Chapter 7: Programming with MPI

This chapter will describe and evaluate MPI, the Message Passing Interface, which provides a programming interface that is portable across all parallel computers. In particular, whereas Pthreads assume that the hardware supports a shared address space, MPI makes no such assumption. Moreover, MPI supports collections of heterogeneous machines. There are other message passing libraries, including the Parallel Virtual Machine library, PVM.

Getting Started

We will again use the count 3's example to illustrate the basics, before we discuss more advanced features. Here, we will use C bindings for MPI, but the standard also provides bindings for Fortran and C++.

Execution Model

The MPI execution model differs from Pthreads. In MPI the unit of parallelism is a process, not a thread, so each process has its own address space. The only way that two processes can communicate is to send messages to one another using the operations of `MPI_Send()` and `MPI_Recv()`. Thus, a parallel program's data structures actually consist of a collection of independent portions of data, each residing in a different process. Furthermore, MPI execution is initiated with a static number of processes, typically with processes assigned to different processors. Some external mechanism is thus needed to initiate the MPI program, specifying the total number of processors that will be used.

Initialization and Cleanup

The following code is a typical skeleton MPI program that initializes MPI for a single process and then cleans up afterwards.

```
#include <stdio.h>
#include “mpi.h”
#include “globals.h”

int main (argc, argv)
int argc;
char **argv;
{
   int myID, value, size;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &myID);

   /* compute stuff in parallel */
. . .
   MPI_Finalize()
   return 0;
68 }
```

The call to `MPI_Init()` on line 11 initializes the MPI runtime system, in this case passing along the runtime arguments with which this program was invoked. This initialization routine should be called exactly once for each MPI process.

The calls on lines 12 and 13 define a context in which communication can occur. In particular, the first argument to these calls specifies a ***communicator***, which is an MPI scoping mechanism for grouping sets of logically related communication operations. A process may use different communicators to keep logically distinct communication operations separate. In our example above, the program uses a single communicator, the predefined `MPI_COMM_WORLD` communicator that includes all available MPI processes. The call to `MPI_Comm_size()` returns to this process, through its second parameter, the number of processes that participate within this communicator, and the `MPI_Comm_rank()` call returns the ***rank*** of this process within the communicator. A rank is a unique identifier used by MPI to identify this process within the communicator. The ranks are numbered from 0 to size-1. The `MPI_Finalize()` call cleans up MPI data structures. Of course, this routine should be the last MPI function called by a process.

MPI_Comm_Size()

```
int MPI_Comm_Size (                // Retrieve size of a communicator
  MPI_Comm comm,                   // Communicator
  int *size,                       // Size
};
```

Arguments:

- The communicator of interest.
- A pointer to the size, whose target will contain the size of the specified communicator.

Return value:

- An MPI error code.

Code Spec 1. MPI_Comm_Size(). MPI routine to obtain the number of processes in a communicator.

With the skeleton code in place, we are now ready to write the meat of a parallel program. First assume that the file `globals.h` includes the following lines:

```
#define RootProcess 1
int length;
int length_per_process;
int myStart;
int myCount = 0;
int globalCount;
MPI_Status status;
int tag = 1;
```

MPI_Comm_Rank()

```
int MPI_Comm_Rank (                // Retrieve rank of a communicator
  MPI_Comm comm,                   // Communicator
  int *rank,                       // Rank
  };
```

Arguments:

- The communicator of interest.
- A pointer to the rank, whose target will contain the rank of the specified communicator.

Return value:

- An MPI error code.

Code Spec 2. MPI_Comm_Rank(). MPI routine to obtain a process' rank within in a communicator.

MPI_Send()

```
int MPI_Send (                     // Blocking Send routine
  void *       buffer,             // Address of the data to send
  int          count,              // Number of data elements to send
  MPI_Datatype type,               // Type of data elements to send
  int          dest,               // ID of destination process
  int          tag,                // Tag to distinguish this message
  MPI_Comm *   comm                // An MPI communicator
  };
```

Arguments:

Use MPI_STATUS_IGNORE if you

Notes:

- This routine has blocking semantics, which means that the routine does not return until the message is received at the destination process. See `MPI_Isend()` for a non-blocking version of the send operation.

Return value:

- An MPI error code.

Code Spec 3. MPI_Send(). MPI routine to send data to another process.

With the assumed global variables, we can now give the body of the Count 3's code..

```
      length_per_process = length/size;

      /* Read the data, distribute it among the various processes */
      if (myID == RootProcess)
      {
```

```
21        if ((fp = fopen(*argv, "r")) == NULL )
22        {
23            printf("fopen failed on %s\n", filename);
24            exit(0);
25        }
26        fscanf(fp,"%d", &length);   /* read input size */
27
28        for (p=0; p<size-1; p++)    /* read data on behalf of each */
29        {                           /* of the other processes */
30            for (i=0; i<length_per_process; i++)
31            {
32                 fscanf(fp,”%d”, myArray+i);
33            }
34            MPI_Send(myArray, length_per_process, MPI_INT, p+1,
35                     tag, MPI_COMM_WORLD);
36        }
37
38        for (i=0; i<length_per_process; i++)   /* Now read my data */
39        {
40            fscanf(fp,”%d”, myArray+i);
41        }
42    }
43    else
44    {
45        MPI_Recv(myArray, length_per_process, MPI_INT, RootProcess,
46                 tag, MPI_COMM_WORLD, &status);
47    }
48
49    /* Do the actual work */
50    for (i=0; i<length_per_process; i++)
51    {
52        if (myArray[i]==3)
53        {
54            myCount++;     /* Update local count */
55        }
56    }
57
58    MPI_Reduce (&myCount,&globalCount, 1, MPI_INT, MPI_SUM,
59                RootProcess, MPI_COMM_WORLD);
60
61    if (myID==RootProcess)
62    {
63        printf(“Number of 3’s: %d\n”, globalCount);
64    }
65
```

This code starts by having a single process, designated the Root Process, read the contents of an array from a file and distribute this data to the other processes. On Lines 21-26 the Root Process opens the specified file name, which is assumed to be the first command line argument, and then reads the size of the file. Then, on Lines 28-36, the Root Process reads the file contents in `size` chunks, sending the first `size-1` of these chunks to the other processes, and then keeping the last chunk for its own use. The data is sent to other processes on Line 34 using the `MPI_Send()` routine. Here, each message contains an array of `length_per_process` integers. This routine is an example of ***point-to-point*** communication, in which data is sent from one process to another. In MPI, such communication is specified redundantly by both the sender and the receiver, so Lines 43-47 show that each of the size-1 non-root processes invoke `MPI_Recv()` to accept the data. The first three parameters of the send and receive routines describe the message that is being sent, and the 4th parameter identifies the sending or receiving process. The 5th parameter provides a tag, which identifies this message. In our example, all of the tags are identical because there is never more than one message sent between any pair of processes. The 6th parameter identifies the communicator, and the `MPI_Recv()` routine has an additional 7th parameter that is used to return the completion status of the operation. MPI specifies that messages between the same source and destination will be delivered in order. However, MPI does not provide any fairness guarantee—when multiple processes send to the same destination process, nothing can be said about the ordering of these messages.

The actual work is performed on Lines 50-56, where each process counts the number of 3's in its portion of the array. Finally, each of the local values of count is reduced to a single value by summing them with the call to `MPI_Reduce()`. `MPI_Reduce()` is an example of a ***collective communication operation*** that involves all members of a communicator. In this case, each process provides a single integer, and all of these values are summed and returned to the root process, as specified by the 6th parameter, at the address that is specified by the second parameter. The distribution of the array data that was performed with calls to `MPI_Send()` and `MPI_Recv()` on Lines 34 and 45 could also have been performed more succinctly using `MPI_Scatter(),` a collective communication operation that distributes data from one process to all other processes.

Logic is split up, with some code applying to some processes and not others. The logic is also broken up by messages. In this example, the messages already reside in contiguous memory, but in many cases the data must be marshaled, that is, placed contiguously in memory, before it can be sent to another process. Dichotomy between local data, which can be addressed directly, and remote data, which can only be accessed through special function calls.

MPI is a very low level interface. Programmers need to specify many details and operate continually in two worlds—the local and the global. It can be challenging.

MPI_Recv()
```
int MPI_Send (                      // Blocking Receive routine
  void *       buffer,              // Address at which to receive data
  int          count,               // Number of elements to receive
  MPI_Datatype type,                // Type of each element
  int          source,              // ID of sending process
  int          tag,                 // Identifier to distinguish message
  MPI_Comm     comm,                // MPI communicator
  MPI_Status * status               // Status of this receive operation
  };
```

Arguments:
- To receive a message from any other process, use MPI_ANY_SOURCE as the source.
- To match on any tag, use MPI_ANY_TAG as the fifth parameter

Notes::
- This routine has blocking semantics—it does not return until the message the message is received. See `MPI_Irecv()` for a non-blocking version of the receive operation.

Return value:
- An MPI error code.

Code Spec 4. MPI_Recv(). MPI routine to receive data from another process.

Safety Issues
In MPI, there is no shared data, so there is no need to provide explicit mutual exclusion. There are however other safety issues, including deadlock and livelock. Moreover, because point-to-point communication is specified redundantly by both the sender and receiver, there is the need to match communication operations.

Performance Issues
We saw in Chapter 3 that dependences among threads constrain parallelism. Because locks dynamically impose dependences among threads, the granularity of our locks can greatly affect parallelism. At one extreme, the coarsest locking scheme uses a single lock for all shared variables, which is simple but severely limits concurrency when there is sharing. At the other extreme, fine-grained locks may protect small units of data. For example, in our Count 3's example, we might use a different lock to protect each node of the accumulation tree. As an intermediate point, we might use one lock for the entire accumulation tree. As we reduce the lock granularity, the overhead of locking increases while the amount of available parallelism increases.

MPI_Reduce()

```
int MPI_Reduce (                      // Reduce routine
  void *        sendBuffer,           // Address at which to receive data
  void *        recvBuffer,           // Number of elements to receive
  int           count,                // Type of each element
  MPI_Datatype  datatype,             // ID of sending process
  MPI_OP        op,                   // MPI operator
  int           root,                 // Process that will contain result
  MPI_Comm      comm                  // MPI communicator
  };
```

Notes:

- A special form of this routine, `MPI_Allreduce()`, treats all processes as if they were the root, meaning that the reduced value will be passed to all processes at the address specified by the second argument. `MPI_Allreduce()` is equivalent to a call to `MPI_Reduce()` followed by a call to `MPI_Bcast()`, which broadcasts values to all processes within a communicator.

Code Spec 5. MPI_Reduce(). MPI routine to perform reduction operation.

Reducing Communication Latency

Why are there so many flavors of point-to-point communication? To understand this, realize that there is significant synchronization and copy of data that must occur for each point-to-point communication operation because the message must be copied across four address space, as shown in Figure 1.

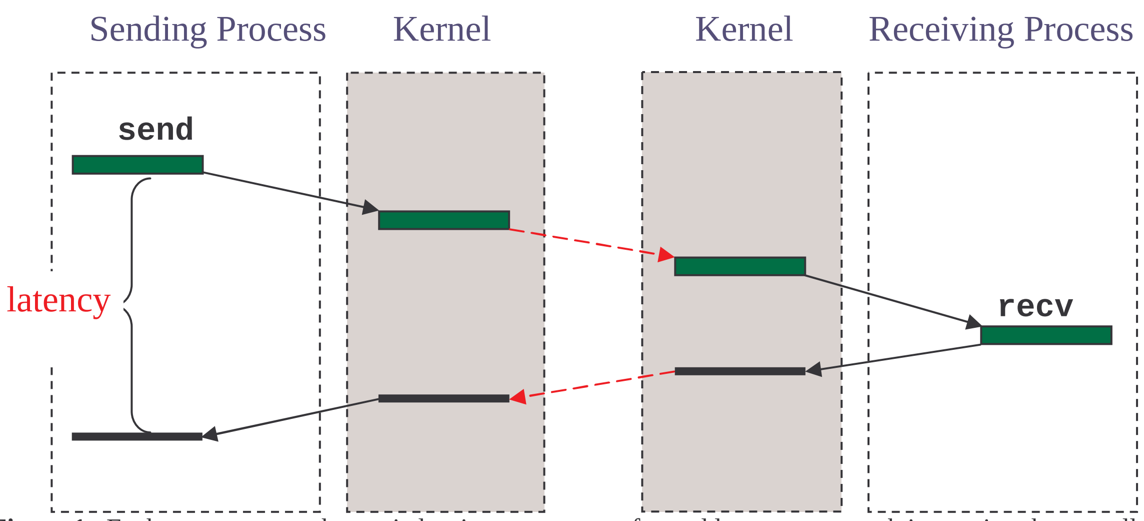

Figure 1. Each message must be copied as it moves across four address spaces, each increasing the overall latency.

Thus, to allow users to hide some of this latency, the interface provides different versions that expose some of these details. For example, non-blocking versions of the routines allow a process to perform some other independent work while it waits for a message to be transmitted. Such overlapping of communication and computation is analogous to the split-phase barrier that we saw in Chapter 6, so it improves performance at the expense of added program complexity.

By default, MPI also buffers messages in kernel space. Buffering is needed because multiple messages may arrive at a process that can only perform one receive operation at a time. By buffering the messages in kernel space, the system can ensure that all messages will eventually be delivered. Besides the added use of memory space, the drawback to kernel buffering is that messages are copied when placed in the buffer and copied again when delivered to the recipient. This extra copying increases communication latency in cases when the message could be delivered directly.

To give programmers the potential to improve performance, MPI provides a non-buffered version of the send operation to further reduce copying and reduce memory utilization.

> `MPI_Ssend()`—blocks if the destination buffer is available and the receiving process has started to receive this message.
>
> `MPI_Rsend()`—assumes that the receiving process is synchronized with the sending process, so no buffering or handshaking is required. Exploiting Rsend is very tricky and error prone.
>
> `MPI_Bsend()`—allows the programmer to specify a user-space buffer to which the message will be copied, allowing the send to return as soon as the message has been copied to this buffer.

While these more sophisticated versions of send and receive can improve performance, they can hurt *performance portability*. As machine characteristics change, the tradeoffs among the various versions also change. Moreover, the use of some of these routines, particularly Rsend, can severely complicate the program text.

Overlapping Synchronization with Computation

As we mentioned in Chapter 4, it is often useful to overlap long-latency operations with independent computation. In the same way that we can implement split-phase operations to hide the latency of barrier synchronization (see Chapter 6), we can use the `MPI_Isend()` and `MPI_Irecv()` operations to hide communication latency.

[SHOW THE CODE FOR THE FOLLOWING EXAMPLE]

To see a concrete example of how split-phase operations can help, consider a 2D successive relaxation program, which is often used—often in 3D form—to solve systems of differential equations, such as the Navier-Stokes equations for fluid flow. This computation starts with an array of n+2 values: n interior values and 2 boundary values. At each iteration, it replaces each interior value with the average of its 2 neighbor values,

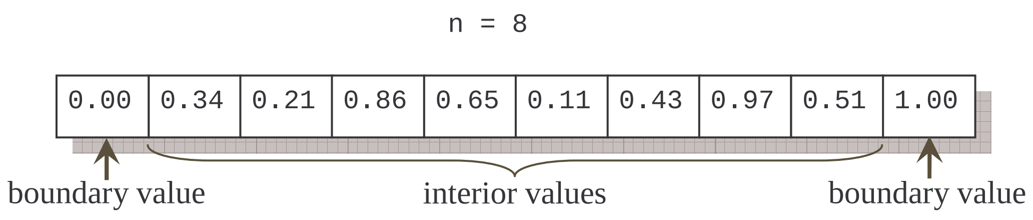

Figure 7-2. A 2D relaxation replaces on each iteration all interior values by the average their two nearest neighbors.

Critique

Without a doubt the greatest strength of MPI and other message passing libraries is their universality. The ability to transfer a block of memory from one processor to another is fundamental to parallel computers, and message passing libraries give programmers access to that facility. It must exist on all parallel machines, so these libraries can in principle—and do in fact—run on any parallel machine. This universality is an essential component of the libraries' popularity.

With similar certainty the greatest weakness of MPI and other message passing facilities is their low level of abstraction. Parallel programming is difficult and benefits greatly from computational abstractions as described in earlier chapters. MPI provides only rudimentary support for a few of these. For example, the basic reduce operation to combine all elements of an array is supported only to the extent that a single summary item on each processor can be combined by one of a small set of basic operators; the task of local combining—that is, elevation of the reduce concept to an entire distributed data structure—is left to each programmer.

Curiously, the strength—universality—and the weakness—low level—combine to enable programmers to write fast, reasonably portable programs. And that property has assured message passing libraries success.

Exercises

Chapter 8: The Z-level Programming Language

As we have seen, threads and message passing are approaches that support parallel programming through libraries that extend a standard sequential programming language. The advantages of such an approach are significant: Programmers are already familiar with the base language, so learning only involves learning the library facilities; library systems can be produced quickly because they only involve wrapping the machine's features in a standard form; and libraries are versatile, because programmers have latitude to choose a base language. The main problem with libraries is that they provide few parallel abstractions, implying that all of the parallel mechanisms programmers need to solve a problem must be handmade with a custom implementation. For example, to use the scan abstraction requires that the parallel prefix algorithm be manually constructed. The absence of abstractions places a significant burden on programmers. High-level parallel languages contain statement forms for parallel abstractions, and assign the implementation details to the compiler.

In this chapter we present ZPL, a high-level parallel programming language. No general-purpose high-level parallel language is yet in wide use, including ZPL, despite years of research and development. There are a variety of explanations for this odd situation. They range from the fact that current languages are (in some ways) incomplete due to deep unsolved technical problems, to the psychological behavioral of programmers when deciding to adopt a new language. (Of course, even with deficiencies they could be used, but with greater programming effort.) Despite not yet being in wide use, learning a high-level language can teach us how expressive a high-level parallel language can be. We can think in the language even if we do not program in it. ZPL is a good example because it is an effective tool for algorithm design and high-level program structuring; its abstractions produce fast and portable parallel programs. Thus, "thinking in ZPL" produces better programs in whatever language one programs.

ZPL is an implicitly parallel programming language, meaning that the compiler generates all parallel threads, it inserts all necessary communication calls, it attends to synchronization, it protects against data races, etc. Programmers only specify the logic of the computation; the compiler does the rest. What makes it possible for the compiler to do all of the "heavy lifting" is that the language provides expression- and statement-forms for common parallel abstractions. Programmers write plus scan of an array in a few characters (`+||A`) and the compiler implements it. Thus, ZPL is a rich source of succinct notation for parallel abstractions, and that alone justifies our study of it.

Get The Software. The ZPL compiler and documentation are available at `http://www.cs.wasington.edu/research/zpl/`. The compiler runs under Unix/Linux systems and easily targets to new parallel machines.

Basic Concepts of ZPL

ZPL is an array language, meaning that entire arrays are operated upon as a unit. Thus, to increment all elements of an array `A`, write

```
A := A + 1;
```

or equivalently,

```
A += 1;
```

Notice that the assignment operation in ZPL is `:=` rather than simply `=`. The primitive updates to the array are (logically) performed in parallel.

Regions

Though it is common to want to modify all elements of an array, it is equally common to want to limit the modification to particular elements. To control which elements are to be referenced in an array expression, we require that all array operations must be executed in the context of a region, as in

```
[1..n] A := A + 1;
```

The bracketed text is a *region*. Regions specify a set of indices, and they are a key idea in ZPL. Assuming `A` is declared to have *n* elements, indexed 1 to *n*, then the statement references them all. The statement

```
[1..n/2] A := A + 1;
```

references only the first half of `A`'s elements.

Region Form. Regions take several forms as described below. In the common *index range* form shown, the lower limit, *ll*, and the upper limit, *ul*, can have any value such that *ll* ≤ *ul*; the bounds are separated by double dots for each dimension; dimensions are separated by commas. Thus, we write

`[-100..100]`	*A linear array of 201 indices with balanced index range*
`[1..8, 1..8]`	*A square array for a chessboard*
`[1..4, 1..4, 1]`	*A plane in 3D, equivalent to* `[1..4, 1..4, 1..1]`

When, as shown in this last case, an index range is a single value, it is called a *collapsed dimension*.

The limits *ll* and *ul* do not have to be constants; they can also be integer expressions, as in

```
[min/2..2*max]
```

where `min` and `max` are scalar, that is non-array, variables.

byte types	2-byte types	4-byte types	8-byte types	16-byte types
`boolean`				
`sbyte`	`shortint`	`integer`	`longint`	
`ubyte`	`ushortint`	`uinteger`	`ulongint`	
		`float`	`double`	`quad`
		`complex`	`dcomplex`	`qcomplex`

The prefix ‘`u`’ indicates that the representation is unsigned, giving it an additional bit of precision. The `quad` type is available only if it is available in C on the target architecture; otherwise it defaults to `double`. A complex type using *k* bytes, uses *k* bytes for the real and *k* bytes for the imaginary parts of the number.

Code Spec 8.1. Primitive data types available in ZPL.

Regions In Declarations. In addition to their use in specifying which elements of an array participate in a computation, regions are also used to declare arrays with the `var` statement. For example, three $m \times n$ arrays, `B`, `C` and `D`, are declared by

```
var B, C, D : [1..m,1..n] float;
```

These are floating point arrays. ZPL supports a variety of types as shown in Code Spec 8.1.

Naming Regions. It quickly becomes cumbersome to write explicitly the same regions, so they are usually named. To name a region, use the `region` declaration, as in

```
region R = [1..m, 1..n];
```

Thereafter, the region’s name can be used wherever regions can appear, such as declarations

```
var B, C, D : [R] float;
```

and statement control

```
[R] B := 2*C + D;
```

Notice that the brackets are required around the region name.

Region Scoping. Finally, regions are *scoped*. That is, the region that applies to a statement is the region specification on the closest enclosing statement. So, for example, in the looping statement

```
     [R] repeat
           stmt 1;
           stmt 2;
    [1..n] stmt 3;
           stmt 4;
         until condition;
```

the region `[R]` prefixes the `repeat` which encloses the statements, and so, applies to the first two statements and the last. Further, a different region controls `stmt 3`, because its region is closer and takes precedence. It is common for programs to operate over many arrays with the same shape making it typical for a program to declare a single region and to prefix the main program block with that region, which causes all statements to operate on arrays of that shape, unless otherwise specified explicitly.

ZPL Control-Flow Statements

`if` ***logical-expression*** `then` ***statements*** **{**`else` ***statements*** **}** `end;`
`for` ***var*** `:=` ***low*** `to` ***high*** **{**`by` ***step*** **}** `do` ***statements*** `end;`
`while` ***logical-expression*** `do` ***statements*** `end;`
`repeat` ***statements*** `until` ***logical-expression***`;`
`return` {***expression***}`;`
`begin` ***statements*** `end;`
Text in brackets is optional; text in italics must be replaced by program constructs of the indicted kind.

Code Spec 8.2. Syntax of control statements in ZPL.

Array Computation

Arrays in ZPL are generally combined element-wise using standard operators. Code Spec 8.3 lists ZPL's primitive operators and operator-assignments. For example, the statement (from a program we discuss below)

```
     [R] TW := (TW & NN = 2) | (NN = 3);
```

operates just as it would for simple scalar values, except that it is applied to corresponding array elements for all indices in R. It is as if many statements of the form

```
     TW[1,1] := (TW[1,1] & NN[1,1] = 2) | (NN[1,1] = 3)
     TW[1,2] := (TW[1,2] & NN[1,2] = 2) | (NN[1,2] = 3)
     TW[1,3] := (TW[1,3] & NN[1,3] = 2) | (NN[1,3] = 3)
                 ...
     TW[m,n] := (TW[m,n] & NN[m,n] = 2) | (NN[m,n] = 3)
```

are all executed simultaneously. In actuality, the compiler generates code equivalent to

```
     for (i = lo1-1; i < hi1-1; i++)
     {
        for (j = lo2-1; j < hi2-1; j++)
        {
           TW[i,j] = (TW[i,j] && NN[i,j] == 2) || (NN[i,j] == 3);
```

```
          }
      }
```

We discuss the parallel execution of this code below.

Datatype	Operators
Numeric	`+` (unary), `-` (unary), `+`, `-`, `*`, `/`, `^`, `%` (modulus, `a%b` is a mod b)
Logical	`!`, &, `|`
Relational	=, `!=`, <, >, <=, >=
Bit-wise	`bnot(a)`, `band(a,b)`, `bor(a,b)`, `bxor(a,b)`, `bsl(s,a)` (shift `a`'s bits `s` places left, fill with `0`s), `bsr(s,a)` (shift `a`'s bits right `s` places, fill with `0`s)

Exponentiation (`^`) is optimized to multiplication for small powers, e.g. 2, but generally compiles to a call on C's `pow()` function.

The operator assignments recognized are: `+=, -=, *=, /=, %=, &=, |=`

Code Spec 8.3. ZPL's primitive operators and operator-assignments.

Controlling Array Element Reference

Regions control the indices that participate in a computation. The specified indices must exist in all arrays of the statement, but the arrays need not be the same shape. For example, from above, `B` is $m \times n$; suppose `E` is $m \times m$, $m < n$; then

```
        [1..m, 1..m] E := 1/B;
```

references all of `E`, but only the $m \times m$ subarray of `B`; the other elements are unaffected by the computation. So, it is an obvious condition: *all indices specified in the region must exist for all arrays* in the statement.

Of course it is possible to change individual elements of an array by simply referencing a degenerate region, as in

```
        [x,y]  D := sqrt(2);
```

which sets the single element `D[x,y]` to 1.414….

One thing that we are not allowed to do is to combine arrays of different rank. That is, if `A` is 1-dimensional array declared to have indices `[1..n]`, then it cannot be added to the first row of the 2-dimensional array `C`,

```
        [1, 1..n] C := C + A;
```
ILLEGAL for the given conditions.

because `A` and `C` are declared to have different ranks. (This computation is possible and easy using the flooding operator described below.) The purpose of the "like rank" rule is to maintain control of the parallel memory allocation and thus control over locality; see the section on the WYSIWYG performance model.

@-communication

Referencing corresponding elements, though common, is not the only association of elements of interest. In many computations it is essential to reference an element's neighbors. To reference neighbors, ZPL provides *directions*. Directions are a relative offset from an index position. So, for each element of A to reference the index to its left and its right, we declare

```
direction left = [-1];  right = [1];
```

The value in brackets is a vector pointing (in index space) to the element to be referenced. Directions are applied to an operand using the @-operator. Thus, the local average of the interior elements of A is computed as

```
[2..n-1] A := (A + A@left + A@right)/3;
```

In the statement, A uses the indices given by the region; A@left specifies the set of indices one less than the indices in the region, that is, 1 to *n*-2, and A@right specifies the set of indices one larger than the indices in the region, that is 3 to *n*; accordingly, the statement has the effect of replacing each element (interior to the array) with the average of itself, its left and its right neighbor. As a general rule it is possible to add the direction to the region's elements to find the referenced set of indices.

As another example, declare the eight compass directions

```
direction nw = [-1,-1]; no = [-1, 0]; ne = [-1, 1];
          we = [ 0,-1];               ea = [ 0, 1];
          sw = [ 1,-1]; so = [ 1, 0]; se = [ 1, 1];
```

which allow the eight nearest neighbors of an element to be referenced. If TW is a 2-dimensional array of 0s and 1s, then the expression

```
TW@nw + TW@no + TW@ne +
TW@we +         TW@ea +
TW@sw + TW@so + TW@se
```

computes an array whose value in each position is the count of its neighbors in TW that are 1. This computation will be useful in the Life program shown below. But first we need one more concept to explain that program.

Reduce

Recall from Chapter 4 that reduce is the operation of combining the elements of an array using a primitive operator; we say we have "reduced the array to a single value using the operator." ZPL's reduce operation is given by the form *op* `<< A`

where *op* is one of the primitive associative and commutative operators: `+`, `*`, `&`, `|`, `max`, `min`. To add up the interior elements of `A`, we write

```
[2..n-1] total := + << A;
```

and note that like all operations on arrays in ZPL, it is essential to specify a region.

Reduce can be applied to arrays of any rank. So, to find the largest element in `B`, write

```
[R] biggest := max << B;
```

It is not necessary to store the scalar result. So,

```
[R] span := (max << B) - (min << B) + 1;
```

The reduce operation is implemented using the parallel prefix algorithm discussed in Chapter 4. ZPL also has a partial scan operator using the syntax `+||A`.

Notation: As a convention ZPL programmers capitalize the names of arrays and regions to emphasize that the reference to many elements, and use lower case for everything else.

Life, An Example

To illustrate the concepts introduced so far, consider Conway's game of Life. It is trivial computation, often used as a screensaver, which makes a clean, simple example.

The Problem. Recall that the game simulates generations of organisms. The initial configuration is generation 0. The rules are

a) An organism lives to generation *i*+1 if it has at least 2 neighbor organisms and no more than 3;
b) An organism is born into generation *i*+1 if its position is empty and it has exactly 3 neighbor organisms in generation *i*;
c) All other organisms die before generation *i*+1.

The rules reduce to a condition that says an organism exists in generation *i*+1 either because it exists in generation *i* and has exactly 2 neighbors, or its position (whether occupied by an existing organism or not) has exactly 3 neighbors in generation *i*.

The Solution. We solve the problem in a rectangular world by the array `TW`, *the world*. Organisms are represented as 1-bits. To know how many neighbors exist for a position, we add up their 8-nearest neighbors, as discussed above, into a variable `NN`, *neighbor number*. We use the logic shown in Figure 8.1.

```
program Life;
config const n : integer = 50;

region
  R    = [1..n,   1..n  ];
  BigR = [0..n+1, 0..n+1];

var
  TW : [BigR] boolean = 0; -- The World
  NN : [R]    integer;     -- Number of Neighbors

direction
  nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];
  we = [ 0, -1];               ea = [ 0, 1];
  sw = [ 1, -1]; so = [ 1, 0]; se = [ 1, 1];

procedure Life();
begin
  -- Initialize the world
[R] repeat
      NN := TW@nw + TW@no + TW@ne +
            TW@we +         TW@ea +
            TW@sw + TW@so + TW@se;
      TW := (TW & NN = 2) | (NN = 3);
    until !(|<< TW);
end;
```

Figure 8.1. ZPL program for Conway's game, Life.

How It Works. The first half of the program is declarations, which have the following meaning:

`config const n : integer = 50` specifies the array bound, `n`, as a configuration constant, meaning that its value does not change after initially being set, and the initial setting is either the default value from the declaration, or a value specified on the command line.

`region R = [1..n,1..n]; BigR = [0..n+1,0..n+1]` declares two regions, `BigR` being larger than `R` by border elements; the boundary will be uninhabited, i.e. assigned 0s, and is required for the @-references.

`var TW:[BigR] boolean = 0; NN:[R] integer` declares the problem representation (`TW`) initialized to 0, and the intermediate count of neighbors (`NN`).

`direction nw=[-1,-1]; ...` defines the eight compass directions needed to reference the nearest neighbors.

Notice that naming the regions was mostly pedagogical because they are only used three times in the program, excluding declarations, and so could have been written explicitly. Nevertheless, we recommend naming regions.

Entry Point. ZPL requires that some procedure have the same name as the program, i.e. matching the word following `program` on the first line. That procedure is the entry point for the ZPL computation.

Code Spec 8.4. Specifying the entry procedure for ZPL

The program is a single procedure, `Life`. After the world is initialized—we assume a random configuration is created or an input file is read—the computation enters a repeat-loop. The first line

```
NN := TW@nw + TW@no + TW@ne +
      TW@we +         TW@ea +
      TW@sw + TW@so + TW@se;
```

computes the number of living neighbors for each array position by type-casting the Boolean arrays into integer arrays and adding. This line could be read, "Add the array of northwest neighbors in `TW` to the array of north neighbors in `TW` to the array of northeast neighbors in `TW` ...." That is, ZPL programmers think of such operations from the global, array viewpoint rather than the local, index viewpoint.

The next line creates the next generation by applying Conway's rules. The next generation is

```
TW := (TW & NN = 2) | (NN = 3);
```

the logical-or of two arrays, the array of organisms with exactly two neighbors, and the array of positions with exactly three neighbors.

When an iteration of the loop is complete, the termination test checks to see if there are still living organisms, and if not it exits. The termination condition

```
!(|<< TW);
```

computes an or-reduce over the world array, `TW`, which is 0 if no organisms exist, and negates the result.

Summary of Life. The Life game is simple, and the ZPL program for it is also simple, some declarations plus two lines in a loop. Notice that the one loop drives the sequence of generations only. The programmer did not write any array traversal loops or write any index expressions; the compiler took care of generating all of the code for array manipulations.

Perhaps more importantly, the compiler produces highly parallel code for the Life computation, though the programmer didn't specify any parallel constructs. The parallelism is embedded in the semantics of the operations: The two statements of the loop are fully parallel and the reduce operation uses the efficient Schwartz algorithm. (Of course, the declarations are no cost or trivial, one-time overhead.)

The price for such convenience is that we had to think of the solution as an array computation. Though different, it was not so difficult.

Manipulating Arrays Of Different Ranks

With limited exceptions ZPL requires all of the arrays in a statement or expression to have the same rank, that is, the same number of dimensions. This is the common case, a natural consequence of algorithm design. But in some situations computations produce arrays of different rank, and in other cases arrays of different ranks must be operated on together. In these cases ZPL applies two basic ideas:

- **Use the larger rank:** When arrays of two different ranks are to be used together, make all arrays the same (larger) rank. This is always possible, because a *d*-dimensional array can always be considered a *d*+1 dimensional array by picking a single index for that dimension. The lower rank array becomes a higher ranked array with a collapsed dimension. For example, to operate on arrays whose regions are `[1..n, 1..p]` and `[1..m, 1..n, 1..p]`, simply make the first region `[1, 1..n, 1..p]`. The idea applies inductively when the gap in rank is larger than 1.

- **Replicate elements:** When values of lower rank arrays are used repeatedly with elements of higher ranked arrays, logically replicate the elements of the lower ranked array so that they match element-for-element the higher ranked array. ZPL has an operator, called *flood*, that performs this operation logically.

We will see these two ideas merge in the concept of the “flood dimension” later in this section.

The reason for all of this attention to array rank is that in ZPL the region defining an array not only tells how many dimensions it has, how many elements it has, and what the indices are, it also specifies the allocation. Different rank arrays will generally have different allocations. For example, the regions `[1..64]`, `[1..8, 1..8]` and `[1..4, 1..4, 1..4]` might be allocated as shown in Figure 8.2. Such allocations dramatically affect which values are “close” to each other. Such proximity dramatically affects locality. Locality dramatically affects performance. We have more to say on this below.

The main consequence is that regions such as `[1..m]`, `[1..m, 1]` and `[1..m, 1, 1]`, though they are all conceptually a sequence of values, have different allocations. Since arrays with the same allocation exhibit better locality, and so better performance, the ZPL designers opted for the “like rank” requirement.

In this section we introduce operators that change rank, such as partial reduce, and operators that accommodate rank differences, such as flood.

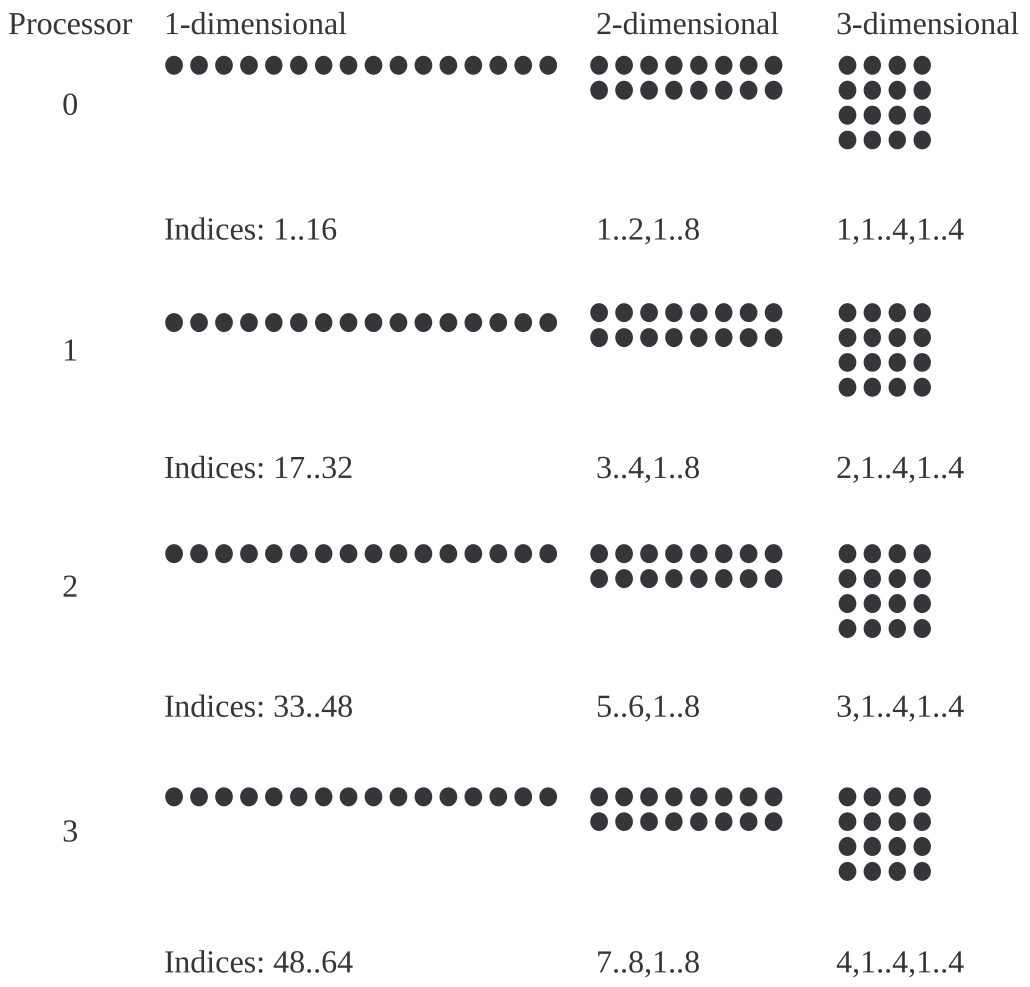

Figure 8.2. Typical default allocations for 1-, 2- and 3-dimensional regions of 64 elements each on four processors, depicted simply as dots.

Partial Reduce

A basic property of the reduce operation is that it converts an array into a scalar. That is,

```
sum := +<< A;
```

produces a single value by adding the elements of array A. If this result is seen as "reducing" *all* of the dimensions of an array, then a partial reduce can be viewed as reducing *some* of the dimensions of an array. Considering the $m \times n$ array B, we note that we could reduce the first dimension by combining the columns of values to produce a single row, or reduce the second dimension by combining the rows of values to produce a single column.

Not surprisingly, in ZPL partial reduce uses two regions *with the same rank*, one to specify the source indices and one to specify the target indices. The source region, specified with the operand, defines which operand indices are input to the reduce; the target region, usually on the statement, defines the indices of the result. So, to partially reduce `B` along the first dimension using add, we write

```
[1, 1..n] C := +<< [1..m, 1..n] B;
```

where the "operand region," `[1..m,1..n]` specifies which indices of `B` are to participate in the reduce (source), and the statement region `[1, 1..n]` specifies the indices of the result (target). For example, for *m* = 3 and *n* = 4,

```
[1, 1..4] 7 7 6 5  ⇔  +<< [1..3, 1..4] 3 1 4 1
                                       1 4 1 4
                  ≠                    3 2 1 0
```

The compiler computes the "difference" between the two regions—the first dimension "reduces" from *m* indices to one and the second dimension is unchanged—and figures out that the first dimension, the columns, are to be reduced, that is, added.

To reduce `B` in the second dimension using multiply, we write

```
[1..m, 1] D := *<< [1..m, 1..n] B;
```

and for *m* = 3 and *n* = 4 produce

```
[1..3, 1] 12  ⇔  *<< [1..3, 1..4]  3 1 4 1
          16                       1 4 1 4
           0                       3 2 1 0
                    ≠
```

Again, the compiler computes the "difference" between the operand region and the statement region figuring out that multiply is applied to the second dimension. It is not necessary to give the statement region explicitly; if the desired target region is already the applicable region—because the statement is executed within its scope—then the region need not be repeated.

Notice that the region on the statement is really just defining the prevailing context for the computation. The closest applicable region might be another operand region, as in the more complex operation on the $p \times m \times n$ array `F`,

```
[1,1,1..n] G := max<< [1,1..m,1..n] (min<< [1..p,1..m,1..n] F);
```

which finds the "plane" of minimum values over the first dimension, and then finds the row of maximums over the columns of the "plane." Thus, for *m* = 3, *n* = 4 and *p* = 2, an example is

```
[1,1..3,1..4] 2 1 3 1   ⇔   min<< [1..2,1..3,1..4]  3 1 4 1
              1 3 1 2                               1 4 1 4
              0 2 0 0                               3 2 1 0

                                                    2 4 3 4
                                                    1 3 2 2
                                                    0 5 0 3


    [1,1,1..4] 2 3 3 2   ⇔   max<< [1, 1..3, 1..4]  2 1 3 1
                                                   1 3 1 2
                                                   0 2 0 0
```

For the `min` reduction the compiler compares its operand region with the operand region of the `max` reduction to determine that the first dimension is reduced; for the `max` reduction, the compiler compares the operand region with the statement region. This idea generalizes.

Flooding

If it is possible to reduce an array dimension, then it ought to be possible to expand an array dimension. ZPL has an operation that expands in (one or more) dimensions by replicating. It is called flooding (`>>`), and it is the opposite of partial reduction but with the similar syntax. So,

```
[1..m, 1..n] B := >> [1, 1..n] C;
```

fills the array `B` with copies of the first row of `C`. For *m* = 3 and *n* = 4, an explicit example is

```
 [1..3, 1..4] 7 7 6 5   ⇔   >> [1, 1..4] 7 7 6 5
              7 7 6 5
              7 7 6 5
                           ≠
```

Like partial reduce the compiler figures out which dimensions to flood by comparing the two regions, and noting the differences.

Of course, flooding can apply to any dimension, including the second dimension, as in

```
[1..m, 1..n] C := >> [1..m,1] D;
```

which can be illustrated by an example, for *m* = 3 and *n* = 4,

```
[1..3, 1..4] 12  12  12   ⇔   >> [1..3,1] 12
             16  16  16                  16
              0   0   0                   0
```

In addition, it is possible to flood only a portion of a dimension with replicated values.

The Flooding Principle

What is the point of flooding; why copy values? Flooding is used when arrays of different ranks must be operated on together. Suppose, for example, that every column of a matrix is to be scaled by, say, its column two. The matrix is 2-dimensional and the column is in concept 1-dimensional. But, because only one region applies to a statement, ZPL requires that the two arrays match in rank. So, we replicate column two using flooding, making it a logical 2-dimensional array, and divide (element-wise) by the result. Specifically, we write

```
[1..m,1..n] B := B / (>> [1..m, 2] B);
```

The expression in parentheses is an $m \times n$ array composed of n copies of column two, and because the division is performed element-wise, the result is that every column is scaled by column two. Specifically, let $m = 3$ and $n = 4$, then an example of the principle would be

```
[1..3, 1..4] 3.00  1.00  4.00  1.00   ⇔  3  1  4  1  /  1  1  1  1
             0.25  1.00  0.25  1.00      1  4  1  4     4  4  4  4
             1.50  1.00  0.50  0.00      3  2  1  0     2  2  2  2
```

The values are only *logically* replicated; that is, the compiler does not actually make copies of the values. The benefit of this approach is to allow better locality in the computation, and to manage data transfers more efficiently.

Data Manipulation, An Example

Imagine a dataset `D` of shape `[1..m, 0..n]` containing rows of n observations, say cups of coffee consumed in a day, for m subjects. To record summary data, the array has been given an additional 0th column. Consider some illustrative computations on this data.

The most coffee consumed on any day by any subject is the `max` reduce over the data portion of the entire array

```
 [1..m,1..n] most := max<< D; -- Compute top score
```

The variable `most` is a scalar.

The maximum for each subject is the partial reduction across the rows, which we store in the 0th column

```
[1..m, 0] D := max<< [1..m, 1..n] D;  -- Record individual maxima
```

The computation produced a column of values.

The test for any non-coffee drinkers is simply a check for a 0 in the summary column and an OR-reduce (over that column only) to accumulate the result, as in

```
[1..m, 0] tFans := |<<(D = 0); -- Does anyone not like coffee?
```

The variable `tFans` is a scalar. (Of course, a simple AND-reduce `&<<D` would also work, but it may be less clear.)

In the case where everyone is a coffee drinker, we can scale everyone's coffee habit in the range (0,1] relative to their biggest day by flooding the first column across the array and dividing the result into the data array,

```
if !tFans then
     [1..m,1..n] D := D / (>> [1..m, 0] D);  -- Scale by maximum
end;
```

Finally, we can determine the percentage of days of the study that each person achieved his or her maximum. We begin by comparing the whole data array to 1, then partially reducing each row using addition, which produces a count of max days for each person, and then dividing the results by `n`.

```
[1..m, 0] D := 100 * (+<< [1..m,1..n](D = 1)) / n; --Pct days@max
```

Though some programmers might find it natural to use column 0 as a summary column, others would prefer to declare the dataset by its proper dimensions `[1..m, 1..n]` and use a separate array `Sum [1..m, 1]` to store summary results. Of course, we require `Sum` to be 2-dimensional, because it will be used in expressions involving `D`. With this approach the forgoing computations become

```
[1..m,1..n]  most    := max<< D;
  [1..m, 1]  Sum     := max<< [1..m, 1..n] D;
  [1..m, 1]  tFans   := |<<(Sum = 0);
             if !tFans then
[1..m,1..n]        D := D / (>> [1..m, 1] Sum);
             end;
  [1..m, 1]  Sum     := 100 * (+<< [1..m,1..n](D = 1)) / n;
```

By such computations ZPL programmers perform routine data manipulation, switching back and forth among various data sizes but remaining within a given rank.

Flood Regions

The use of the `Sum` variable in this last example illustrates a curiosity with the way we have used ZPL so far. Though it made sense in our first solution, perhaps, to place the summary column in the 0th position, why do we specify the array `Sum` to have its second index be 1? Could it be 0 or 9 or `n`? Yes. The actual consequences of this decision for memory allocation will be explained below, but our point now is that the choice will generally be arbitrary. To emphasize the arbitrariness, notice that one of the operations on `Sum` is to flood it, that is, replicate it in all column positions.

ZPL has the concept of a flood dimension, denoted by an asterisk (*) in the region expression. The flood dimension is effectively a "don't care" for a (collapsed) index. So, the best way to define the summary array from the last example would be

```
var Sum : [1..m, *] float;
```

which specifies that the data is flooded in the second dimension. The final four statements from the earlier example now become

```
  [1..m, *]  Sum     := max<< [1..m, 1..n] D;
  [1..m, *]  tFans   := |<<(Sum = 0);
             if !tFans then
[1..m,1..n]        D := D / Sum);
             end;
  [1..m, *]  Sum     := 100 * (+<< [1..m,1..n](D = 1)) / n;
```

There is no need to flood `Sum` in the third line because the data is already flooded by the properties of flood dimensions. This means that there are values to correspond element-wise with the *n* elements of `D`'s second dimension. How many elements does `Sum` have in its second dimension? Any number needed with any needed indices. Graphically, the values of `Sum` can be visualized as

$$\begin{array}{l}
\ldots, v_1, v_1, v_1, v_1, \ldots \\
\ldots, v_2, v_2, v_2, v_2, \ldots \\
\ldots, v_3, v_3, v_3, v_3, \ldots \\
\ldots, v_4, v_4, v_4, v_4, \ldots \\
\qquad\ldots \\
\ldots, v_m, v_m, v_m, v_m, \ldots
\end{array}$$

so they match arrays of any size in the second dimension. Like all flood dimensions, however, `Sum`'s second dimension is only logical.

Flood dimensions are sensible based on the principle that programmers should never be asked to specify more than they mean. But there is another important reason. Flood dimensions enable the compiler to use very efficient data representations—data replication is avoided—and very efficient communication protocols—multicast is often possible. As a result, it is always better to select a flood dimension than it is to make an arbitrary choice of a collapsed index value. (Sometimes a choice is appropriate, as when controlling allocation, however.)

Matrix Multiplication

To apply the ideas of this section consider computing the product of two dense matrices, *A*, which is $m \times n$, and *B*, which is $n \times p$ to produce $C = AB$. This computation is often programmed in sequential programming languages as a triply nested loop

```
for (i = 0; i < m; i++) {
{
    for (j = 0; j < p; j++)
```

```
          {
              C[i,j] = 0;
              for (k = 0; k < n; k++)
              {
                  C[i,j] += A[i,k]*B[k,j];
              }
          }
      }
```

computing the *dot product* for each element, in which the `i`th row of `A` is multiplied times the `j`th column of `B` and reduced to produce the `C[i,j]`.

We present this sequential solution only to be precise. It is not the right way to think about a parallel matrix product. Indeed, van de Geijn and Watts argued that computing the dot products separately, i.e. a row of A times column of B, is exactly backwards. In their SUMMA (Scalable Universal Matrix Multiplication Algorithm) approach they bring the initialization and the *k*-loop to the outside, effectively computing all of the *k*th terms of all of the dot-products at once. This contrary way of thinking produces an extremely efficient matrix multiplication. It is also the easiest ZPL matrix multiplication because it exploits flooding.

To see the key idea of SUMMA and why flooding is so fundamental to it, notice that in the computation $C = AB$ for 3x3 matrices A and B, the definitions of the first two columns of the result are

$$\begin{array}{ll}
C_{1,1} = A_{1,1}xB_{1,1} + A_{1,2}xB_{2,1} + A_{1,3}xB_{3,1} & C_{1,2} = A_{1,1}xB_{1,2} + A_{1,2}xB_{2,2} + A_{1,3}xB_{3,2} \ldots \\
C_{2,1} = A_{2,1}xB_{1,1} + A_{2,2}xB_{2,1} + A_{2,3}xB_{3,1} & C_{2,2} = A_{2,1}xB_{1,2} + A_{2,2}xB_{2,2} + A_{2,3}xB_{3,2} \ldots \\
C_{3,1} = A_{3,1}xB_{1,1} + A_{3,2}xB_{2,1} + A_{3,3}xB_{3,1} & C_{3,2} = A_{3,1}xB_{1,2} + A_{3,2}xB_{2,2} + A_{3,3}xB_{3,2} \ldots
\end{array}$$

Notice that the first term of all of these equations can be computed by replicating the first *column* of A across a 3x3 array, and replicating the first *row* of B down a 3x3 array, that is, flooding A's first column and B's first row, and then multiplying corresponding elements; the second term results from replicating A's second column and B's second row and multiplying, and similarly for the third term.

The ZPL matrix product code is shown in Figure 8.3. The program begins with the obvious variable declarations. The `Col` variable will be used to flood columns of `A`, and `Row` will be used to flood rows of `B`. In the body of the procedure, the entire computation is executed in the context of the result array `C`'s dimensions. The result array is initialized to 0, and the computation enters the `k`-loop that processes through the `n` terms of the dot product.

In the body of the loop the next column of `A` is flooded across `Col` and the next row of `B` is flooded down `Row`. In the final statement of the loop the two flooded arrays are multiplied element-wise and accumulated into the result array, `C`. Then the next term of the dot-product is considered.

```
var A   : [1..m, 1..n] double;
    B   : [1..n, 1..p] double;
    C   : [1..m, 1..p] double;
    Col : [1..m, *]    double;
    Row : [*, 1..p]    double;
    k   :              integer;
           ...

procedure MM();
[1..m, 1..p] begin
               C := 0;
               for k = 1 to n do
     [1..m, *]    Col := >> [1..m, k] A;
     [*, 1..p]    Row := >> [k, 1..p] B;
                  C += Col*Row;
               end;
             end;
```

Figure 8.3. The SUMMA matrix multiplication algorithm in ZPL.

Notice that the use of the temporary arrays `Col` and `Row` was actually unnecessary. The procedure could have been written as,

```
procedure MM();
[1..m, 1..p] begin
               C := 0;
               for k = 1 to n do
                    C += (>> [1..m, k] A ) * (>> [k, 1..p] B);
               end;
             end;
```

using expression floods. In fact, the compiler will generate temporaries for the expression floods anyway corresponding to `Col` and `Row`, but it saves the programmer a few lines of typing. The SUMMA is not only an easy matrix multiplication program to write, it is extremely fast to run.

The constraints on partial reduce and flood are summarized in Code Spec 8.5.

Partial Reduce (<<) and Flood (>>). These two operations require two regions, a source region (included as a operand) and a target region (usually on the statement), as in

```
        [1, 1..3]     ...  +<< [1..3, 1..5] A ...  // reduction
        [1..3,1..5]   ...   >> [1, 1..3]    A ...  // flood
```

For each dimension in the two regions the index ranges are either identical or one is a collapsed dimension (singleton value). For partial reduction (collapsed dimension(s) on target region), the elements of the operand are combined to collapse the dimension; for flood (collapsed dimension(s) on the source region), the element is replicated to flood the index range of the dimension.

Code Spec 8.5. Requirements of ZPL's partial reduce and flood.

Reordering Data With Remap

ZPL emphasizes computing on data that is local, but often data has to be moved around to become local. The remap performs arbitrary restructuring of data, including changing its rank. Before learning about remap, we must introduce the Index*i* arrays.

Index*i*

ZPL provides compiler-generated constant arrays of indices denoted by the form `Index` + <*dimension number*>, as in `Index1`, `Index2`, `Index3`, etc. The arrays, which are only logical, contain the index values for the indicated dimension as specified by the region. So, for example

```
[1..3,1..3]  ... Index1 ...  ⇔ 1  1  1
                                2  2  2
                                3  3  3
```

is an array of the first dimension indices, and

```
[1..3,1..3]  ... Index2 ...  ⇔ 1  2  3
                                1  2  3
                                1  2  3
```

is an array of second dimension indices. The only constraint on the use of `Index`*i* arrays is that the statement's region have an *i*th dimension.

The constant Index*i* arrays are used frequently in ZPL programs, as in

```
[1..n,1..n]  Diag := Index1 = Index2;
```

for constructing an array with 1s down the diagonal, and

```
[1..n,1..n]  RMO := n*(Index1-1) + Index2;
```

for computing the row-major order index of elements of a 2D array. They are also used frequently in remap.

Remap

The remap operator is denoted by the hash symbol (#) and has two forms, known as *gather* and *scatter*. Both forms take an argument in brackets, the *remap array*(*s*), `A#[P]`, that specifies the indices used to produce the result. One remap array is required for each dimension.

1-Dimensional Case. For example, suppose `A` and `P` are declared over the region `[1..n]`, and that for *n*=7, the values are

```
A ⇔ d d e e o r r
P ⇔ 5 6 1 3 7 4 2
```

then `A#[P]` has the value

o r d e r e d ⇔ *d d e e o r r* # [*5 6 1 3 7 4 2*]

The result can be found by indexing the operand array with the remap array; so, for example, the first element of the remap array is 5, so the fifth element of the operand, o, is selected for the first position.

It is common to use expressions based on the `Index`*i* values. For example, `A#[8-Index1]`, which is the descending index values for the first (and only) dimension of this array,

r r o e e d d ⇔ *d d e e o r r* # [*7 6 5 4 3 2 1*]

reverses the operand. To spell *ordered* backwards, write `A # [P # [8-Index1]]`.

Gather and Scatter. The form of remap shown so far is the *gather* form, which is used on the right-hand side of assignment statements. It is also possible to write remap on the left-hand side of an assignment statement, as in

```
A#[P] := A
```

to get the *scatter* semantics. In this case the right-hand side values are produced and assigned to the variable on the left-hand side according to the indices of the remap array. For the `A` and `P` data from above, the statement `A[P] := A` results in

e r e r d d o ⇔ *d d e e o r r* # [*5 6 1 3 7 4 2*]:= *d d e e o r r*

proving, if it was necessary, that gather and scatter are different. (The terms come from the fact that *gather* picks the values from operand positions using the indices and *scatter* puts the values into result positions using the indices.)

Repeats in Remap Arrays. The values in the remap arrays do not have to be unique, though that is the most common case. For example, the gather `A#[`*1 1 1 1 1 1 1*`]` is

d d d d d d d ⇔ *d d e e o r r* # [*1 1 1 1 1 1 1*]

is a cumbersome (and expensive) way to flood the first element of `A`. For scatter the issue is more curious. Because scatter assigns values, an index value appearing multiple times in the remap array can result in different orders of assignments, resulting in unpredictable results. So, the scatter form `A # [` *1 1 1 1 1 1* `]:= A`

? d e e o r r ⇔ *d d e e o r r* # [*1 1 1 1 1 1 1*]

will result in one of `d`, `e`, `o` or `r` being assigned to the first index position, but which is undefined. Different (parallel) executions will produce different results.

Higher Dimensions. Higher dimensions require multiple remap arrays in the brackets, `B # [C, D]`, the array in the ith dimension specifying the index values for that dimension. That is, the elements of `B` are reordered using `C` as the source of all of the first indices and `D` as the source of all of the second indices. For example,

```
[1..n, 1..m] Btranspose := B # [Index2, Index1];
```

is a standard idiom for computing the transpose of an array, because the indices of the two dimensions are interchanged. For `Btranspose` declared over region `[1..n, 1..m]` then, for *m*=3 and *n*=2 the transpose is illustrated by

$$\begin{array}{ccc} a & c & e \\ b & d & f \end{array} \Leftrightarrow \begin{array}{cc} a & b \\ c & d \\ e & f \end{array} \# \left[\begin{array}{ccc} 1 & 2 & 3 \\ 1 & 2 & 3 \end{array},\ \begin{array}{ccc} 1 & 1 & 1 \\ 2 & 2 & 2 \end{array}\right]$$

The operation of this gather is clear: Item *i*, *j* in the result comes from the $\mathtt{C}_{i,j}$, $\mathtt{D}_{i,j}$ position of the operand.

Ordering Example

Remap is used regularly in ZPL. An excellent example of its use is to move rows around so they are in order by some criterion. Illustrating such a computation is the task of this section.

Recall the “coffee drinkers” data array from the Data Manipulation Section. The array was defined over the region `[1..m, 0..n]`, recording the number of cups of coffee consumed by *m* people over *n* days; the first column is used for summary statistics. For example, we know that we can compute the average coffee consumption of the participants by

```
[1..m, 0] D := (+<< [1..m,1..n] D)/n;
```

Now, suppose we want to order the coffee drinkers by their average consumption, least to greatest. This means that we need to reorder the rows of the `D` array based on the value in column 0. For this, we need to compute everyone’s rank based on the 0th column.

Our strategy is to break the task into three parts

- compute the rank by using flood to make all comparisons
- reduce to get the ranking
- remap the array into the right order

For convenience, we will assume the averages are unique, but it is simple to handle duplicates.

All Comparisons Ranking Algorithm. To compare every element with every other element of a sequence, we flood a 2D array with the 0th column averages. This gives one of the operands for the comparison. To get the other operand, we transpose the array, and flood it in the other dimension. Making a comparison of the two arrays yields an array of bits.

Begin with the declarations,

```
      RepC : [1..m, *] float; -- Temp for replicated columns
      RepR : [*, 1..m] float; -- Temp for replicated rows
```

and using the averages in column 0 of `D` to flood the arrays. For `RepC` the flood is direct because the two arrays are oriented properly. For `RepR` column 0 must first be transposed to be a row before flooding,

```
      [1..m,*] RepC := >> [1..m, 0] D;            -- Replicate Ave Col
      [*,1..m] RepR := >> [*, 1..m] D#[Index2, 0];-- Repl Ave as a Row
```

Making all of the comparisons is a simple matter,

```
      [1..m, 1..m]  ... RepC >= RepR;  Make an array of bits
```

Obviously, care must be taken to chose the right relational operator. The `>=` will have the effect of setting only one bit in the row corresponding to the smallest item; all of the bits will be set in the row containing the maximum item.

All Comparisons Ranking Algorithm. To find the rank of the items, simply add up the 1s in each row using a partial reduce. This will produce a column of results, but because we will be using the result in a remap, we don’t want just a column; we want an array flooded with the rank values. So, we include the declaration

```
      var Rank : [1..m, *] integer;
```

which makes the second dimension a flood dimension.

With the Rank variable set up we can perform the partial reduction followed by the flood

```
      [1..m, *] Rank := >> [1..m,*] (+<< [1..m,1..m] (RepC >= RepR));
```

producing the desired result.

Sorting With The Rank Array. Now, using the values in `Rank` we can reorder the rows of `D` using remap.

```
       [1..m, 0..n] D#[Rank, Index2] := D;
```

which orders the rows by the value in the 0th column. The final program is shown in Figure 8.4.

```
region R = [1..m,0..n];
var      D: [R] float;
     RepR : [*, 1..m] float;
     RepC : [1..m, *] float;
     Rank : [1..m,*] integer;
           ...
  [1..m,*] RepC := >>[1..m,0] D;
  [*,1..m] RepR := >>[*,1..m] D#[Index2,0];
  [1..m,*] Rank := >>[1..m,*](+<<[1..m,1..m] (RepC >= RepR));
           D#[Rank, Index2] := D;
```

Figure 8.4. Declarations and code for reordering the rows of D according to its column 0.

This reordering operation seems complicated when considered for the first time, but it is a standard ZPL paradigm. It becomes second nature very quickly, especially once the apparatus has been set up.

Parallel Execution of ZPL

The beauty of a high level language filled with parallel abstractions is that programming is simple because the compiler does the tough work. There are no threads to keep track of, no communication calls to insert, etc. The compiler not only generates the code, it can optimize it as well. The result is a fast easy-to-write program that need only be recompiled to run well on the next parallel platform.

Although it is sensible to benefit from the compiler’s help, it is also essential to know how the program will run. That is, there are always a variety of ways to solve a given programming problem, and they often require different resources—more instructions, more data motion, more memory, etc. To write quality programs we need to know enough about how the language is implemented to know which of the competing solutions is the best. ZPL was the first parallel programming language to embed within its specification a performance model, known as the *WYSIWYG Performance Model*. By taking a few minutes to learn how the model describes the performance of the compiled code, programmers can apply the knowledge to write better programs. It’s easy.

Specifying Number of Processors

All computations begin with a standard organization, which we describe now. It is possible for programmers to assign work and data to processors differently using features not yet discussed, but any such changes begin from the standard organization.

On the command line programmers specify the number of processors and their arrangement using the `–p` option and the `–g` option. So, to run the compiled program `myProgram` on sixteen processors arranged in two rows of eight, write

```
myProgram –p16 –g2x8
```

The arrangement, which is typically expressed as a 2D grid, is key to the allocation of arrays to processors.

Assigning Regions to Processors

Since arrays inherit their indices from regions, it is not surprising that the first step in assigning arrays to processors is to assign regions to processors. The regions of a program are assigned in a consistent way, so that all regions with an index [*i*, *j*, …, *k*] have that index assigned to the same processor. To achieve this effect think of all regions being superimposed on one another so that their indices align, as shown in Figure 8.5. From this superimposition their bounding region is computed; the bounding region is the smallest region that includes all of the indices of the superimposed regions.

Region Allocation Policy. The policy of aligning the regions so that index [*i*, *j*, …, *k*] of any region is assigned to one processor guarantees that when element-wise operations are performed on arrays, such as `A + B`, all of the computation is local to the processors. There is no communication.

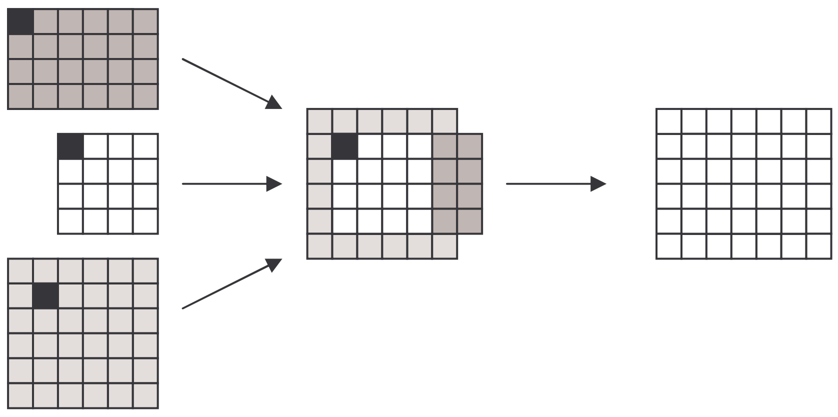

Figure 8.5. Bounding region. Regions used in the program are superimposed so that their indices align; the black square has the same index in all regions. Once aligned, the bounding region is the smallest region containing the same indices as the superimposed region.

Once computed the bounding region can be allocated based on the processor gird given on the command line with the –g option. The allocation is a block allocation using the Ceiling – Floor assignment described in Chapter 4. The indices are allocated to processors indexed in the obvious way: low index to high index in row major order. Once the bounding region has been assigned to the processors, the task is complete because the contributing regions simply make the same assignment. Thus, a `–p4 –g2x2` allocation of the region in Figure 8.5 is shown in Figure 8.6. The assignment shown illustrates that allocating regions so that all indices align can result in a slightly suboptimal allocation. The effect is generally small, and only occasionally arises.

As another example, the allocation in Figure 4.1(b) is achieved for a 16×16 ZPL region by specifying `–p16 –g16x1`.

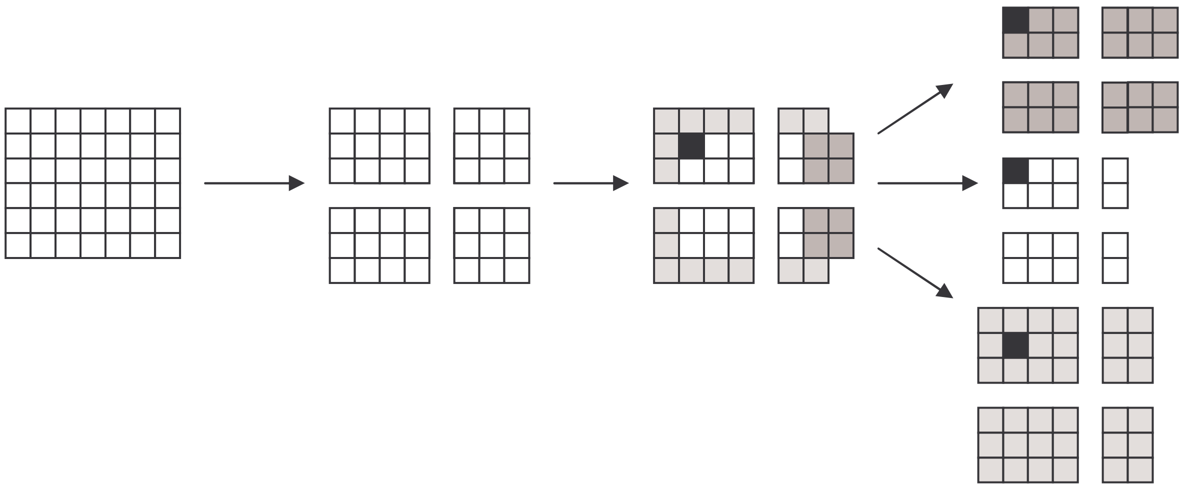

Figure 8.6. Block allocation of the bounding region. The bounding region is partitioned using the Ceiling-Floor allocation, which assigns a set of indices. The contributing regions' indices are assigned in the same way.

Array Allocation

Given the assignment of the regions' indices to the processors, arrays are trivially defined by allocating space for their elements on the processors according the where their indices are located, that is, the arrays inherit the regions' allocations. The order of allocating elements matches that of the C language, which is row-major order. Additionally, the allocations have fluff buffers where needed.

For example, the statement

```
      var B, C, D : [1..8, 1..8] float;
```

allocated to a 2×2 processor grid, would assign processor p_0 the subregion `[1..4,1..4]`, implying that those indices for arrays `B`, `C` and `D` are also allocated to it; p_1 would be allocated the subregion `[1..4, 5..8]`, etc.

Scalar Allocation

Non-array variables are redundantly allocated; that is, all processors are assigned all of the scalars. Scalar computation, such as

```
      i := i + 1;
```

is redundantly computed by each processor. On the plus side this redundancy eliminates communication (recall the random number example of Chapter 2), but on the minus side scalar computation is not a source of improved performance through parallelism.

Work Assignment

With the arrays allocated, the work assignment is easily specified: Each processor computes the values for the elements allocated to it. So, the region on the statement, which can be thought of as stating the indices to be computed in the array statement, imply which processor performs the actual computation. For the statement

```
    [1..8, 1..8]  B := C + D;
```

processor p_0 would perform the update to the subregion `[1..4, 1..4]` of `B` using the local elements of `C` and `D`. By implication when the data is well-balanced among the processors, the work for such statements will be well-balanced. Thus, the work to update this array will be distributed among the four processors evenly enough that a 4-fold speedup can be anticipated.

Performance Model

Given the previous description of how regions, arrays and work are allocated to processors by the ZPL compiler, it is easy to see how a program with only element-wise computations would perform in parallel: It exhibits essentially perfect speedup. But what about the other ZPL constructs? In fact, they are almost as easy to understand.

ZPL's performance model is based on the element-wise specification plus an overhead communication cost for those operations, such as @-references and remaps, requiring off-processor values. The communication component adds to the base cost of the work. The model rests on the idea that these two costs—basic work and communication overhead—constitute a good first approximation to the performance of any algorithm on any (CTA) platform. The model is easy to use because programmers can "see" the places in their code where they incur the added communication costs by simply noting where the operators are used. Code Spec 8.6 shows a brief summary of these costs.

Syntactic Cue	**Example**	**Parallelism (*P*)**	**Communication Cost**	**Remarks**
`[R]` *array ops*	`[R]` ... `A+B` ...	full; work/*P*		
`@` *array transl.*	... `A@east` ...		1 point-to-point	xmit "surface" only
`<<` *reduction*	... `+<<A` ...	work/*P* + log *P*	2log *P* point-to-point	fan-in/out trees
`<<` *partial red*	… `+<<[ ] A` …	work/*P* + log *P*	log *P* point-to-point	
`||` *scan*	… `+||` …	work/*P* + log *P*	2log *P* point-to-point	parallel prefix trees
`>>` *flood*	… `>> [ ] A`…		multicast in dimension	data not replicated
`#` *remap*	... `A#[I1,I2]` ...		2 all-to-all, potentially	general data reorg.

Code Spec 8.6. ZPL's performance model specifications for worst-case behavior; the actual performance is influenced by *n*, *P*, processor arrangement and compiler optimizations in addition to the physical features of the computer.

To amplify further on the cost model, consider the following operations.

- **@-translations:** The @-modifier on operands implies data transmission from values stored on adjacent processors to local fluff buffers to implement the translation. Only edge elements are transferred, so by the CTA model, these point-to-point communications will in general collectively require 1 or a constant number of λ communication delays.

- **<< reductions:** The reduce operation uses the Schwartz algorithm to combine the array values into a scalar; this is followed by a broadcast to distribute the scalar value back to all processors. Thus, the communication pattern is a combining tree followed by a broadcast tree, each of which is at most *log P* height, resulting in *λlog P* communication cost.
- **<< partial reductions:** The partial reductions follow the combining concepts of full reductions, but without the broadcast.
- **|| scan:** Scan uses the parallel prefix operation, and therefore has two traversals of a *log* height *P* tree, one up and one down, resulting in a *2λ log P* communication expense.
- **>> flood:** The flood must distribute stored values to other processors representing portions of its dimension; a multicast—a broadcast to a subset of the processors—can be used if available. (Special hardware is generally fast, but even without it broadcast can be performed by a tree, resulting in *log P* concurrent transmissions.) The primary feature of flood that bounds its communication complexity, is that if processors are assigned to more than one dimension, only a small subset of the processors will be recipients of any flood.
- **# remap:** The remap operation is ZPL's most expensive because it entails two communication cycles: one to distribute the pattern of communication (remap arrays), and one to distribute the data itself. Potentially, these are both all-to-all communications, meaning that each processor might have to communicate with every other processor. ZPL attempts to optimize remap to reduce its expense: Examples, include exploiting constant arguments as occur in transpose (`#A[Index2, Index1]`), or reusing remap arrays if they have not changed since the last remap.

Using this information, it is possible to know roughly how a statement will perform.

Applying the Performance Model: *Life*

When we wrote the Life program we focused on realizing the proper computation, but we could also know in approximate terms how the program will perform. Recall that the main computation was

```
[R] repeat
      NN := TW@nw + TW@no + TW@ne +
            TW@we +         TW@ea +
            TW@sw + TW@so + TW@se;
      TW := (TW & NN = 2) | (NN = 3);
    until !(|<< TW);
```

The loop contains essentially three computations: calculating `NN`, calculating `TW` and computing the reduction for the loop-termination test. Analyze each.

- **Calculating `NN`**. The statement involves eight @-translation followed by local computation. According to Code Spec 8.6, each @-translation requires a λ delay or so, because a CTA computer can be expected to perform many such point-to-point communications at once. So, we charge constant communication plus local computation.

- **Calculating `TW`**. This statement requires only local computation on the array elements; there is no communication charge. This is parallel computation's best case.
- **Or Reduce**. The loop termination expression requires the $2\lambda \log P$ time for the Schwartz algorithm.

Further, the default block allocation will result in reasonably balanced work, implying a factor of *P* speedup on the computation. So, asymptotically, as *n* increases, the problem continues to enjoy full speedup with O(*log P*) communication overhead; if *P* grows, the increase in communication overhead remains modest.

Applying the Performance Model: *SUMMA*

The matrix multiplication algorithm of Figure 8.3 has the text

```
[1..m, 1..p] begin
                C := 0;
                for k = 1 to n do
                     C += (>> [1..m, k] A ) * (>> [k, 1..p] B);
                end;
              end;
```

as the main part of the computation.

Ignoring the onetime initialization of array `C`, the loop has for each of the *n* iterations two flood operations and then multiply-add computations on the local elements. Again, the default block operation will result in a reasonably balanced allocation, so the multiply-add computations will be fully parallel. If we arrange the *P* processors into a $\sqrt{P} \times \sqrt{P}$ grid, then each multicast tree implementing a flood will have height $\log P/2$. Thus, the communication overhead for the two iterations can be estimated to be O(*log P*) per iteration, making it an efficient parallel matrix product solution.[1]

Summary of the Performance Model

The bound is an estimate of the worst-case time complexity of the computation based on how the model describes ZPL's execution on a CTA computer. As programmers we can depend on it as a reliable machine-independent bound. Though certain ZPL compiler optimizations can lead to better performance, the model guarantees that the program will realize at least this level of performance. For example, the compiler moves communication calls around, which can result in overlapping communication with computation. If successful in this case, the communication overhead might be entirely eliminated; but if it is not the performance is still quite satisfactory.

[1] This type of analysis can be used to compare algorithms. The original paper announcing the computation model compared SUMMA with Canon's algorithm, and found SUMMA to be better, a prediction that was confirmed by experimentation.

Summary

ZPL is a high-level array programming language with implicit parallelism. We write array computations as we might in any array language without consideration to the issues of parallelism. The compiler performs all of the parallelization, communication placement, process spawning, etc. We can be completely oblivious to parallelism in ZPL.

Nevertheless, we will pay attention to the parallelism by using ZPL's performance model to estimate how well our computation will run. Such estimates are based on the CTA and are sound for all parallel computers modeled by the CTA. ZPL is unique in allowing programmers to write well-designed parallel programs even though they do not write the implementing parallel code.

Exercises

Exercise 1: Develop a small 3 x 3 data array. By hand, work out example values for the computations in the Data Manipulation Section.

Exercise 3: Revise the row rank ordering of the coffee data to handle duplicates.

Historical Context

WYSIWYG paper, ZPL programmer's guide, remap paper.