by the keyword `global`. The processor object class specifies the interface to objects of that type. Public member functions and data may be accessed by anyone with a global pointer to that processor object.

Processor object types can be inherited. As with C++ objects, private and protected members are only accessible from member functions of that processor object, or objects derived from it.

In our mergesort, we have `Merger` objects and `Sorter` objects. We declare them in the common header file `pobj_MergeSort.h`

```
#include "gptr_dlist.h"        // Distributed linked list
const int ENDVALUE = -1;

global class Sorter  {   // Sort a list and place it into out
  private:
    int start_index;
    int stop_index;
    DList_appending *global out;
    void sort();

  public:
    Sorter (DList_removing *global out_receiver, int start, int stop);
};

global class Merger {   // Merge sorted in1 and in2 into sorted out
  private:
    DList_removing* in1;
    DList_removing* in2;
    DList_appending* out;
    void merge();

  public:
    Merger (DList_removing *global);
    DList_removing *global get_in1() { return in1; }
    DList_removing *global get_in2() { return in2; }
};
```

We are going to use the distributed list built in Chapter 6 to send sorted lists between our processor objects. Thus, each `Sorter` has a global pointer to a `DList_removing` on the `Merger` object that is its parent in the tree. Similarly, each `Merger` has a global pointer to its parent.

7.3 Defining Processor Object Types

Processor object types are defined by assigning a type to an executable compiled using CC++. The processor object type to assign to an executable is specified using the compiler option `-ptype=`. The type must have been declared in the executable; otherwise, a link-time error will result.

Defining a processor object as an executable means there are two types of members for processor objects: implicit and explicit. Implicit members are those functions and objects at file scope in the executable, while explicit members are those explicitly declared in the processor object type. Implicit members are protected members of the processor object type and cannot be accessed using a global pointer to the processor object.

In our mergesort, we will define the `Sorter` processor object by compiling the file `pobj_Sorter.cc++`, shown here:

```
// Definition of Member Functions of Processor Object Sorter
#include "pobj_MergeSort.h"

void Sorter::sort()
{
  // Sort a portion of an array, perhaps reading it from disk.
  // In this example, just output a sorted list of numbers.
  for (int i=start_index; i<stop_index; i++)
    out->append(i);
  out->append(ENDVALUE);
}

Sorter::Sorter (DList_removing *global remover, int start, int stop)
  :  start_index(start), stop_index(stop)
{
  out = new DList_appending(remover);
  spawn sort();
}
```

We define the constructor `Sorter::Sorter` and the member function `Sorter::sort` as explicit members of the type `Sorter`. When we compile this using

```
>cc++ pobj_Sorter.cc++ -o pobj_Sorter.out -ptype=Sorter gptr_dlist.o
```

all file scope objects and variables in `gptr_dlist.o` become implicit members of `Sorter`. The executable `pobj_Sorter.out` is now a processor object of type `Sorter`.

We similarly define `Merger`

```
>cc++ pobj_Merger.cc+ -o pobj_Merger.out -ptype=Merger gptr_dlist.o
```

where `pobj_Merger.cc++` contains

```
// Definition of Member Functions of Processor Object Merger
#include "pobj_MergeSort.h"

void Merger::merge()
{
  int top1 = in1->remove();  // Smallest UnMerged Element in in1
  int top2 = in2->remove();  // Smallest UnMerged Element in in2

  while ((top1!=ENDVALUE) && (top2!=ENDVALUE))  {
    if (top1<=top2)  {
      out->append(top1);
      top1 = in1->remove();
    }
    else  {
      out->append(top2);
      top2 = in2->remove();
    }
  }
  while (top1!=ENDVALUE)  {
    out->append(top1);
    top1 = in1->remove();
  }
  while (top2!=ENDVALUE)  {
    out->append(top2);
    top2 = in2->remove();
  }
  out->append(ENDVALUE);
}
```

```
Merger::Merger(DList_removing *global remover)
{
  in1 = new DList_removing();  in2 = new DList_removing();
  out = new DList_appending(remover);
  spawn merge();
}
```

7.4 Allocating Processor Objects

Processor objects are allocated using the C++ `new` operator:

```
{
  proc_t placement("pobj_Merger.out","fides");
  Merger* global merger1 = new (placement) Merger(constructor-arguments)
}
```

The placement argument must be of type `proc_t`. `proc_t` is an implementation-defined type that specifies where to place a processor object and where to find its definition. In our implementation of CC++, `proc_t` contains two fields: an executable name and a machine name. The executable name states where the definition of the processor object can be found, and the machine name states on what machine that processor object should be created.

The interface to type `proc_t` is as follows:

```
class proc_t {
  public:
    char* host_name;
    char* executable_path;
    proc_t();
    ~proc_t();
    proc_t (const proc_t &);
    proc_t (char* executable,char* host);
    proc_t & operator=(const proc_t &);
};
```

When creating a processor object, CC++ checks that the type assigned to the executable given in the `proc_t` matches the type of the processor object being created. If these do not match, a run-time error occurs. For example, we get a run-time error with this piece of code:

```
{
  proc_t placement("pobj_Sorter.out","fides");
  Merger *global merger = new (placement) Merger(constructor-arguments);
}
```

The type assigned to `pobj_Sorter.out` was `Sorter`, while the allocation statement is creating an object of type `Merger`. The call to `new` returns a global pointer to the newly created processor object.

7.5 Using Processor Object Pointers

A processor object acts like any other C++ object: it stores data members and can be requested to perform member functions on that data. Invoking a member function of a processor object through a global pointer to that processor object results in a thread of control being created to perform that member function. When the member function terminates, that thread is terminated. Multiple member functions can be executing on a processor object simultaneously.

These member functions might return global pointers to objects in the processor object. For instance, in our mergesort we need a global pointer to the `DList_removing` object in a `Merger` in order to construct a `Sorter` object. Thus, member functions (`get_in1()` and `get_in2()`) in type `Merger` return global pointers to their `DList_removing` members.

Thus, we could create a mergesort with two sorters and one merger as follows:

```
{
  proc_t merger_placement("pobj_Merger.out",argv[2]);
  proc_t sorter0_placement("pobj_Sorter.out",argv[3]);
  proc_t sorter1_placement("pobj_Sorter.out",argv[4]);

  Sorter *global sorters[2];
  Merger *global merger;

  // Create Merger Processor Object
  DList_removing* final_output = new DList_removing();
  merger = new (merger_placement) Merger(final_output);
```

```
  // Create Sorter Processor Objects
  DList_removing *global merger_left_input = merger->get_in1();
  sorters[0] = new (sorter0_placement) Sorter(merger_left_input,0,N/2);

  DList_removing *global merger_right_input = merger->get_in2();
  sorters[1] = new (sorter1_placement) Sorter(merger_right_input,N/2,N);
}
```

7.6 Deallocating Processor Objects

Processor objects are deallocated using the C++ `delete` operator:

```
delete merger;
```

When a processor object is deallocated, all member functions currently running are terminated. Deleting a pointer to an object that has already been deleted results in undefined behavior.

Since all member functions which are executing on a processor object are terminated when a processor object is deallocated, we have to be careful. Many threads of control in the computation may be waiting for member functions of the deleted processor object to complete. These threads of control will be suspended forever, perhaps resulting in the suspension of our entire computation.

In our mergesort example, the constructors for `Sorter` and `Merger` `spawn` member functions `sort` and `merge` respectively. The semantics of CC++ make no guarantees about when these functions will terminate. However, we know that when the end of the merged output stream is received, these functions have in fact terminated and it is safe to delete all the processor objects.

7.7 CC++ Computations

A CC++ computation is initiated by specifying an initial processor object. Only in this processor object is the function `main` executed. This processor object may create other processor objects, which may create still other processor objects.

A computation is terminated when `main` terminates on the initial processor object, or when `exit()` or `abort()` is called from any processor object.

Terminating a computation results in the termination of all threads of control on all processor objects and the deallocation of all processor objects.

The initial processor object is specified by executing a program of the type of the initial processor object. When we compile a CC++ program without specifying a type for the executable, an anonymous type is created. Thus, all the programs we wrote in Chapters 2- 4 defined anonymous processor object types. When we executed them, we created a single processor object.

7.8 The ::this Pointer

Every processor object member, whether implicit or explicit, has a pointer to the processor object on which it is being invoked. This pointer is analogous to the C++ `this` pointer. In CC++, `::this` is a pointer to the current processor object. In the current implementation, however, this syntax is replaced by `THIS(type)` where `type` is the type of the current processor object.

7.9 Pitfalls

Take care to remember these things when using processor objects:

1. Multiple threads of control can be executing on one processor object at any one time, since anyone with a global pointer to a processor object can perform an RPC. Use `atomic` and `sync` to prevent dangerous sharing.

2. The destructor for a processor object is just another member function of that object. It can be running concurrently with other threads on the processor object, and will not wait for those other threads to finish before deallocating the processor object. In CC++ it is bad style to finish a computation when all processor objects have not been deallocated. The system will try to deallocate those processor objects left by the user. The system is not always able to do this, and in CC++ the consequences of an undeallocated processor object are significant: a process left running, wasting processor time and resources. That process may even exist on another machine. Ending a computation with `exit()` from any processor object guarantees that all processor objects are deleted, while ending it with `abort()` will not. Killing a

single process in the computation, for instance using the UNIX `kill` command, will not terminate the entire computation.

In addition, the current implementation has the following pitfalls:

1. The syntax `delete []` to delete an array of pointers cannot be used with an array of pointers to processor objects.

2. CC++ defines a function, called the entry function, for each type to handle RPCs to objects of that type. This function is automatically generated by the compiler. When compiling many modules into one executable, the same type can be declared many times. If the entry function is defined in each module, a link-time error will result. Because of this, CC++ defines the entry function for a type at the point of the first non-inline non-constructor member function of that type. If there are no non-inline non-constructor members of a type, you can force entry functions to be defined at the point of type declaration by using the compiler option `+ee1`.

7.10 Examples

MergeSort Here is a complete `MergeSort` that uses the `Merger` and `Sorter` processor objects discussed in this chapter. This `pobj_MergeSort.cc++` creates 2 sorters and 1 merger, splitting the work evenly between the sorters.

```
#include "pobj_MergeSort.h"
#include <iostream.h>
#include <stdlib.h>
```

```
int read_output (DList_removing* out, int N)
{
  int prev = -1;
  int all_correct = 1;
  for (int i=0; i<N; i++)  {
    int temp = out->remove();
    if (temp<prev)  {
      cout << "GOT ITEM #"<<i<<" OUT OF ORDER" << endl;
      all_correct = 0;
    }
    prev = temp;
  }
  out->remove(); // ENDVALUE
  return all_correct;
}

int main (int argc, char* argv[])
{
  if (argc<5)  {
    cout << "MergeSort::Not enough arguments.    Expect:" << endl;
    cout << "  Argument 1) # of Elements to sort (N)" << endl;
    cout << "  Argument 2) Machine to place merger" << endl;
    cout << "  Arguments 3,4) Machines to place sorters" << endl;
    exit(1);
  }
  int N = atoi(argv[1]);
  proc_t merger_placement("pobj_Merger.out",argv[2]);
  proc_t sorter0_placement("pobj_Sorter.out",argv[3]);
  proc_t sorter1_placement("pobj_Sorter.out",argv[4]);

  Sorter *global sorters[2];
  Merger *global merger;

  // Create Merger Processor Object
  DList_removing* final_output = new DList_removing();
  merger = new (merger_placement) Merger(final_output);

  // Create Sorter Processor Objects
  DList_removing* global merger_left_input = merger->get_in1();
  sorters[0] = new (sorter0_placement) Sorter(merger_left_input,0,N/2);

  DList_removing* global merger_right_input = merger->get_in2();
  sorters[1] = new (sorter1_placement) Sorter(merger_right_input,N/2,N);
```

```
  // Check that output list is in ascending order
  int result = read_output(final_output,N);
  if (result==0)  cout << "Incorrect MergeSort" << endl;
    else          cout << "Correct MergeSort" << endl;

  // Deallocate Processor Objects
  delete merger; delete sorters[0]; delete sorters[1];
  return result;
}
```

To compile and run this, we write

```
>cc++ pobj_MergeSort.cc++ -o pobj_MergeSort.out gptr_dlist.o
>pvmd hostfile &
>pobj_MergeSort.out 100 fides hebe rhea
```

Chapter 8

Data Transfer Functions

8.1 Introduction

In Chapter 6 we learned that when a function with arguments is invoked through a global pointer, those arguments are copied to the remote processor object and the function invoked with those copies. Function return values are similarly transferred back to the processor object that invoked the remote function.

While transferring the arguments is simple if they are basic types, it is more complex when they are user-defined structures, particularly if they contain local pointers. (Recall that local pointers are only valid in the processor object in which they are created.)

To give you control over how types are transferred, in CC++ every type has a pair of functions which define how to transfer that type to another processor object. These functions are the data transfer functions for that type.

Once defined for a type, these functions are automatically invoked by the compiler to perform all transfers of that type. You do not need to call these functions explicitly; they are invoked implicitly by calling a function through a global pointer that takes an argument of that type. They are also automatically invoked when a remote function returns a value of that type.

The function

```
CCVoid& operator<<(CCVoid&,const TYPE& obj_in);
```

defines how TYPE should be packaged up. It is called by the compiler whenever an object of TYPE needs to be transferred to another processor object.

Similarly, the function

```
CCVoid& operator>>(CCVoid&,TYPE& obj_out);
```

defines how TYPE should be unpackaged. It is called by the compiler whenever an object of TYPE is received from another processor object. Upon termination, `obj_out` will be a copy of the `obj_in` used as the argument to the `operator<<` in the initial processor object.

The type `CCVoid` is a compiler-defined type analogous to `class ios` of the `iostream` library. Data transfer functions are used much like the input and output streams of C++. In C++ the functions

```
ostream& operator<<(ostream&,const TYPE& obj_in);
istream& operator>>(istream&,TYPE& obj_out);
```

define how TYPE should be packaged to and retrieved from storage.

8.2 Building Transfer Functions

CC++ defines these packaging and unpackaging routines for the following types: basic integer types, float, double and global pointers. The basic integer types are: char, short, int, long, sync char, sync short, sync int, sync long and the unsigned varieties of each of these. With these building blocks, the transfer functions for other types can be defined. For instance:

```
class Point {
  float x_coordinate;
  float y_coordinate;
  friend CCVoid& operator<<(CCVoid&,const Point&);
  friend CCVoid& operator>>(CCVoid&,Point&);
  friend ostream& operator<<(ostream&,const Point&);
  friend istream& operator>>(istream&,Point&);
};
```

```
CCVoid& operator<<(CCVoid& v,const Point& p_out)
{
  v << p_out.x_coordinate << p_out.y_coordinate;
  return v;
}

ostream& operator<<(ostream& v, const Point& p_out)
{
  v << p_out.x_coordinate << p_out.y_coordinate;
  return v;
}

CCVoid& operator>>(CCVoid& v,Point& p_in)
{
  v >> p_in.x_coordinate >> p_in.y_coordinate;
  return v;
}

istream& operator>>(istream& v, Point& p_in)
{
  v >> p_in.x_coordinate >> p_in.y_coordinate;
  return v;
}
```

Notice the similarities between the data transfer functions and the input/output stream functions for class `Point`. The data transfer functions are declared friends of `Point` so that they may access the private data members of `Point`.

Both `istream& operator>>` and `CCVoid& operator>>` operate on an object for which memory has already been allocated and initialized. The compiler invokes the default constructor to initialize an object, and then invokes `CCVoid& operator>>` with the initialized object. Thus, a default constructor must be defined for each type. Like C++, CC++ will automatically generate a default constructor for a type if there is no other constructor defined for that type.

8.3 Structures with Local Pointers

CC++ does not define how local pointers are passed between processor objects. While the value of an integer means the same thing in all processor objects, a local pointer is valid only in the processor object in which it was created.

For structures with local pointers, then, the information needs to be packaged in such a way as to enable the reconstruction of the same structure in the other processor object. For instance:

```
class Vector {
  int length;
  double* elements;
  friend CCVoid& operator<<(CCVoid&,const Vector&);
  friend CCVoid& operator>>(CCVoid&,Vector&);
};

CCVoid& operator<<(CCVoid& v,const Vector& input)
{
  v << input.length;
  for (int i=0; i<input.length; i++)
    v << input.elements[i];
  return v;
}

CCVoid& operator>>(CCVoid& v,Vector& output)
{
  v >> output.length;
  output.elements = new double[output.length];
  for (int i=0; i<output.length; i++)
    v >> output.elements[i];
  return v;
}
```

The local pointer is never really transferred. Rather, the elements of the array that it references are sent in an agreed upon order – from lowest index to highest index – so that the identical array can be reconstructed remotely. Also notice that no constructor has been defined for type `Vector`, and thus CC++ will define one automatically.

The problems with transferring local pointers are also present for arrays, and must be dealt with similarly.

8.4 Automatic Transfer Function Generation

If there are no local pointers or arrays in a user-defined type, then the CC++ compiler can generate the correct transfer functions automatically. For instance, the correct transfer functions for `class Point` can be generated automatically, while those for `class Vector` cannot be.

This implementation of CC++ follows these rules for automatic transfer function generation:

- All types must have data transfer functions defined.
- The compiler can generate the correct transfer functions for structures where all data members are basic types, global pointers, or user-defined structures. The compiler cannot generate the correct transfer functions for types with local pointers or arrays (even statically sized).
- The compiler will generate transfer functions for all types that the user does not. If the type contains a local pointer or an array, and the user has not declared the transfer functions, a compile-time warning will be given, and the generated transfer function will not try to pass the local pointer or the array. This is a warning rather than an error so that users interested only in a single address space will not have to write data transfer functions.
- The user notifies the compiler that they will specify the transfer functions for a type by declaring them as friends of that type. The user should make these functions friends, even if that friendship is not required to access the private members of the type. A link-time error will result if these functions are declared but not defined. A link-time error will result if these functions are defined without being declared as friends in the type declaration.
- The compiler will generate either zero or two transfer functions for each type. The user may not define one transfer function and have the compiler define the other.
- When compiling multiple modules into one executable, the same type can be declared many times. If the transfer functions for that type are defined in each of them, a link-time error will result. Because of this, if the CC++ compiler is going to generate the transfer functions for a type, it does so where the first non-inline, non-constructor member

function of that type is defined. If there are no such members, then the compiler option `+ee1` will force transfer functions to be generated for all types in that compile, at the point where the type is declared.

8.5 Pitfalls

Here are some things about data transfer functions to watch out for:

1. A default constructor must be defined for all types. The default constructor is invoked before an object is unpacked using `operator>>`.

2. Although the compiler may be able to generate the correct transfer functions for a type, where correct means an identical copy of the object is produced in the remote processor objct, that may not be what you want. You can generate the transfer functions for any type you want; the compiler only generates functions for types you do not.

3. The `const` in the argument to `operator<<` is important! Modifying the structure while it is being packaged is modifying a mutable variable while it is being read.

4. Be careful when passing structures with global pointers. The compiler-generated transfer functions will pass the global pointer, not the object referenced by the global pointer. The Examples section below explores this issue in more detail.

5. It is good practice to think of `operator<<` and `operator>>` as two more functions to be defined for each type, along with the constructor, the destructor, the assignment operator, etc.

8.6 Examples

Here we present data transfer functions for some complicated structures.

Linked List Suppose we need to transfer a linked list between processor objects. The type is defined much like the linked list used in Chapter 4, except that no `sync` links are used.

```
struct ListNode {
  ListNode* next;
  int data;
  ListNode (int d) { data = d; next = 0;}
  ListNode() {}  // Default constructor to be called before  operator>>
  friend CCVoid& operator<<(CCVoid&,const ListNode&);
  friend CCVoid& operator>>(CCVoid&,ListNode&);
};

struct List {
  int size;
  ListNode* head;
  ListNode* tail;

  List() { head = 0; tail = 0; size = 0; }  // Called before  operator>>
  void append (ListNode* nn) {
    if (tail!=0) { tail->next = nn; tail = nn; }
    else { head = nn; tail = nn; }
    size++;
  }
  void remove();

  friend CCVoid& operator<<(CCVoid&,const List&);
  friend CCVoid& operator>>(CCVoid&,List&);
};
```

We might write the transfer functions as follows

```
CCVoid& operator<<(CCVoid& v,const ListNode& in)
{
  v << in.data; return v;
}

CCVoid& operator>>(CCVoid& v,ListNode& out)
{
  v >> out.data; return v;
}

CCVoid& operator<<(CCVoid& v,const List& in)
{
  v << in.size;
  ListNode* temp = in.head;
  for (int i=0; i<in.size; i++)  {
    v << *temp; temp = temp->next;
  }
  return v;
}
```

```
CCVoid& operator>>(CCVoid& v,List& out)
{ // Assume head==0 and tail==0 and size==0
  int size; v >> size;
  for (int i=0; i<size; i++) {
    ListNode* new_node = new ListNode(); v >> *new_node;
    out.append(new_node);
  }
  return v;
}
```

We send the `ListNode` structures in head to tail order, and reconstruct the list in the remote processor object. The `ListNode` structures are just integers here, but in general they could be arbitrarily complex data structures. Note that the unpacking function for the list assumes that `head==0 && tail==0 && size==0`, i.e., that the default constructor has been called for the object into which the data is being unpacked.

Linked List with global pointers If we modify `List` and `ListNode` to use global pointers, and allow the compiler to generate the transfer functions, then the code generated by the compiler would look something like this:

```
struct ListNode {
  ListNode *global next;
  int data;
  ListNode(int d) { data = d; next = 0; }
  ListNode() {}  // Default constructor to be called before  operator>>
  friend CCVoid& operator<<(CCVoid&,const ListNode&);
  friend CCVoid& operator>>(CCVoid&,ListNode&);
};

struct List {
  int size;
  ListNode *global head;
  ListNode *global tail;

  List() { head = 0; tail = 0; size = 0; }  // Called before  operator>>
  void append (ListNode* nn);
  void remove();

  friend CCVoid& operator<<(CCVoid&,const List&);
  friend CCVoid& operator>>(CCVoid&,List&);
  int size;
};
```

```
CCVoid& operator<<(CCVoid& v, const ListNode& in)
{
  v << in.next << in.data;
  return v;
}

CCVoid& operator>>(CCVoid& v, ListNode& out)
{
  v >> out.next >> out.data;
  return v;
}

CCVoid& operator<<(CCVoid& v,const List& in)
{
  v << in.size << in.head << in.tail;
  return v;
}

CCVoid& operator>>(CCVoid& v,List& out)
{
  v >> out.size >> out.head >> out.tail;
  return v;
}
```

However, these transfer functions would *not* result in the list being wholly transferred to the other processor object. When a global pointer is transferred, the object it references is *not*. Thus, the transferred list still points to the same block of memory in the initial processor object. The transferred list would look as shown in Figure 8.1.

This is known as a shallow copy of an object. A shallow copy is one where only the object, and not memory referenced by it, is copied. In contrast, a deep copy is one where the object, and all memory referenced by it, is copied. The compiler-defined global pointer transfer is a shallow copy. Thus, if you want a deep copy, you have to write the transfer function yourself.

Tree We want to write transfer functions for this tree class:

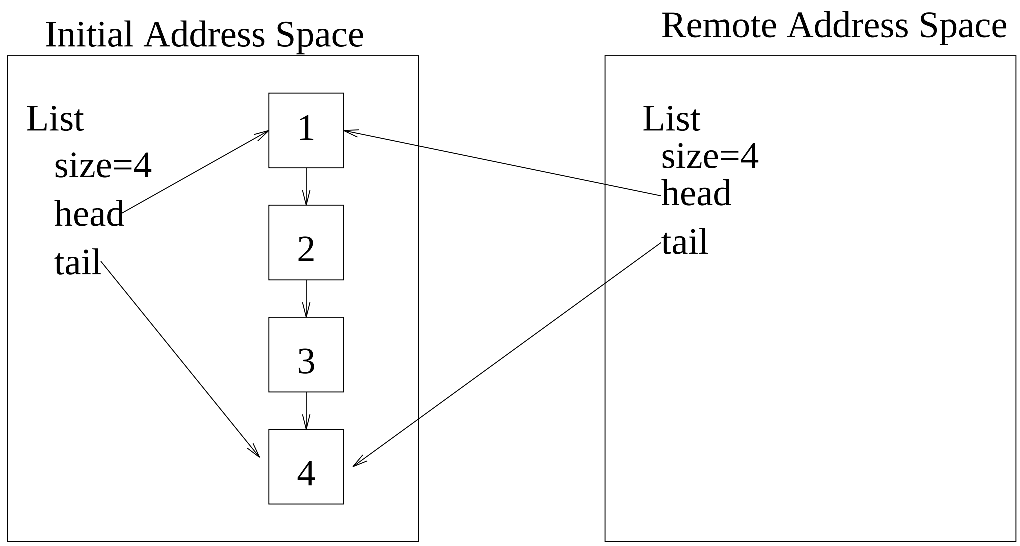

Figure 8.1: Transferred List Object

```
enum Tree_type{no_children,left_child_only,right_child_only,both_children};
struct Tree {
  int data;
  Tree_type info;
  Tree* left_child; Tree* right_child;
  Tree() {}   // Default constructor to be called before  operator>>
  friend CCVoid& operator<<(CCVoid&,const Tree&);
  friend CCVoid& operator>>(CCVoid&,Tree&);
};
```

The transfer functions might be written as follows:

```
CCVoid& operator<<(CCVoid& v,const Tree& in)
{
  int info = in.info;
  v << info;  // Transfer the enumerated type as an integer
  v << in.data;
  switch (in.info) {
   case no_children:  break;
   case left_child_only:  v << *(in.left_child); break;
   case right_child_only:  v << *(in.right_child); break;
   case both_children:  v << *(in.left_child) << *(in.right_child); break;
  }
  return v;
}

CCVoid& operator>>(CCVoid& v,Tree& out)
{
  int info; v >> info;  out.info = (Tree_type)info;
  v >> out.data;
  switch (out.info) {
   case no_children:
      out.right_child = out.left_child = 0;
      break;
   case left_child_only:
      out.right_child = 0;
      out.left_child = new Tree(); v >> *(out.left_child);
      break;
   case right_child_only:
      out.left_child = 0;
      out.right_child = new Tree(); v >> *(out.right_child);
      break;
   case both_children:
      out.left_child = new Tree(); v >> *(out.left_child);
      out.right_child = new Tree(); v >> *(out.right_child);
      break;
  }
  return v;
}
```

Again, the unpacking function initializes the already allocated object `out`. The packing and unpacking functions agree to use prefix notation for the tree, and to preface each data value with information about what, if any, children that node has.