Advanced Algorithms **Fall 2015**

Lecture 6: Parallel Algorithms, Dec. 01, 2015

Faculty: K.R. Chowdhary *: Professor of CS*

6.1 Modeling parallel computation using Work-Depth models

There are many different ways to organize the parallel computers, hence to model them also. An alternative way is to focus on algorithm. One approach is work-depth where cost of an algorithm is determined by examining the total number of operations that it performs and the dependencies among those operations. An algorithm's work W is the total number of operations that it performs; and its depth D is the longest chain of dependencies among its operations. The ratio $P = W/D$ is *parallelism* of the algorithm. A parallel algorithm is work efficient as compared to a sequential algorithm.

The work-depth models are more abstract than the multi-processor models. The algorithms that are efficient in the work-depth (w-d) models can be translated to algorithms that efficient in the multiprocessor models, and from these models they can be translated to parallel computers. The advantage of w-d models are that they do not have machine dependent details, which otherwise would complicate the design and their analysis.

There are three types of w-d models: *circuit models, vector machine models, language-based models.*

6.1.1 Circuit Model

The most abstract *w-d* model is circuit model. An algorithm is modeled as a family of directed acyclic circuits. There is a circuit for every possible size of the input. A circuit consist of nodes and arc. A node represents a basic operation, like addition of two values. For each node, there is an input of two or mode values, and one or more outputs, to other nodes sending the result. The work of the circuit is total number of nodes (also called size). The depth of the circuit is longest directed path between any two nodes (see fig. 6.1). The figure shows a circuit for adding 16 numbers. In this figure all arcs are directed toward the bottom. The input arcs are at the top of the figure. Each '+' node adds the two values that arrive on its two incoming arcs, and places the result on its outgoing arc. The sum of all of the inputs to the circuit is returned on the single output arc at the bottom.

The work in above example is 15. The depth is the number of nodes on the longest directed path from an input arc and an output arc, is 4. For a family of circuits, the work and depth are typically parameterized in terms of the number of inputs. For example, the circuit in figure 6.1 can be easily generalized to add n input values for any n that is a power of two. The work and depth for this family of circuits is $W(n) = n - 1$ and $D(n) = \log_2 n$.

6.1.2 Vector Model

In vector model, an algorithm is expressed as a sequence of steps, each of them performs an operation on vector of input values, and produces vector result. Work of each step is equal to length of input or output

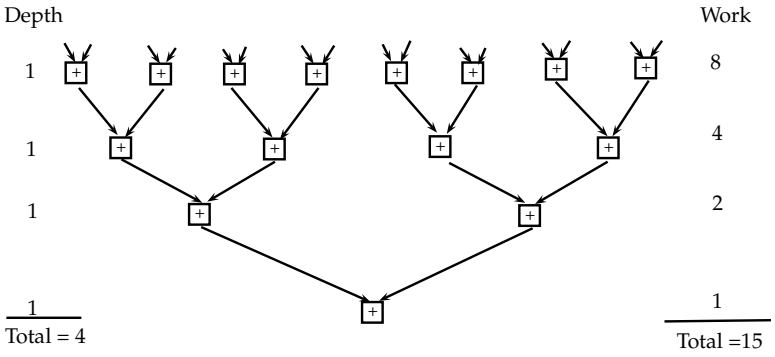

Figure 6.1: Work-depth model.

vector, where as the work of the algorithm is sum of work of its steps. The depth of an algorithm is number of vector steps.

Let there are three vectors A, B, C, expressed as $A_1, A_2, \ldots A_n$, and $B_1, B_2, \ldots, B_n$, and $C_1, C_2, \ldots, C_n$. It is required to compute: $C_1 = A_1 \times B_1$, $\ldots$, $C_n = A_n \times B_n$. The input vectors are A, B and output vector is C. There is only one vector step, i.e., $\times$ on the input vectors. Since size of input is n, so work is n. The depth is number of vector steps, which is 1, so parallelism is $n/1 = n$.

6.1.3 Language Model

In a language model, a work-depth cost is associated with each programming language construct. For example, work of calling two functions in parallel is equal to sum of the work of two calls. The depth is equal to the maximum of the depth of the two calls.

Assigning cost to algorithms. In the work-depth models, the cost of an algorithm is determined by its work and by its depth. The notions of work and depth can also be defined for the multiprocessor models. The work W performed by an algorithm is equal to the number of processors multiplied by the time required for the algorithm to complete execution. The depth D is equal to the total time required to execute the algorithm.

The *depth* of an algorithm is important, because there are some applications for which the time to perform a computation is crucial. For example, the results of a weather forecasting program are useful only if the program completes execution before the weather does!

Generally, however, the most important measure of the cost of an algorithm is the work. This can be argued as follows. The cost of a computer is roughly proportional to the number of processors in the computer. The cost for purchasing time on a computer is proportional to the cost of the computer multiplied by the amount of time used. The total cost of performing a computation, therefore, is roughly proportional to the number of processors in the computer multiplied by the amount of time, i.e., the work.

In many instances, the cost of running a computation on a parallel computer may be slightly larger than the cost of running the same computation on a sequential computer. If the time to completion is sufficiently improved, however, this extra cost can often be justified. As we shall see, however, there is often a trade-off between time-to-completion and total work performed. To quantify when parallel algorithms are efficient in terms of cost, we say that a parallel algorithm is work-efficient if asymptotically (as the problem size grows) it requires at most a constant factor more work than the best sequential algorithm known.

6.2 Designing Parallel Algorithms

Many parallel algorithms are purely sequential, with the same overall structure as an algorithm designed for a more standard "serial" computer. That is, there may be an *input and initialization* phase, then a *computational* phase, and then an *output and termination* phase. The differences, however, are manifested within each phase. For example, during the computational phase, an efficient parallel algorithm may be inherently different from its efficient sequential counterpart.

For each of the phases of a parallel computation, it is often useful to think of operating on an entire structure simultaneously. This is an SIMD-style approach, but the operations may be quite complex. For example, one may want to update all entries in a matrix, tree, or database, and view this as a single (complex) operation. For a fine-grained machine, this might be implemented by having a single (or few) data item per processor, and then using a purely parallel algorithm for the operation. For example, suppose an $n \times n$ array A is stored on an $n \times n$ two-dimensional torus, so that $A(i,j)$ is stored on processor $P_{i,j}$. Suppose one wants to replace each value $A(i,j)$ with the average of itself and the four neighbors $A(i-1,j)$, $A(i+1,j)$, $A(i,j-1)$, and $A(i,j+1)$, where the indices are computed *modulo* n (i.e., "neighbors" is in the torus sense). This average filtering can be accomplished by just shifting the array right, left, up, and down by one position in the torus, and having each processor average the four values received along with its initial value.

For a medium or coarse-grained machine, operating on entire structures is most likely to be implemented by blending serial and parallel approaches. On such an architecture, each processor uses an efficient serial algorithm applied to the portion of the data in the processor, and communicates with other processors in order to exchange critical data. For example, suppose the $n \times n$ array of the previous paragraph is stored in a $P \times P$ torus, where P evenly divides n, so that $A(i,j)$ is stored in processor $P_{\lfloor (i*p)/n \rfloor, \lfloor (j*p)/n \rfloor}$. Then, to do the same average filtering on A, each processor $P_{k,l}$ still needs to communicate with its torus neighbors $_{k\pm1,l}$, $P_{k,l\pm1}$, but now sends them either the leftmost or rightmost column of data, or the topmost or bottommost row. Once a processor receives its boundary set of data from its neighboring processors, it can then proceed serially through its subsquare of data and produce the desired results. To maximize efficiency, this can be performed by having each processor send the data needed by its neighbors, then perform the filtering on the part of the array that it contains that does not depend on data from the neighbors, and then finally perform the filtering on the elements that depend on the data from neighbors. Unfortunately, while this maximizes the possible overlap between communication and calculation, it also complicates the program because the order of computations within a processor needs to be rearranged.

6.2.1 Global Operations

To manipulate entire structures in one step, it is useful to have a collection of operations that perform such manipulations. These global operations may be very problem-dependent, but certain ones have been found to be widely useful. For example, the average filtering example above made use of shift operations to move an array around. Broadcast is another common global operation, used to send data from one processor to all other processors. Extensions of the broadcast operation include simultaneously performing a broadcast within every (predetermined and distinct) subset of processors. For example, suppose matrix A has been partitioned into submatrices allocated to different processors, and one needs to broadcast the first row of A so that if a processor contains any elements of column i, then it obtains the value of $A(1,i)$. In this situation, the more general form of a subset-based broadcast can be used.

Besides operating within subsets of processors, many global operations are defined in terms of a commutative, associative, semigroup operator $\otimes$, and $\oplus$. Examples of such semigroup operators include `minimum`, `maximum`, `or`, `and`, `sum`, and `product`. For example, suppose there is a set of values $V(i), 1 \leq i \leq n$, and the goal is to obtain the maximum of these values. Then $\otimes$ would represent maximum, and the operation of applying $\otimes$ to all n values is called reduction. If the value of the reduction is broadcast to all processors, then it is

sometimes known as report. A more general form of the reduction operation involves labeled data items, i.e., each data item is embedded in a record that also contains a label, where at the end of the reduction operation the result of applying $\otimes$ to all values with the same label will be recorded in the record.

Global operations provide a useful way to describe major actions in parallel programs. Further, since several of these operations are widely useful, they are often made available in highly optimized implementations. The language APL provided a model for several of these operations, and some parallel versions of APL have appeared. Languages such as $C*$ provide for some forms of global operations, as do message-passing systems such as MPI. Reduction operations are so important that most parallelizing compilers detect them automatically, even if they have no explicit support for other global operations.

Besides *broadcast*, *reduction*, and *shift*, other important global operations include the following.

Sort: Let $X = \{x_0, x_1, \ldots, x_{n-1}\}$ be an ordered set such that $x_i < x_{i+1}$, for all $0 \leq i < n-1$. (That is, X is a subset of a linearly ordered data type.) Given that the n elements of X are arbitrarily distributed among a set of p processors, the sort operation will (re)arrange the members of X so that they are ordered with respect to the processors. That is, after sorting, elements $x_0, \ldots, x_{\lfloor n/p \rfloor}$ will be in the first processor, elements $x_{\lfloor n/p \rfloor + 1}, \ldots, x_{\lfloor 2n/p \rfloor}$ will be in the second processor, and so forth. Note that this assumes an ordering on the processors, as well as on the elements.

Merge: Suppose that sets D_1 and D_2 are subsets of some linearly ordered data type, and D_1 and D_2 are each distributed in an ordered fashion among disjoint sets of processors P_1 and P_2, respectively. Then the merge operation combines D_1 and D_2 to yield a single sorted set stored in ordered fashion in the entire set of processors $P = P_1 \cup P_2$.

Associative read/write: These operations start with a set of master records indexed by unique keys. In the associative read, each processor specifies a key and ends up with the data in the master record indexed by that key, if such a record exists, or else a flag indicating that there is no such record. In the associative write, each processor specifies a key and a value, and each master record is updated by applying $\oplus$ to all values sent to it. (Master records are generated for all keys written.)

These operations are extensions of the CRCW PRAM operations. They model a PRAM with associative memory and a powerful combining operation for concurrent writes. On most distributed memory machines the time to perform these more powerful operations is within a multiplicative constant of the time needed to simulate the usual concurrent read and concurrent write, and the use of the more powerful operations can result in significant algorithmic simplifications and speedups.

6.3 Parallel Algorithm Techniques

Like the sequential algorithms, there are techniques in parallel algorithms. These are divide-and-conquer, randomization, and parallel pointer manipulation.

6.3.1 Divide-and-conquer

It first splits the problem into subproblems that are easier to solve than the original, either because they are smaller instances of the original problem, or because they are different but easier problems. Next, the algorithm solves the subproblems recursively. These subproblems can be solved independently. Finally, the solutions are merged to construct the original algorithm. It is also possible to divide the original problem in many steps and then solve them all in parallel.

As an example of divide-and-conquer, consider the sequential mergesort algorithm. It takes n number of

input keys and returns the keys in sorted order. It works on splitting the keys to two sets of $n/2$ keys, recursively sorting each set, and merging the two sorted sets into a single set of sorted keys. The running time of the algorithm is given according to recurrence rule as follows:

$$T(n) = \begin{cases} 2T(n/2) + O(n), n > 1 \\ O(n), \qquad\qquad n = 1. \end{cases}$$

The above has solution of $T(n) = O(n \log n)$. Although not designed as a parallel algorithm, mergesort has some inherent parallelism since the two recursive calls can be made in parallel. This can be expressed as:

ALGORITHM: MERGESORT(A).

if($|A| = 1$) **then return A**

else

in parallel do

L:= MERGESORT($A[0..|A|/2)$

R:= MERGESORT($A[|A|/2 + 1..[A]]$)

return MERGE(L, R)

6.3.2 Sorted Array Search

In the off-line case, we can search faster, if we allow some time to preprocess the set and the elements can be ordered. Certainly, if we sort the set (using $O(n \log n)$ comparisons in the worst case) and store it in an array, we can use the well-known binary search. Binary search uses divide and conquer to quickly discard half of the elements by comparing the searched key with the element in the middle of the array, and if not equal, following the search recursively either on the first half or the second half (if the searched key was smaller or larger, respectively). Using binary search, we can solve the problem using at most $U_n = \lceil \log_2(n + 1) \rceil$ comparisons. Therefore, if we do many searches we can amortize the cost of the initial sorting.

On an average, a successful search is also $O(\log n)$. In practice, we do not have three-way comparisons; so, it is better to search recursively until we have discarded all but one element and then compare for equality. Binary search is optimal for the RAM comparison model in the worst and the average case. However, by assuming more information about the set or changing the model, we can improve the average or the worst case, as shown in the next sections.

6.3.2.1 Parallel Binary Search

Suppose now that we change the model by having p processors with a shared memory. That is, we use a parallel RAM (PRAM) model. Can we speed up binary search? First, we have to define how the memory is accessed in a concurrent way. The most used model is *concurrent read but exclusive write* (CREW) (otherwise it is difficult to know the final value of a memory cell after a writing operation).

In a CREW PRAM, we can use the following simple parallel binary search. We divide the sorted set into $p+1$ segments (then, there are p internal segment boundaries). Processor i compares the key to the element stored in the ith boundary and writes in a variable c_i a 0, if it is greater or a 1 if it is smaller (in case of equality, the search ends). All the processors do this in parallel. After this step, there is an index j such

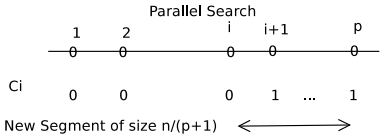

Figure 6.2: Parallel binary search.

that $c_j = 0$ and $c_{j+1} = 1$ (we assume that $c_0 = 0$ and $c_{p+1} = 1$), which indicates in which segment the key should be. Then, processor i compares c_i and c_{i+1} and if they are different writes the new boundaries where the search continues recursively (see figure 6.2). This step is also done in parallel (processor 1 and p take care of the extreme cases). When the segment is of size p or less, each processor compares one element and the search ends. Then, the worst-case number of parallel key comparisons is given by

$$U_n = 1 + U_{\frac{n}{p+1}}$$

for $U_i = (i \leq p)$, which gives $U_n = \log_{p+1} n + O(p)$. That is, $U_n = O(\log n / \log(p+1))$. Note that for $p = 1$, we obtain the binary search result, as expected. It is possible to prove that it is not possible to do it better. In the PRAM model, the optimal speedup is when the work done by p processors is p times the work of the optimal sequential algorithm. In this case, the total work is $p \log n / \log p$, which is larger than $\log n$. In other words, searching in a sorted set cannot be solved with optimal speedup. If we restrict the PRAM model also to exclusive reads (EREW), then $U_n = O(\log n - \log p)$, which is even worse. This is because, at every recursive step, if all the processors cannot read the new segment concurrently, we slow down all the processes.

Exercises

1. What do you understand by the work and depth of a parallel algorithm? How these are computed for multiprocesor systems?
2. What are the criteria for the design of a parallel algorithm? What stages / operations are retained from sequential algorithms and what not?
3. Find the work-done, depth, and parallelism in the work-depth models shown in figure 6.3 (a)-(d).

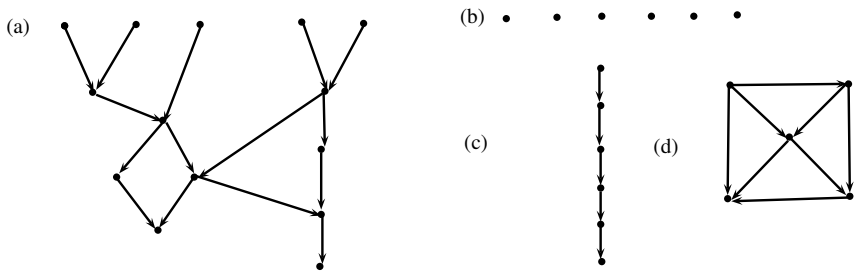

Figure 6.3: Work-depth models.

4. Construct a work-depth model structure such that the work is maximum and depth is minimum.
5. Construct a work-depth model structure such that the depth is maximum and work is minimum.

6. What are the commonly used global-operations in parallel algorithms? How are they implemented? Explain each in detail.

References

[1] Thomas H. Corman, et al., "Introduction to Algorithms, 3rd ed." *PHI*, 2009.

[2] Mikhail J. Atallah and Marina Blanton, "Algorithms and Theory of Computation handbook, Second Edition, Special Topics and Techniques", CRC Press, Taylor and Francis Group - A Chapman and Hall Book, 2010.

[3] Allen B. Tucker, Jr., "The computer Science and Engineering Handbook," *CRC Press*, 1997.