Parallel Algorithms – Part 4

Parallel Algorithm Pseudocode Conventions for Mesh Models – Continued

Read/Write Statements

We will assume the existence of two parallel I/O statements, **read** for input and **write** for output. Suppose, for example, that we have a list of *n* values stored sequentially in the input device. These *n* values might be read into a one-dimensional mesh M_P by the following statement:

for P_i, $1 \leq i \leq n$ **do in parallel**
    read(P_i: L)
end in parallel

This **in parallel** statement simultaneously reads the *i*th value in the external input device into the variable $\mathbf{P}_i$:L in the local memory of processor P_i, $i \in \{1, \ldots, n\}$.

Typically, when using **read** or **write** statements in a procedure or function, a **dcl** statement is included which covers all variables participating in I/O, together with **range** descriptions when interconnection models are considered (the range variable is not required for PRAM models due to the shared nature of the memory). Also, the terms **external input** and **external output** will be utilized in the opening syntax.

Algorithms for the Mesh Model

As we did for the PRAM model, we now will develop a complete algorithm for searching on the two-dimensional mesh model.

Recall from our earlier discussions of this algorithm, that similar to the PRAM algorithm for the same problem, the 2-d mesh algorithm for searching consisted of three distinct phases, the first of which was broadcasting phase in which the value of the search element *x* was broadcast as a distributed element to all the participating processors. Using the pseudocode that we have developed for the interconnection network models, this algorithm would have the pseudocode shown in Figure 1.

```
procedure broadcast2Dmesh (P1,1:x, n)

model: Two-dimensional mesh Mq,q with p = n = q^2 processors

input:  P1,1:x (element to be broadcast)

output: P1,1:x (element is broadcast to each processor in Mq,q)

        for i := 1 to  q-1 do
            P1, i+1:x ⇐ P1, i: x          {propagate x to right across first row}
        endfor

        for i := 1 to q-1 do
            for Pi, j, 1 ≤ j ≤ q do in parallel
                  Pi+1, j:x ⇐ Pi, j: x      {propagate x down row by row}
            end in parallel
        endfor

end broadcast2Dmesh
```

Figure 1 – Parallel pseudocode algorithm for broadcasting in a 2D mesh.

The second phase of the algorithm begins after the search value has been broadcast through the distributed variable. Each processor $\mathbf{P}_{i,\ j}$ in parallel compares $\mathbf{P}_{i,j}$: L to $\mathbf{P}_{i,j}$: x and writes the value ∞ into $\mathbf{P}_{i,j}$: *index* if $\mathbf{P}_{i,j}$: L $\neq$ $\mathbf{P}_{i,j}$: *x*. This comparison is accomplished using the pseudocode shown in Figure 2.

```
for Pi, j, 1 ≤ i, j ≤ q do in parallel
   if Pi, j: L ≠ Pi,j: X then
        Pi, j: index := ∞
    endif
end in parallel
```

Figure 2 – Pseudocode to perform the parallel search in the 2d mesh.

The third and final phase of the algorithm consists of a reverse broadcast technique to filter the minimum processor *index* value into processor $P_{1,1}$. The algorithm for this final phase is shown in Figure 3.

```
function min2Dmesh (x, n)

model: Two-dimensional mesh M_{q,q} with p = n = q^2 processors

input: x (a list of n numbers)        range: P_{i, j}, 1 ≤ i, j ≤ q

output: min {x_1, x_2, …, x_n}

        for row := q-1 downto 1 do          {compute column minimums}
            for P_{i, j}, i = row .and. 1 ≤ j ≤ q do in parallel
                  {compute (i+1)st row and ith row minimums in parallel}
                  P_{i, j}: temp ⇐ P_{i+1, j}: x  {communicate up from x to temp}
                  x := min{x, temp}  {compute min of P_{i, j}: x and P_{i, j}: temp}
             end in parallel
        endfor

        {compute first row minimum sequentially}
        for column := q-1 downto 1 do
            for P_{i, j}, i = 1 .and.  j = column do in parallel
                P_{i, j}: temp ⇐ P_{i, j+1}: x  {communicate left from x to temp}
                x := min{x, temp}    {only P_{1, j} is active}
            end in parallel
        endfor
        return( P_{1,1}: x)

end min2Dmesh
```

Figure 3 – Pseudocode parallel algorithm for reverse broadcast technique.

Having developed the necessary pseudocode for the three basic phases of our searching algorithm on the 2-d mesh, we can now put the pieces together and present our complete searching algorithm which is shown in Figure 4. Recall that we will input the search value to the front end processor as a front end variable as opposed to a distributed variable that has an instantiation in each processor.

```
function search2Dmesh( L, n, x, index)

model: two dimensional mesh M_{q,q} with  p = n= q^2 processors

input:  L (a list of elements)          range: P_{i, j}, 1 ≤ i, j ≤ q
        x (a search element)            range: front end variable
        index (P_{i, j}: index = (q-1)i + j)   range: P_{i, j}, 1 ≤ i, j ≤ q

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

        P_{1,1}: x = x
        call broadcast2Dmesh( P_{1, 1}: x, n)
        for 1 ≤ i, j ≤ q do in parallel
            if P_{i, j}: L ≠ P_{i, j}: x then
                P_{i, j}: index := ∞
            endif
        end in parallel
        return(min2Dmesh(index, n))

end search2Dmesh
```

Figure 4 – Complete searching algorithm in parallel pseudocode for 2d mesh.

Performance Measures for Parallel Algorithms

SIMD computers have the property that all the active processors (remember that some may be idle on a given parallel step) perform the same operation (on possibly different data) during any parallel step. The set of these common operations performed during a parallel step is known as a *parallel basic operation*. The best-case, average, and worst-case complexities of a parallel algorithm are defined in terms of the number of parallel basic operations performed. For example, the worst-case complexity W(n) of a parallel algorithm using *p(n)* processors is defined to be the maximum number of parallel basic operations performed by the algorithm over all inputs of size *n*. The best-case complexity B(n) and average complexity $\lambda(n)$ are defined in a similar manner.

For parallel algorithms, the number of parallel basic operations performed usually depends only on the size of the input *n*, so that the best-case and average complexities are the same as the worst-case complexity.

One measure of the performance of a parallel algorithm results from comparing W(n) to the smallest worst-case complexity W*(n) over all sequential algorithms

for the problem. This leads to the formal definition of the *speedup* S(n) of the parallel algorithm shown in Figure 5.

Speedup of a Parallel Algorithm

Let W(n) denote the worst-case complexity of a parallel algorithm for solving a given problem, and let W*(n) denote the smallest worst-case complexity over all known sequential algorithms for solving the same problem. Then the speedup S(n) of the parallel algorithm is defined by:

$$S(n) = \frac{W^*(n)}{W(n)}$$

Figure 5 – Definition of the speedup of a parallel algorithm in terms of sequential counterpart.

The definition of speedup does not explicitly depend on the number of processors used by the algorithm. Thus, the speedup in isolation is not a true measure of the efficiency of the parallel algorithm. Good speedup usually comes with the additional cost of using many processors. Thus, when measuring efficiency of a parallel algorithm, it is important to consider both the worst-case complexity W(n) and the number of processors *p(n)* used.

Cost of a Parallel Algorithm

Given the definition shown in Figure 5 for the worst-case complexity of a parallel algorithm using *p(n)* processors for solving a given problem, the *cost* C(n) of the parallel algorithm is defined as shown in Figure 6 below.

$$C(n) = p(n) \times W(n)$$

Figure 6 – Definition of the cost of a parallel algorithm.

To judge the quality of a parallel algorithm, it is always useful to compare its cost to W*(n). A parallel algorithm is *cost optimal* if C(n) = W*(n). A parallel algorithm is considered *cost efficient* if C(n) is within a polylogarithmic factor of being cost optimal (a polylogarithmic function belongs to $O(\log^k n)$).

There is a tradeoff between using more processors to achieve better speedup and fewer processors to achieve cost optimality. Indeed, a sequential algorithm having optimal worst-case complexity for a given problem is at the same time cost optimal. Note that the cost C(n) equals the total number of basic operations performed by the algorithm only if each parallel basic operation consists of *p(n)* basic operations; that is, only if none of the *p(n)* processors utilized by the algorithm are idle when a parallel basic operation is performed.

Thus, if the *p(n)* processors are all active doing useful work, we can expect the ratio W*(n)/C(n) to be close to 1 (since the worst-case complexity of a sequential algorithm is measured in terms of the total number of basic operations). The ratio W*(n)/C(n), which indicates how effectively the processors are utilized, is known as the *efficiency* E(n).

$$E(n) = \frac{W^*(n)}{C(n)}$$

Figure 7 – Definition of the efficiency of a parallel algorithm.

From the definition of the efficiency of a parallel algorithm given in Figure 7, it is immediately apparent that:

$$E(n) = \frac{S(n)W(n)}{p(n)W(n)} = \frac{S(n)}{p(n)}$$

Figure 8 – Definition of the efficiency of a parallel algorithm in terms of speedup and number of processors.

Note that $E(n) \leq 1$, since otherwise a faster sequential algorithm can be obtained than a parallel one! Further, a parallel algorithm is cost optimal iff $E(n) = 1$. Finding cost-optimal algorithms that also show good speedup is usually difficult because of the trade-off we mentioned earlier with respect to the increasing the number of processors to achieve a better speedup opposed to reducing the number of processors to achieve cost optimality. Good speedup might come at the cost of using many processors, perhaps forcing more and more of the processors to remain idle as the algorithm progresses toward completion (consider the binary fan-in technique as an example).

To illustrate these performance measures for parallel algorithms let's consider the minPRAM algorithm we developed in the previous set of notes (duplicated below as Figure 9.

We saw earlier that the minPRAM algorithm has complexity $W(n) = \log_2 n$. Since the best sequential algorithm for finding the maximum of *n* elements has complexity $W^*(n) = n - 1$, minPRAM exhibits a speedup of:

$$S(n) = \frac{n-1}{\log_2 n}$$

Since the minPRAM algorithm utilizes *n/2* processors, it has cost and efficiency given by:


```
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 9 – minPRAM parallel algorithm from previous set of notes (Parallel Algorithms III).

$$C(n) = \binom{n}{2}\log_2 n \qquad \text{and} \qquad E(n) = \frac{2(n-1)}{n\log_2 n}$$

However, minPRAM is not cost optimal because C(n) is greater than W*(n)!

To illustrate the importance of using more than just speedup as a measure of performance, consider the problem of finding the minimum on a CRCW PRAM. Using $\binom{n}{2} = (n^2 - n)/2$ processors and the CRCW PRAM model, we can design an algorithm minCRCW that finds the minimum value in a list *L*[1:n] of size *n* using a single parallel comparison step! However, the cost $(n^2 - n)/2$ of minCRCW is even higher than the cost of minPRAM. Using the CRCW PRAM model allows the processors to write concurrently to the same memory location only if they are writing the same value. Let's denote the $\binom{n}{2} = (n^2 - n)/2$ processors used by minCRCW by $P_{i,j}$, i, j ∈ {1, …, n}, i < j. In one parallel step a shared memory array *win*[1:n] is initialized to 0. The array *win* is used to store the results of the "win-loss" comparisons of the elements in *L*. For each pair of numbers L[i] and L[j], i < j, $P_{i,j}$ reads L[i] and L[j], compares them, writes a 1 to *win*[i] if L[i] > L[j], and writes a 1 to *win*[j] otherwise. Obviously, only one index *k* has the property that the corresponding array element L[k] loses each of the *n-1* comparisons involving L[k]. Therefore, *win*[i] = 1, i ≠ k, and *win*[k] = 0. The value of *k* is determined in one parallel step by assigning *n* processors the task of reading the array *win.* The action of the minCRCW algorithm is shown in Figure 10.

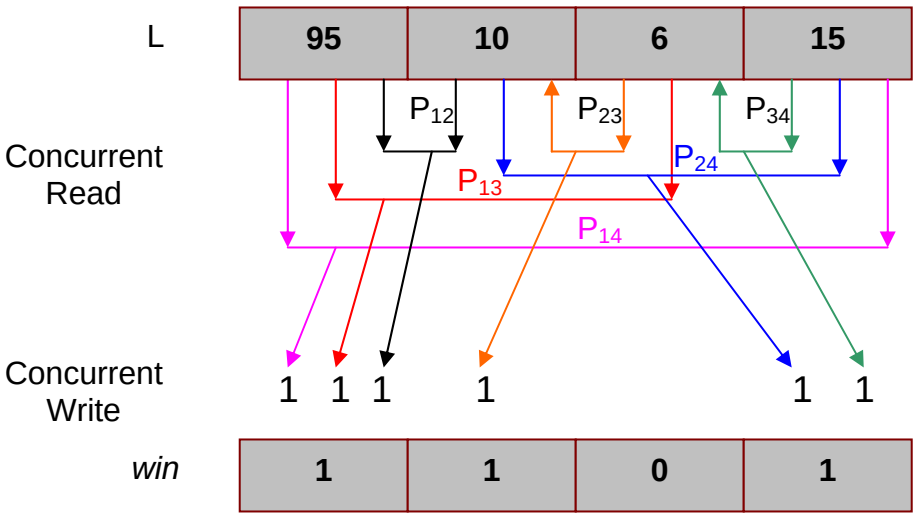

Figure 10 – Action of algorithm minCRCW on sample list using six processors.

Figure 11 gives the parallel pseudocode algorithm for finding the minimum value in a list using the CRCW PRAM model. Use this algorithm to determine the actions which are shown in parallel in Figure 10.

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

model: CRCW PRAM with p = (n^2 – n)/2 processors

input: L[1:n] (a list of size n)
output: the minimum value in the list L

        for 1≤ i ≤ n do in parallel
            win[i] := 0
        end in parallel
        for 1 ≤ i, j ≤ n .and. i < j do in parallel
            {P_i, j reads and compares L[i] and L[ j ]}
            if L[i] > L[ j ] then
                win[i] := 1  {processors P_i, j concurrently write 1 to win[i]}
            else
                win[ j ] := 1 {processors P_i, j concurrently write 1 to win[ j ]}
            endif
        end in parallel
        for 1 ≤ i ≤ n do in parallel
            if win[i] = 0 then
                indexmin := i
            endif
        end in parallel
        return(L[indexmin])
end minCRCW
```

Figure 11 – Parallel algorithm for finding minimum value using CRCW PRAM.

The minCRCW algorithm has worst-case complexity W(n) = 1, since it only performs a single comparison step. Therefore, minCRCW has speedup S(n) = n – 1. However, there is clearly a drawback to this algorithm…can you tell what it is? The drawback is in the large cost and low efficiency measures since:

$$C(n) = \frac{n^2 - n}{2} \qquad \text{and} \qquad E(n) = \frac{2}{n}$$

Table 1 lists the various performance measures for the parallel algorithms that we have examined in this set of notes.

Algorithm	Basic Operation	p(n)	W(n)	S(n)	C(n)	E(n)
minPRAM	<	n	$\log_2 n$	$n/\log_2 n$	$n \log_2 n$	$1/\log_2 n$
SearchPRAM	<	n	$\log_2 n$	$n/\log_2 n$	$n \log_2 n$	$1/\log_2 n$
min2Dmesh	<	n	$\sqrt{n}$	$\sqrt{n}$	$n^{3/2}$	$1/\sqrt{n}$
search2Dmesh	<	n	$\sqrt{n}$	$\sqrt{n}$	$n^{3/2}$	$1/\sqrt{n}$
minCRCW	<	n^2	1	n	n^2	1/n

Table 1 – Big-Oh performance measures for parallel algorithms.

Speedup and Amdahl's Law

Utilizing *p* processors versus a single processor can yield significant speedup for certain problems. Ultimate speedup occurs for problems that involve independent operations requiring little or no communication between processors. For example, adding two *n*-dimensional vectors *x = (x1, x2, …, xn)* and *y = (y1, y2, …, yn)*, *p(n) = n*, can be done in a single parallel step: simultaneously processor P_i adds *xi* and *yi* for $i \in \{1, \ldots, n\}$. This represents a speedup of a factor of *n* over the *n* sequential steps required to do this vector addition on a single processor machine. However, problems which yield ultimate speedup are rare. Most problems have an inherently sequential component and therefore cannot be completely parallelized. Amdahl's law expresses an upper bound for the speedup achievable by any parallel algorithm for a given problem in terms of the inherently sequential component of the problem.

Note that any parallel algorithm can be thought of as a parallelization of the associated sequential algorithm that performs the operations in each parallel operation sequentially. Now suppose that we are given a sequential algorithm that we wish to parallelize and that a fraction *f* of the basic operations must be performed sequentially for any input (no matter what the input size). If we have at most *p* processors, then the parallelization of the sequential algorithm has complexity at least *f + (1- f)/p* times the complexity of the sequential algorithm. Thus, for any input to the algorithm, the parallelized algorithm achieves a speedup of at most $1/[f + (1-f)/p]$ over the sequential algorithm. This upper bound on the speedup achievable for any input by parallelizing a sequential algorithm is known as Amdahl's law which is:

Amdahl's Law: $S \leq \frac{1}{f + (1-f)/p}$

It would appear that Amdahl's law severely limits the speedup that is achievable, in practice however, the fraction *f* is often dependent on the size of the input and diminishes as the input size increases.

Summary

The wide variety of parallel architectures that are available presents a fundamental problem of determining the portability of an algorithm written for a specific architecture. In the case of PRAMs, algorithms designed for CRCW and CREW models with *p* processors can be simulated on the EREW PRAM with *p* processors at a cost of a multiplicative complexity factor of $\log_2 p$. In the case of interconnection network models, the portability question is usually handled by establishing efficient ways in which to embed one interconnection model into another. An embedding from an interconnection model *A* into another model *B* yields a canonical translation of any algorithm written for *A* to one suitable for *B.* Embeddings between interconnection networks is a very active research area today.

A sequential algorithm is entirely impractical unless it has polynomial complexity $O(n^k)$ for some positive integer *k*. Sequential algorithms are sometimes called time efficient if they have polynomial worst-case complexity. A parallel algorithm on a PRAM is considered to be time efficient if it has polylogarithmic $O((\log_2 n)^k)$ worst-case complexity. A parallel algorithm for solving a given problem, which can be solved by a sequential algorithm in polynomial time (belongs to the class P), is said to be in the class NC (Nick's class) if it has polylogarithmic complexity using a polynomially bounded number of processors. The class NC is an important class from a theoretical point of view, although with present technology, the assumption of more than a linear number of processors is impractical for large *n*, is impractical. A fundamental unsolved question is whether P = NC which is equivalent to the question of whether P = NP.