Parallel Algorithms – Part 3

Parallel Algorithm Pseudocode Conventions for the PRAM Model

While there is no standard for parallel algorithm design, there are several conventions that are widely adopted for describing parallel algorithms. Since the PRAM model exhibits a shared memory, the pseudocode for PRAMs is an extension of the pseudocode which is utilized for sequential algorithms. The major new elements are the **in parallel** statement, the **parallelcall** statement, and the various opening syntax specifications. To avoid side effects, we will assume that all input parameters (that are not also output parameters) are passed by value. All parameters of a parallel function are assumed to be input parameters.

Opening Syntax Specification – PRAM

The opening syntax for a procedure written for a PRAM machine has the following form:

procedure <name> (<list of parameters>)

model: <model name> **with** p = (function of *n*> **processors**)

input: <description of input variables>

output: <description of output variables>

dcl: <declaration of local variables and global variables>

The opening syntax for a function written for a PRAM machine has the following form:

function <name> (<list of parameters>)

model: <model name> **with** p = (function of *n*> **processors**)

input: <description of input variables>

output: <description of returned value>

dcl: <declaration of local variables and global variables>

The in parallel Statement

In addition to the usual instructions for serial machines, parallel models have an **in parallel** statement, which allows operations to be performed on more than one component of the array simultaneously. The general form of the **in parallel** statement is:

for <Boolean expression involving array indices> **do in parallel**
    <statement 1>
    <statement 2>
        .
        .
        .
    <statement *k*>
end in parallel

The *for* clause is a Boolean expression involving array indices. Only those array elements whose indices satisfy the Boolean expression participate in the instructions contained in the body of the **in parallel** statement. These instructions can be any statements from sequential algorithms that apply to arrays. Care must be taken however, to ensure that we remain within the confines of SIMD architecture (one of our assumptions). Each of the statements within the **in parallel** statement are executed simultaneously by a set of active processors. Within the pseudocode, there is no specification for how or which of the processors is active, we simply assume that there are enough processors available to handle the task. However, when we draw the pictures to illustrate the action of the algorithm it is still common to match up processors with operations on array components.

Since we have assumed that each processor is as powerful as we need it to be, the built-in functions *odd, even, interchange sin, read, write* and so forth are assumed to be executable by each processor in parallel. An example of the **in parallel** statement is shown below:

```
for  1 ≤ i  ≤ n do in parallel
        read (A[ i ], B[ i ])
        A[ i ] := A[ i ] +  sin B[ i ]
        if A[ i ] > B[ i ] then
                call  interchange (A[ i ], B[ i ])
        endif
        write( A[ i ])
end in parallel
```

The **parallelcall** Statement

While it is possible to call a function or procedure from within an **in parallel** statement, it is sometimes more convenient, particularly when using recursion, to use a more compact syntax, hence the **parallelcall** statement. The general form of the **parallelcall** statement is:

parallelcall <name> (<parameter 1> | <parameter 2> |…| <parameter q>)

where the | symbol separates the parameters corresponding to each simultaneous call to the named procedure.

SIMD versus MIMD Considerations for the PRAM Model

Since we have assumed an SIMD approach for our PRAM model, we must take care in our algorithms not to allow two processors to perform different instructions simultaneously. Thus, **if then else** statements and **case** statements usually need to be altered to maintain the SIMD model. To illustrate this, consider Figure 1 which illustrates an example involving an **if then else** statement for a MIMD PRAM and Figure 2 which illustrates an altered form for a SIMD PRAM.

```
for 1 ≤ i ≤ n do in parallel
    if i ≤ 10 then
        A[ i ] := 2 * A[ i ]
    else
        A[ i ] := A[ i ] + 1
endif
end in parallel
```

Figure 1 – Example algorithm for MIMD PRAM model.

```
for 1 ≤ i ≤ n do in parallel
   A[ i ] := 2 * A[ i ]
end in parallel
for 11≤ i ≤ n do in parallel
   A[ i ] := A[ i ] + 1
end in parallel
```

Figure 2 – Example algorithm for SIMD PRAM model.

In the MIMD algorithm of Figure 1, ten processors would be assigned the task of executing the instruction A[i] = 2*A[i], and a different set of *n-10* processors would execute the instruction A[i] = A[i] + 1, all executions being simultaneous. However, this would not be allowed in the SIMD PRAM, since two different

instructions would be performed simultaneously. This necessitates the altered version of the algorithm illustrated in Figure 2, which utilizes two **in parallel** statements. An equivalent case arises for **case** statements executed in parallel. In the SIMD model these statements would be translated into a sequence of **in parallel** statements, one for each case clause.

Algorithms for the EREW PRAM Model

Using the syntax we have developed for the PRAM model, we'll develop algorithms in this section of the notes for the searching problem that we discussed previously.

Searching on a EREW PRAM Model

Recall that searching a list of values for the occurrence of a specific value involved assigning the value of the search element to each entry in a temporary array, *temp[1:n]*. At first glance, it might seem that the following code segment would accomplish this task:

```
for 1 ≤ i ≤ n do in parallel
   temp[i] := x
end in parallel
```

However, this statement is not admissible on an EREW PRAM (unless $n = 1$), since it calls for the single memory location x to be read concurrently by n processors. In order to fit the EREW PRAM model, we need to use the broadcasting method which is implemented in the algorithm in Figure 3.

```
procedure broadcastPRAM (A[1:n], x)

input:  A[1:n]  (an array of size n, where n = 2^k)
          x (a value to be broadcast throughout A[1:n])

output:  A[1:n] (array where A[i] = x, i = 1, …, n)

    A[1] := x
    for i := 1  to k do
        for 2^(i-1) + 1 ≤ j ≤ 2^i do in parallel
            A[ j ] := A[ j – 2^(i-1)]
        end in parallel
    endfor
end  broadcastPRAM
```

Figure 3 – EREW PRAM broadcast algorithm.

Recall from our early examination of this parallel algorithm that once the value of *x* has been broadcast throughout *temp[1:n]*, each processor P_i in parallel compares *L[i]* to *temp[i]* = *x* and writes *i* back to *temp[i]* if *L[i]* = *x*, otherwise it writes the value ∞ in *temp[i]*. The comparison step is accomplished by the code segment shown in Figure 4.

```
for 1 ≤ i ≤ n do in parallel
  if L[ i ] = temp[ i ] then
        temp[ i ] := i
  else
        temp[ i ] := ∞
  endif
end in parallel
```

Figure 4 – The comparison step for parallel searching.

The parallel searching algorithm was not completed until the result of the search was returned to the calling function. This was accomplished through the binary fan-in technique to move the minimum value in the list into the first position in the array. The parallel code to accomplish this is shown in Figure 5.

```
function minPRAM( L[1:n])

model: EREW PRAM with p = n/2 processors

input:  L[1:n] (a list of size n, n = 2^k)

output: the minimum value of a list element in L

   for j := 1 to log2n do
      for 1 ≤ i ≤ n/2^j do in parallel
         if L[2i-1] > L[2i] then
               L[i] := L[2i]
         else
               L[i] := L[2i-1]
         endif
      end in parallel
   endfor
   return(L[1])

end minPRAM
```

Figure 5 – Parallel algorithm for binary fan-in technique.

Now with all the pieces in place, we can assemble our final algorithm for searching in an EREW PRAM, which is shown in Figure 6.

```
procedure searchPRAM( L[1:n], x)

input:  L[1:n] (a list of size n, n = 2^k)
        x (a search element)

output: the smallest index where x occurs in L, or ∞ if x is ∉ L

    call broadcast( temp[1:n], x)
    for 1 ≤ i ≤ n do in parallel
        if L[i] = temp[i] then
            temp[i[ := i
        else
            temp[i] = ∞
        endif
    end in parallel
    return (minPRAM(temp[1:n])

end searchPRAM
```

Figure 6 – EREW PRAM searching algorithm (complete).

Parallel Algorithm Pseudocode Conventions for Mesh Models

In general, the pseudocode algorithms designed for interconnection network models (our focus is on the mesh models) are more complex than that for PRAMs. This is mostly due to the fact that we must now include statements which describe the communication between processors, specify how the processors handle I/O, and distinguish between central control variables (commonly called *front-end variables)* and distributed variables.

Opening Syntax

Each procedure for an interconnection network model begins with opening syntax similar to that for the PRAM model. Parameters in the parameter list of a procedure or function on an interconnection network model will now include distributed variables. Therefore, for each distributed variable parameter is important to describe the set of processors that contain meaningful information on input for that parameter. Thus, the **input** statement not only describes the high-level nature of the parameter, but also includes a **range** clause specifying the range of indices of processors containing meaningful input data for the parameter. Those parameters for which no range is given in the **input**

statement are *central control variables* (also called *front-end variables*) which are resident in the local memory of the front-end processor (central control). The front-end variable parameters are usually used to specify ranges of indices in the **input** and **output** statements. The **output** statement contains similar specifications for all output parameters. As with the PRAM model, the declaration of local variables (if needed) is done via the **dcl** statement. Those local variables that are distributed also have a range specification. The general format of the opening syntax is shown below:

procedure <name> (<list of parameters>)

model: <model name> **with** p = <function of *n*> **processors**

input: <description of input parameter 1> **range:** <processor range 1>
•
•
•
<description of input parameter *i*> **range:** <processor range *i*>

output: <description of output parameter 1> **range:** <processor range 1>
•
•
•
<description of output parameter *j*> **range:** <processor range *j*>

dcl: <local variable1> **range:** <processor range 1>
•
•
•
<local variable *k*> **range:** <processor range *k*>

The in parallel Statement

The **in parallel** statement is similar to that for the PRAM, except that the **for** clause contains a Boolean expression involving *processor* indices instead of array indices. The processor index might be a single index as in the one dimensional mesh, a pair of indices as in the two-dimensional mesh, and a bit string in the case of the hypercube, and so forth. It is implicit in the pseudocode that we are developing that each processor involved in an **in parallel** statement recognizes its own index (*pid*). [In practice, this is commonly accomplished using a distributed variable *id* in the read-only local memory of each processor. Then for each processor *P*, **P**:*id* would contain the index of processor *P*.] Each processor whose index does not satisfy the Boolean expression in the **for** clause is idle (masked out). The active processors simultaneously execute the

instructions in the body of the **in parallel** statement, while the remaining processors are idle. The general form of the syntax for the **in parallel** statement is:

for <processor *P*>, <Boolean expression involving index of *P*> ***do in parallel***
    <statement 1>
    <statement 2>
        •
        •
        •
    <statement *k*>
end in parallel

Note that the **in parallel** statement is counted as representing *k* parallel steps. We'll see the full meaning of this statement later, but keep it in mind now because it represents a fundamental difference between PRAM and mesh model parallel processing.

For interconnection networks, we'll consider two main types of functions: those that are simply parallel invocations of sequential functions and those that require communication between processors. The first type of function works within each processor's local memory and does not require communication between processors. For example, consider a parallel invocation of the built-in function *sin* on a one-dimensional mesh M_p. Then, the statement

for P_i , $1 \leq i \leq n$ **do in parallel**
    B := sin(A)
end in parallel

where *A* and *B* are distributed variables, assigns sin($\mathbf{P}_i$: *A*) to $\mathbf{P}_i$: *B*, $i = 1, \ldots, n$.

The second type of function, which requires communication between processors, is restricted to functions that return a scalar. When such a function terminates, it is assumed that the value to be returned by the function resides in a particular processor's local instantiation of a suitable distributed variable. For example, suppose the scalar value to be returned by a function implemented on a two-dimensional mesh resides in $\mathbf{P}_{1,1}$: *x*. Then the **return** statement is given by: **return**($\mathbf{P}_{1,1}$: *x*). The **return** statement is used only for convenience and should not be interpreted as returning a value to the front-end processor that is directly accessible to all the processors. The returned value is resident in a suitable output register of $\mathbf{P}_{1,1}$ only. If other processors require this returned value, then it must be broadcast.

Interprocessor Communication Statement

In the interconnection network model, recall that we used the syntax:

$$\mathbf{P}_j: y \Leftarrow \mathbf{P}_i: x$$

for communicating processor P_i's local variable x to an adjacent processor P_j's local variable y. This communication process can be viewed as consisting of the following steps: Processor P_i fetches the value of x from its local memory and stores it in an output register. Then the contents of this output register is communicated to an input register of the adjacent processor P_j. Finally, P_j writes the value in this input register to the local variable y. We will assume that these steps constitute a single communication step which requires unit time.

The general form of the interprocessor communication statement is:

<target processor: target variable> ⇐ <source processor: source variable>

The interprocessor communication statement is typically embedded in the body of an **in parallel** statement, in which case the **for** clause includes a Boolean expression involving source processor indices that determine the set of active source processors. Any active source and corresponding target processor must be adjacent in the interconnection network. A sample parallel interprocessor communication statement is shown in Figure 7.

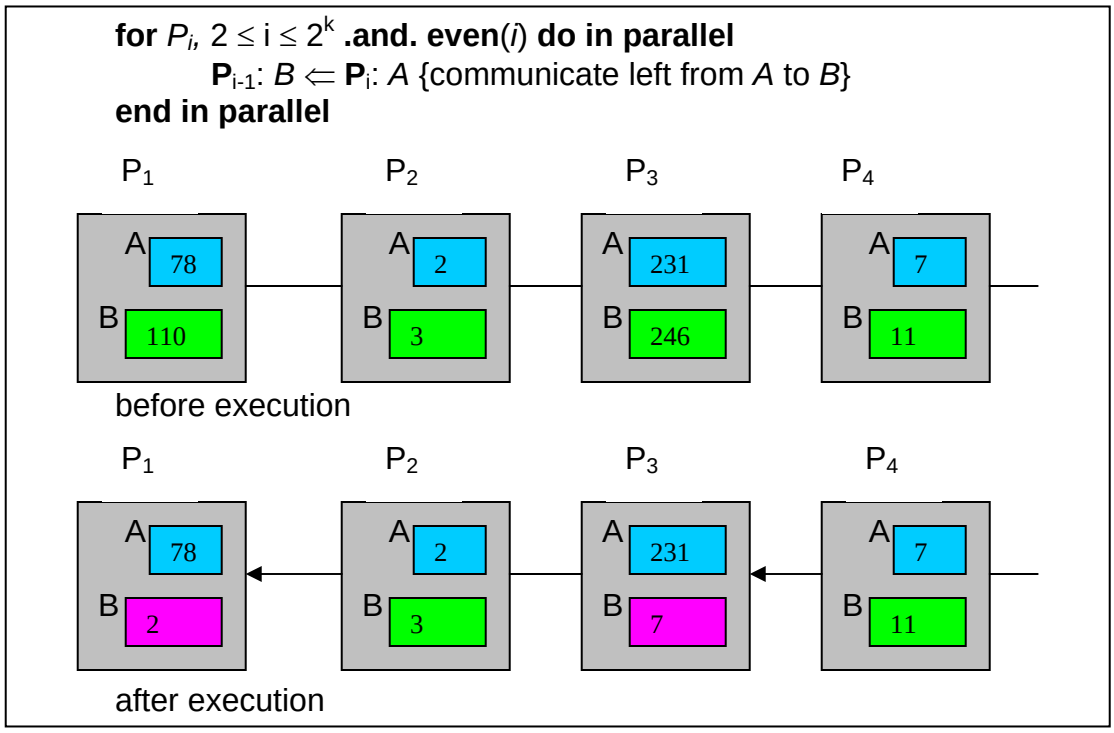

Figure 7 – Action of a parallel communication step on a 1-d mesh.

Communication is only one direction in a single parallel step, in order to maintain a strict SIMD model. For example, in a one-dimensional mesh, some of the active processors cannot be communicating to the left while other active processors are communicating to the right in the same parallel communication step. If the communication involves the same distributed variable, the term *propagate* rather than communicate is used. Figure 8 illustrates a parallel communication step where processors are both source and target.

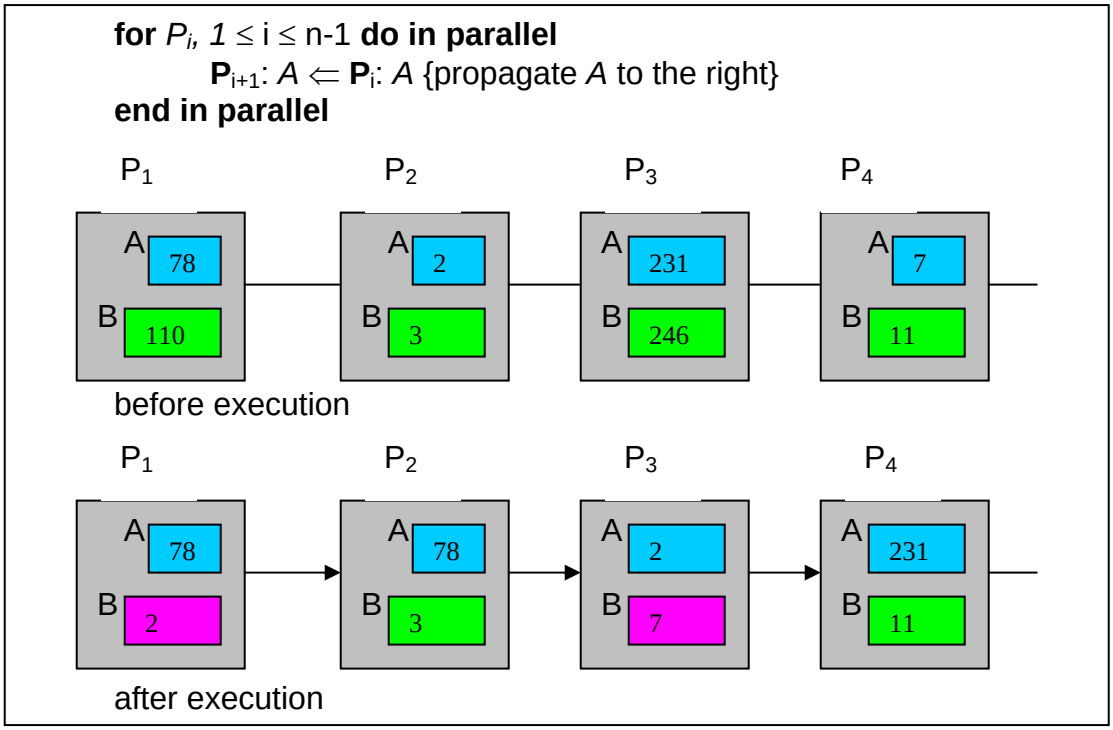

Figure 8 – Action of a parallel communication step on a 1-d mesh with processors being both source and target.

Figure 8 presents a reasonable view of interprocessor communication in light of the systolic nature of a communication step. At the beginning of a communication step, all active source processors initiate the process of sending (pulsing) data from their local memory to the relevant adjacent processor. At the end (after unit time has elapsed) of the communication step, this data has been received and assigned by the target processor. Thus, it is both a natural and common feature of many parallel algorithms to have the same processor send information at the beginning of the communication step and receive information at the end of the step. However, it is *not* permitted for any processor to be the target or source of more than one processor during a single communication step.