CPS222 Lecture: Parallel Algorithms Last Revised 4/20/2015

Objectives

1. To introduce fundamental concepts of parallel algorithms, including speedup and cost.
2. To introduce some simple parallel algorithms.
3. To overview architectures for connecting processors and their impact on algorithm design

Materials

1. Projectables of figures from Miller and Boxer: 5-16, 18, 19, 23
2. Projectable of merge tree
3. Projectable showing implications of Amdahl's law

I. Introduction and Fundamental Concepts

A. One common concern in computing is increased speed - often to allow bigger problems to be solved.

B. There are basically three ways to reduce the time taken to perform a given computation.

1. Use a better algorithm. However, for many problems it appears that we have already discovered the best possible algorithm (at least in terms of asymptotic complexity.)

2. Use faster hardware. This has been an area of steady progress that has produced dramatic results over time - e.g. in 1990 a 16 MHz CPU was considered fast; CPU's today have clock rates in excess of 2 GHz. However, increases in CPU speed have largely stopped in the last 10 years. Moreover, while CPU speeds were increasing steadily, other speeds were not - e.g. DRAM memory access times have changed very little in the same period (from perhaps 100 ns to 60-70 ns.)

3. Parallel processing.

   Moore's law - that the number of transistors that can be placed on a chip doubles every two years - has held true for almost 50 years and appears to be continuing. As hardware prices and sizes have plummeted, it has become increasingly possible to put multiple processors to work on solving a problem in parallel.

   This is the area we want to consider now: using multiple processors to work simultaneously on different aspects of a given problem, in order to reduce the overall time needed to generate a solution.

   We contrast this with sequential processing, in which all the processing is done by a single processor - which is what we have assumed in our discussion of data structures thus far.

C. Parallel processing comes in several forms

1. Almost all CPU's today are multi-core. While each core is a separate processor that can be executing in parallel with the other(s), all cores share a common memory system.

   (It is also possible to build a special-purpose machine with multiple processors chips sharing a common memory.)

   Either way, this model is called shared-memory multiprocessing, because all the processing elements share a common memory.

2. We can build clusters of processors in close physical proximity (often the same rack) - either using some sort of high-speed connection or a conventional network.

   This is called cluster computing.

   a) A cluster is designed to always have all the processors working together as a unit on a computationally-intense problem.

b) A cluster usually has one designated processor - called the head-node - that is responsible for communicating with the user and farming out the tasks to the other processors.

3. One can also engage processors at different physical locations to work together on a single task - communicating via a conventional network - though at other times these same processors may be working on their own or in cooperation with a different group of processors.

   This is called distributed computing or grid computing.

4. Finally, cloud computing services (e.g. Amazon's) can make large numbers of processors available for relatively low cost.

D. Metrics for Parallel Algorithms

1. When comparing sequential and parallel algorithms for the same task, there are two important measures we need to consider: speedup and cost.

   a) Speedup is defined as the ratio of the time taken by the sequential algorithm to the time taken by the parallel one.

   b) The cost of an algorithm is defined as the number of processors multiplied by the time taken by the algorithm.

2. We are particularly interested in algorithmic strategies that achieve linear speedup, in which using p processors for a parallel algorithm achieves speedup of p when compared to the sequential algorithm - i.e. the net cost is the same in both cases (or differs only by a small amount).

E. Two types of parallelism

It is sometimes helpful to distinguish two types of parallelism

1. In data parallelism, multiple processors are simultaneously applying the same algorithm to different portions of the overall data.

Example: this fall three sections of TGC met at 2:10 on MWF with different instructors, students, and rooms. All students took the same course, but the course was taught to several different groups of students (the "data") simultaneously.

2. In task parallelism, multiple processors are simultaneously performing different portions of the same overall task.

   Example: last fall a section of Calculus I met at the same time as a section of Calculus II - each with its a different instructor, students, and room. Each was performing a different part of the task of educating students in Calculus.

II. Why Learn about Parallel Programming?

A. There is a growing recognition in the CS Education community that parallel programming concepts need to permeate CS education, because effective use of parallel resources requires that one learn to "think parallel" as one develops solutions to problems.

B. There is a category of problems sometimes referred to as "embarrassingly parallel" in which there is a natural partitioning of the work that makes adapting a sequential algorithm almost trivial.

   Example: suppose we want to search a large database for a single entry that matches some search criterion involving values not used as the basis for a scheme like hashing or a B-Tree, so the only way to find the desired entry is to look at all of them.

1. A $\Theta(n)$ sequential solution might look like this:

```
for (i = 0; i < numberOfItems)
   if (item[i] matches the criteria)
      print the information associated with item i;
```

2. If we had some number of processors p available, we might partition the data among the processors in such a way that each processor ran the above loop on n/p items.

   a) Speedup?

      ASK

      n / (n/p) = p

   b) Cost?

      p * (n / p) = n - same as sequential algorithm

   c) Hence, this strategy achieves linear speedup

C. Of more concern, though, are problems where efficient parallelization of an algorithm is not obvious. These are the kinds of problems we want to think about today.

   Example: a very simple operation that illustrates this point.

   Suppose we had an array of n elements x[0] .. x[n-1], and we wanted to compute the sum of all the elements (perhaps because we wanted to compute their average.)

   1. The sequential algorithm for this is straightforward:

```
double sum = x[0];
for (int i = 1; i < n; i ++)
        sum += x[i];
```

      Obviously, this is a $\Theta(n)$ algorithm, which is the best we can hope to do since each element of the array must be added.

      If we regard each iteration of the loop as having a cost of 1, then the overall cost of this solution is $1 * \Theta(n) = \Theta(n)$.

2. What might a parallel version of this algorithm look like?

   ASK

   a) One approach would be to use n / 2 processors. On the first time step, each processor adds two of the numbers. On the second time step, half of the processors add their own sum to that produced by one of the other processors. On the third time step, a quarter of the processors add their own sum to that produced by one of the other processors that was active on the second step ...

      (1) What would the time taken by this algorithm be?

      ASK

      $\Theta(\log n)$

      For this reason, this algorithm is commonly called the *log-sum* algorithm. It represents the best time attainable for this task.

      (2) What would the speedup be?

      ASK

      $\Theta (n / \log n)$

      So, for example, if there were a million items the speedup would be about 1 million / 20 = 50,000

      (3) But what would the cost of this parallel algorithm be?

      ASK

      (a) The cost is now $n/2 * \log n = \Theta(n \log n)$

(b) This cost is much higher than the cost of the sequential algorithm - by a factor of (log n) / 2. So, for example, if there were a million items, the parallel algorithm would be about 10 times as costly.

(c) The reason for the higher cost is that many of the processors are idle on most steps - e.g. all the processors are used on the first step, only half on the second step, only a quarter on the third step ... on the last step, all but one processor is idle.

(4) Of course, using n/2 processors like this runs into other practical problems if n is large!

b) Could we discover a parallel solution whose cost is the same as that of the sequential one?

ASK

In this a case, such a parallel algorithm exists.

ASK

(1) Use n / log n processors, each responsible for log n elements of the array.

(2) For the first log n time steps, each processor sums the elements it is responsible for using the conventional sequential algorithm. (During this time, all processors are computing in parallel.)

(3) For the next log (n / log n) time steps, sum the partial results produced by the various processors using the log sum algorithm.

(4) Time taken by this algorithm?

ASK

log n + log (n / log n) = log n + log n - log(log (n)) = Θ(log n)

(5) Speedup?

ASK

Almost as good as before - to within a constant factor. (This algorithm does 2 x log n steps, the other log n steps)

(6) Cost?

ASK

n / log n processors busy for $\Theta(\log n)$ time = $\Theta(n)$ - the same as for the sequential algorithm (to within a factor of 2)!

(7) Actually, the speedup may not be as good or the cost as low depending on how the processors communicate.

In particular, if message passing is used, there can be a communication delay for each step in the log sum part.

However, if the processors use shared memory or a special purpose connection system, this will not be as much of a problem.

D. The previous examples bring out a crucial point. Often, after doing some computation in parallel on multiple machines, we need to somehow combine the results computed in parallel to get a single overall result. Discovering a way to do this final reduction efficiently may be the hardest part of the task.

1. In the case of the first example (the embarrassingly parallel one), the final reduction was trivial - the processor finding the desired item printed information about it, and the others did nothing.

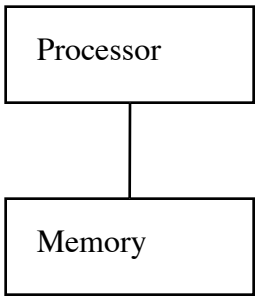
2. But in the case of the second example (summing the elements of the array), it was necessary to reduce the sums calculated in parallel to a single value, and this is where a significant part of the time was spent.

3. The log sum algorithm used in the example above can be used with any sort of binary operation that reduces a pair of values to a single summary value - including arithmetic operations such as + or * or operations like max() or min().

III. **Architectures for Parallel Computation**

A. The sequential model of computation we have been implicitly using all semester is called the RAM (random access memory) model. The corresponding model for parallel processing is called the PRAM (Parallel random access memory) model.

1. In the RAM model, a computer consists of a single processor connected to a memory system



a) Memory is considered to consist of M locations (numbered 0..M–1), each of which can hold a single primitive item of data. Any location in memory can be accessed in $\Theta(1)$ time.

b) The processor has a set of internal registers, each of which can hold a primitive item of data. In one time unit, the processor can perform a single basic operation involving its registers and/or a location in memory - e.g.

(1) Copying a value from memory to a register or from a register to memory.

(2) Performing an arithmetic or logical operation on two values.

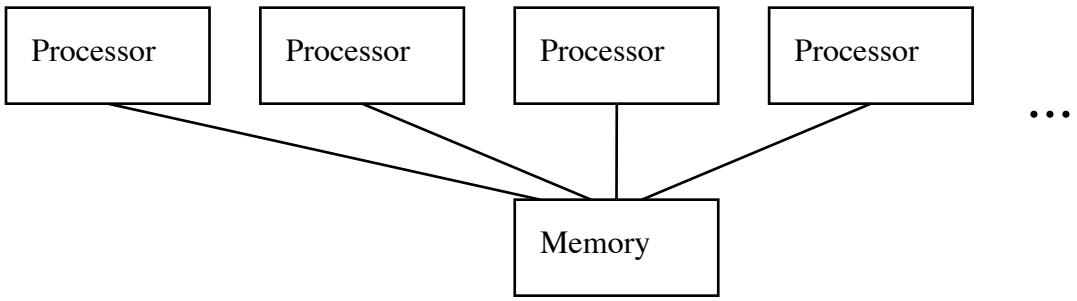
(3) Comparing two values.

etc.

c) Actually, the time taken for these various operations may vary depending on the operation, but for purposes of analysis we consider each to take unit time.

d) Algorithms for such a machine are called *sequential algorithms*, because the individual steps are performed one after another.

2. In the PRAM model, a computer consists of some number of processors, all connected to a single memory system.



a) The memory of the PRAM is similar to the RAM memory - but is accessible to all processors.

b) In one time unit, each processor can perform a single basic operation, as defined above. Thus, in each time unit, as many operations can be performed as there are processors.

While the basic PRAM model does allow all processors to access memory simultaneously, it may restrict multiple processors accessing the same location at the same time - where "same location" may be an entire block of memory cells, not just a single word.

There are three variants of the basic PRAM model, distinguished by what restrictions they place on access to the same location in memory by two different processors at the exact same time.

(1) EREW - exclusive read, exclusive write - only one processor may access a given item in any way at the same time. The algorithm is responsible for ensuring that two processors do not attempt to access the same item at the same time.

(2) CREW - concurrent read, exclusive write - any number of processors can read an item at the same time, but only one processor may write it at any time. The algorithm does not have to be concerned about multiple reads, but must ensure that two processors don't attempt to write the same item at the same time.

(3) CRCW - concurrent read, concurrent write - any number of processors can read or write an item at the same time. (Not yet achieved in practice.)

c) Algorithms for a PRAM machine are called *parallel algorithms*.

B. We have introduced the PRAM model as a basic model for designing parallel algorithms.

1. However, for any significantly large number of processors building a PRAM where all processors share the same physical memory is not feasible. Why?

   ASK

   There is a need to connect the processors to the memory in such a way as to support simultaneous accesses to the memory by *each* processor. The data path between the processors and the memory would become a bottle-neck that would prevent simultaneous access by all processors.

   (E.g. consider 1000's of Red Sox fans leaving Fenway park at the end of the game heading for 1000's of homes all over Eastern Massachusetts. Unfortunately, most of them have to pass through Kenmore Square, either by car or to take the T)

2. Instead, one of two approaches is used when a significant number of processors are needed (not an issue with multicore systems):

   a) It is possible to build a system in which each processor has its own, private memory, with a single shared memory being used only for values that need to be communicated from processor to processor. A processor only accesses shared memory when it has to. This is also called *shared memory multiprocessing*.

      (1) Example: in the second summation algorithm we discussed, each processor would keep the log n values it is responsible for in its private memory, and would write their sum to the shared memory. All of the values used by the log-sum portion of the algorithm would reside in shared memory.

      (2) Unfortunately, in this case the shared memory can still become a bottleneck that slows the computation down. (E.g. since all processors are doing the log-sum phase of the algorithm at the same time, they would all contend for the shared memory.)

   b) Another alternative is to have each processor have its own private memory, and use communication links between the processors to pass information back and forth. (E.g. during the log sum algorithm, a processor that needs to add its value to that computed by another processor would request the value from that processor over a communication link.) *This is called distributed memory multiprocessing*.

      (1) In the case of a cluster system, the connection may be a special purpose high speed hardware link.

      (2) But sometimes in a cluster, and always in a distributed system, this communication link will be a conventional network connection which, of course, entails greater latency for communication.

      (Note: the technical term for a processor and its private memory is processing element, but we commonly use the term processor to refer to the whole unit.)

(3) Of course, the need for using a communication link of any sort implies that reduction steps requiring communication between processors may take significantly more time than other steps than computational steps on a single processor.

Thus, one goal in formulating algorithms for distributed memory machines is minimizing the amount of inter-processor communication needed.

3. Nonetheless, the PRAM model is often used as a basis for designing and analyzing algorithms, even though building a PRAM for large number of processors is not practical.

(A multicore computer and some special parallel machines are truly PRAM, however).

C. Architectures for distributed memory systems

If distributed memory is used, we must now consider how the processors are connected to one another so that they can communicate when they need to - and how this affects the reduction portion of a parallel algorithm. (We'll use the log-sum algorithm for our examples, since it is so versatile.)

1. Basically, we can consider the interconnection structure to be a graph in which the processors are the vertices and the communication links are the edges.

2. We need to consider the following issues:

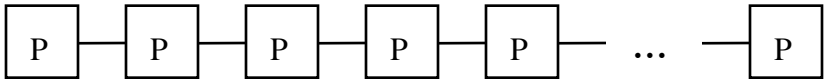
a) The degree of the graph - the number of processors each processor must connect to. The higher the degree, the more costly the machine is to build - indeed, for very high degree it will not be practical at all.

b) Communication diameter: this is the length of the shortest path between the most distant processors. This is critical, because it establishes a lower bound for the time complexity of any operation which involves combining information from all the processors' private memories in some way.

3. Some alternative structures:

a) The most flexible structure would be a complete graph, in which each processor is directly connected to each other processor. However, this rapidly becomes infeasible for a large number of processors, because each processor has degree #of processors - 1.

b) At the other end of the spectrum is a linear array in which each processor (save two) connects to two other processors.



However, this architecture suffers from a long communication diameter - if there are p processors, it takes $\Theta(p)$ time for information held by the first processor to reach the last one.

c) One more practical sort of structure is the mesh, in which each processor has up to four neighbors (some have less).

PROJECT: Miller and Boxer figure 5-16

Note that, for this structure, the mesh has $\sqrt{p}$ rows and $\sqrt{p}$ columns. The communication diameter (and therefore the maximum cost of communicating information from one processor to another) is $2 \times \sqrt{p}$

d) Another structure is a tree-like structure, in which each processor has up to three neighbors (its parent and its two children).

PROJECT: Miller and Boxer figure 5-18

Here, the communication diameter is 2 log p (any processor can communicate information to another by sending it up the tree to the nearest common ancestor, then back down the tree to the destination.)

e) A two-dimensional variant of the tree is a pyramid, in which each processor has up to 9 neighbors, and the communication diameter is log p. ($2 \times \log_4 p = \log_2 p$)

PROJECT: Miller and Boxer figure 5-19

f) Another interesting structure is the hypercube, in which each processor has log p neighbors, and the communication diameter is also log p.

PROJECT: Miller and Boxer figure 5-23

D. Note that we may use the PRAM model to help in deriving a parallel algorithm, but then we use some other architecture to actually implement it. This may require some careful thought to restructure the algorithm to ensure that needed information can be communicated between processors in the allotted time.

EXAMPLE: The second variant of the sum algorithm (the one with linear speedup), adapted to various connection architectures. Note that, in each case, we require $p = n / \log n$ processors.

1. Fully connected - obviously can still be done in $\Theta(\log n)$ time, since any processor can communicate with any other processor in $\Theta(1)$ time.

2. Linear array

a) We cannot perform the log-sum portion of the algorithm in $\Theta(\log n)$ time, since $\Theta(p)$ time is required for information from the left end to reach the right end - or from both ends to reach the middle. In this case, then, the minimum time is $\Theta(n / \log n)$, which for large n is greater than $\Theta(\log n)$.

b) However, we could still create a parallel algorithm for this configuration that gives significant speedup without increased cost, by using $\sqrt{n}$ processors, each responsible for $\sqrt{n}$ data items. During the first $\sqrt{n}$ time units, each processor adds the items it is responsible for. During the second $\sqrt{n}$ time units, the individual sums are passed down the line to a processor at one end, which adds them up as they arrive to produce the final sum. This yields a $\Theta(\sqrt{n})$ algorithm.

Speedup?

ASK

$\sqrt{n}$. For large n, this is not as good as n / log n - e.g. for n = 1024 the speedup is 32, not 102.

Cost?

$\sqrt{n}$ processors busy for $\sqrt{n}$ time units = $\Theta(n)$ - same as the sequential algorithm. (Linear speedup)

3. Mesh

a) Again, cannot perform the log-sum portion of the algorithm in $\Theta(\log n)$ time, because a minimum of $\Theta(\sqrt{p})$ time is needed for information from one corner to reach the other, or from two corners to reach the middle. In this case, then, the minimum time is $\Theta(\sqrt{(n / \log n)})$, which for large n is greater than $\Theta(\log n)$.

b) Using the same idea as for the linear array, we could build a mesh of $n^{2/3}$ processors, arrange as an $n^{1/3}$ x $n^{1/3}$ mesh, with each processor responsible for $n^{1/3}$ data items. During the first $n^{1/3}$ time units, each processor adds its data items; during the second $n^{1/3}$ time units the sum of each row is accumulated, and during the final time units the overall sum is accumulated, yielding a $\Theta(n^{1/3})$ algorithm.

Speedup?

ASK

$n / n^{1/3} = n^{2/3}$. For large n, this is not as good as n / log n speedup - e.g. for n = 1024, this is about 100 - almost as good as what we obtained before. (For larger n, the earlier algorithm would yield significantly better results.)

Cost?

$n^{2/3}$ processors busy for $n^{1/3}$ time units = $\Theta(n)$ - same as the sequential algorithm. (Linear speedup)

4. A tree could execute the original log n time algorithm we developed for the PRAM, since its communication diameter is $\Theta(\log n)$.

5. A pyramid could likewise execute the original log n time algorithm.

6. A hypercube could likewise execute the original log n time algorithm

7. We should not assume from the above analysis that the tree, hypercube and pyramid architectures are better than the mesh architecture. There are other algorithms which adapt just as well, or even better to the mesh architecture, than they do to a tree, pyramid, or hypercube. (Indeed, since the mesh has a matrix-like structure, one should expect it to be especially well suited to matrix operations of the sort that occur frequently in modeling physical systems.)

IV. **Parallel Sorting**

A. As an example of parallel thinking, let's think how we could parallelize the task of sorting.

1. Earlier in the course, we considered the sequential merge sort algorithm, which takes $\Theta(n \log n)$ to sort n items.

2. Suppose, now that we had some number of processors p available to perform the sorting task. (For simplicity, assume p is a power of 2 - though that is not necessary.) How could we use these effectively?

   ASK

   a) We could partition the original data among the processors, so that each processor is responsible for simultaneously performing an initial sort on n / p items using an appropriate sequential sorting algorithm (such as quicksort).

   b) Then use an approach similar to the log-sum algorithm to merge the results to produce a sorted final result

      PROJECT Merge tree

   c) Analysis:

      (1) Each of the initial sequential sorts would require $\Theta(n/p \ (\log n/p))$ time. Since these sorts could be done in parallel, the total time for this step is the same.

      (2) We can do all the merges on each level of the merge tree in parallel - so the first level involves $\Theta(2n/p)$ work, the second level involves $\Theta(4n/p)$ work ... and the last level involves $\Theta(pn/p) = \Theta(n)$ work. The total time for the merging is

      $$2n/p + 4n/p + ... + pn/p = 2n$$

(3) Hence, the overall time is $\Theta((\log(n/p) / p + 2)n)$

(a) If p is large compared to log n/p, the merging time dominates, and we have a $\Theta(n)$ sorting algorithm!

(b) If, more realistically, p is less than log n, then the time is dominated by the initial sorting. In this case, we get speedup of

$$\frac{n \log n}{n/p \log (n/p)}$$

which if p is small compared to n, is approximately p. Moreover, the cost in this case is the same as for the sequential sort, so we have linear speedup.

V. MapReduce

A. One widely-used approach to developing parallel solutions to problems is an approach known as MapReduce.

1. This approach was developed by Google for indexing web pages, and is now used for many of their "big data" applications.

2. A freely-available version of the basic MapReduce software known as hadoop - originally developed by Yahoo - is available from the Apache foundation.

B. MapReduce is designed to be used on a cluster system. The overall task is divided into two parts: mapping and reducing, with a shuffling step in between. Each part is parallelized.

C. Consider the following simple example: we have a large corpus of documents (perhaps 1000's). We want to generate a list of all the words occurring in these documents, together with a count of the number of times it appears.

1. In the mapping phase, we generate a set of key-value pairs - with the keys being the words and the values being a count of the number of times the word occurs. This can be parallelized by assigning different subsets of the documents to different processors.

2. Of course, to get our final result we need to combine the results from all the subsets of the original documents. For example, one processor might discover that the word "think" occurs 37 times in its subset of documents; another might discover that "think" occurs 12 times, and a third may discover that it occurs 28 times. What we want, in the end, is to know that the word occurs a total of 77 times. This is the task of the reduce phase.

   In the reduce phase, all of the key value pairs for the same key are assigned to the same processor, which is responsible for reducing the values to a single value (in this case by adding them together). Parallelism here is achieved by having different processors responsible for different keys.

3. The shuffling phase in between is responsible for sending all the key value pairs generated by different mappers for the same key to the same reducer.

VI. **Amdahl's Law**

A. One final note - there is an upper-limit to how much speedup is possible for a given task using parallelism.

1. In general, any task consists of some portion which is inherently sequential - i.e. step depends on the result of the previous step.

2. It turns out that the potential speedup for a given task using parallelism is strongly dependent on how much of the task is parellelizable and how much is not. A formula known Amdahl's law says that in a task where S is portion of the total time needed for the task that is inherently sequential and P is the portion of the

total time that is parallelizable (so S + P = 1), then the speedup resulting from using n processors will be

$$\frac{1}{(1 - P) + P/n}$$

Example: Suppose 95% of the time needed for a given task is spent doing work that can be parallelized but 5% is spent on inherently sequential work. If the parallelizable part is speeded up by a factor of 10 (say by using 10 processors), then the overall speedup is

$$\frac{1}{(1 - .95) + (.95)/10} = \frac{1}{.05 + .095} = 6.9$$

But now suppose 50% of the time needed is spent doing work that can be done using parallelism, and 50% of the time is spent on inherently sequential work. If the parallel part is speeded up by a factor of 10, the overall speedup is only

$$\frac{1}{(1 - .50) + (.50)/10} = \frac{1}{.50 + .05} = 1.82$$

B. A corollary is that the maximum possible speedup for the task (obtained if an infinite speedup were applied to the parallelizable portion) is

$$\frac{1}{(1 - P)}$$

Example: for the first case cited above, the maximum possible speedup is 20:1, which would be achieved if the time for the non-parallelizable portion remained the same, while the time for the parallelizable part dropped to 0. But in the second case, the maximum possible speedup is only 2:1.

C. The implications of Amdahl's law can be seen graphically.

PROJECT: Graph showing Amdahl's law results for various values of P