NOTES ON PARALLEL ALGORITHMS

A parallel algorithm is a set of instructions for processes that run on several processors simultaneously. We suppose that the processing occurs in time-steps, during each of which the every processor works on its own alone except perhaps to access a common memory unit or to send or receive a single message from another processor.

The two standard broad categories of such architectures are (i) the SHARED MEMORY architecture:

```
   o           o
    \         /
     \_____ /
o----|     |----o
 ... |  M  |...
o----|     |----o
      -----
        |
        |
        o
```

Here each o stands for a processor and M for a memory that each processor can access; there is no communication between processors except by their reading and writing to M.

and (ii) the NETWORK architecture:

```
O----O----O...
|    |    |
O----O----O...
.    .    .
.    .    .
```

Here various processors each have their own local memory and communicate only by sending messages to one another along the indicated paths. Any graph arrangement is possible, where nodes are processors and edges are paths.

The graph shown is called a mesh or grid; other standard arrangements include the following

a. `O----O----O...`   linear array

b. ring (a linear array joined at the ends)

c. torus (a grid wrapped at the edges like a donut)

d. hypercube (a cube and its generalizations)

e. complete graph (every processor joined to every other processor)

A network of McCullough-Pitts neurons can be thought of as a parallel network, but with the limitation that "messages" consist only of "on" or "off", whereas in general the network architecture allows arbitrary messages; and also the computation performed by a single Mc-P unit is always that of summing weighted inputs and comparing to a threshhold, rather than any computable function at all.

AN EXAMPLE

If we wish to add a list of n numbers, we can add them one by one, in sequential fashion, using a single processor: add the first two numbers, then add the third to the previous sum, then the fourth, and so on. This is a non-parallel algorithm.

EXERCISE: prove that it takes >= n-1 steps to sum n numbers sequentially. Hint: use induction. Assume summing two values (and storing the answer) is the only available operation.

However, we can alternatively utilize, say, n processors, and have many (or even all) of them perform additions at the same time, to get the total sum much faster, if we are careful about it such as: add the first pair of numbers, and also second pair, the third pair, etc, simultaneously, each processor adding one of the pairs; this is done in a single time-step, and the result is a list of n/2 sums of pairs. Next (in the next time-step) we use half the processors to add pairs of the previous sums, one per processor, yielding a list of n/4 sums (each of which is the sum of four of the original numbers). We continue until at last we are adding just two previous results, each of which is the sum of half the original numbers.

For instance, if n=8, then after the first step, we have four sums: a+b, c+d, e+f, and g+h. After step two, we have two sums: a+b+c+d and e+f+g+h. After a third step we have only a+b+c+d+e+f+g, i.e., we get the total sum of the eight numbers in three steps of simultaneous (‘‘parallel’’) processes.

To be more precise, we can employ the shared memory architecture with p processors, and p numbers stored in memory locations A[1]...A[p]. Every processor r (where r=0,1,2,...,p-1) runs the following code in parallel (in clocked steps) with the others:

```
for i:=0 to lg(p) - 1 do
   if r - 2^i >= 0 then
      A[r] := A[r - 2^i] + A[r]
```

For p = 8, we can compare this to the sequential approach: we would take n-1 = 7 steps to sum all the numbers, as opposed to 3 -- which is lg 8, and not by accident: each parallel step halves the number of results, so in lg n steps we are down to 1 result. Of course, we have to know where to look for the answer: A[p-1].

Note that the above parallel algorithm never has two processors access the same memory location at the same time. But some algorithms do require so-called concurrent reads or writes to the same location. This leads to four subcategories of shared-memory architectures:

EREW: exclusive read, exclusive write
CREW: concurrent read, exclusive write
ERCW: exclusive read, concurrent write
CRCW: concurrent read, concurrent write

In EREW, for example, it is not allowed for two processors to attempt to access the same location at the same time for reading, nor for writing. There are pros and cons of each of these, which we will not go into here.

Can we also perform the above parallel addition algorithm with a network architecture? Yes and no. In a network, data has to be passed along from processor to processor, it cannot be left in a shared memory location. Thus processor r cannot simply add the value it has to that of processor r-1; first r-1 has to send its value to r. This is not so bad, as long as r and r-1 have a wire connecting them. But at the next step, when r has to get the value at r-2, there also should be a wire there, or else r-2 has to send to r-1 and then r-1 to r, etc. If the network is complete, this is ok. Another interesting case is the hypercube, where all the needed wires are in place for addition algorithm. For n=8 this is just a cube, with the usual 8 vertices, and 12 edges:

```
000--------001
  |\       |\
  | \      | \
  |100--------101
010--|- - -011|
   \ |      \ |
    \|       \|
   110--------111
```

The eight vertices (processors) have been labelled in binary, from

000=0 to 111=7. Then vertex r-2^i turns out to be just one edge away from vertex r, in all cases where r getting data from vertex r-2^i is critical for the result. For instance, 7 - 2^0 = 6, 7 - 2^1 = 5, and 7 - 2^2 = 3, and all three vertices 6 (110), 5 (101), and 3 (011) are adjacent to 7 (111). Note that adjacency (connected by a single wire) means simply differing (by a 0 or 1) in exactly one bit.

The idea is this: data is sent from left to right in the picture (ie, each even vertex, on the left, sends its data to the one-higher-numbered odd vertex to the right), and the sent data is added to whatever is already there on the right. Then the data on the top (right) is sent down to the bottom (ie to vertices that are two-higher-numbered). At this point all the original data has been added into two places: vertices 011 (data from the back four) and 111 (data from the front four). Finally, 011 sends it data to 111 where the final sum is computed.

This process can also be thought of in terms of bits: first we add data in processors with low-order bit 0 into processors with low-order bit 1. Then the same for middle-order bits; and then for high-order bits.

The above shared memory summing algorithm can now be modified to work for the (hyper)cube; each processor r runs the code:

```
for i:=0 to lg(p) - 1 do
   if r + 2^i <= p-1 then
      send own value to processor r+2^i
   if r - 2^i >= 0 then
      increment own value by data from processor r - 2^i
```

Here some of the sendings take more than one wire (edge) to get where they need; but these cases are ones that do not enter into the actual result of the total sum. For instance processor 1 sends its data to processor 2 but along more than one wire; such cases are ones that the algorithm performs but that are extraneous, not part of the idea outlined above, for moving data from even to odd bit values.

A more elegant version avoids the slow sendings altogether and just does what we want; each processor r runs the code:

```
for i:=0 to lg(p) - 1 do
   if the i-th bit of r is 0
      then send own value to processor r+2^i
      else increment own value by data from processor r-2^i
```

SORTING

Each type and subtype of architecture has pros and cons with respect to sorting. Here we just give a tiny glance at one example.

There are algorithms for the hypercube (and also for shared memory models) that merge two lists each of size $n/2$ in $\lg n$ comparison steps. Using this (as a black box), we can sort in parallel, by cutting the list in half, sorting the two halves independently in parallel, and merging the two lists together.

Recall the sequential runtime recurrence for merge sort: $T(n) = 2T(n/2) + n - 1$. This is now replaced by the parallel recurrence

$$T(n) = T(n/2) + \lg n, \quad \text{[where } T(1)=0\text{]}$$

Notice that although there are two computations each of which takes time $T(n/2)$, they are done during the *same* time now, so there is no 2 in front.

Solving the recurrence we get

$$T(n) = \frac{(\lg(n))(\lg(n) + 1)}{2}.$$

EXERCISE: Prove this.

SEARCHING

Parallel *searching* a linear array is no better than ordinary sequential search for $n=p$; this is mainly because the processors need to communicate with one another in order to (i) know what to search for, and (ii) to send the answer to a predetermined "result" processor. Since there are few connections in a linear array, this means lots of processor time is used just in sending messages back and forth, to and from other processors, rather than searching. For $n>p$, one can search in $O(n/p + p)$ time by having each processor search its fair share in $O(n/p)$ time and then communicate the results in $O(p)$ time.

On the other hand, we can do fast parallel search of an unordered list of p elements on a shared memory p-processor CREW (constant time), assuming that the item to be found occurs only once. We simply specify the value to be found in a common memory location that all processors read in parallel, and each then checks to see if it has that value and if so write to a pre-assigned answer location. If the value can occur more than once we can use a CRCW instead but the details are slightly more complex. This takes constant time for n=p, and can be generalized to O(n/p) for n>p.

If we use an EREW instead, we first need to broadcast the sought value so all processors know what to look for (if we let them read it, it will be sequential and take time p); the broadcast can be done in lg p steps: first a single processor reads once and writes a copy somewhere. Then it and one more read and write two more copies, etc. So the search can be done in O(n/p + lg p) time.

For an *ordered* list we can use a CREW to search in (lg n)/lg(p+1) time. In one step compare the item to be searched to p evenly spaced items in the ordered list. Determine which interval of size n/(p+1) the item belongs in and broadcast this information to all of the processors (by putting it in a common location). Continue iteratively on the new interval. The problem size decreases by a factor of p+1 at each iteration.

SOME ADDITIONAL MEASURES

Associated with a network is the notion of diameter: the largest "distance" between two processors, where distance between pi and pj is measured as the number of "hops" (wires) from processor to processor needed to communicate between pi and pj. [In a shared memory architecture this is simply 2.] Here are some basic facts for p processors and various networks:

Network	Wires	Diameter
linear array	p-1	p-1
ring	p	|_ p/2 _|
equilateral grid	kp*	2[sqrt(p) - 1]
hypercube	(p/2) lg p	lg p
complete	p(p-1)/2	1

* This is correct to the high-order term in p, where k is the dimension of the grid. The exact formula is kp - kp^((k-1)/k).

EXERCISE: Prove that the square grid of dimension k has exactly $kp - kp^{(k-1)/k}$ wires.

The diameter is a measure of how easy it is for processors to communicate. Clearly the complete graph network comes out best in that sense. On the other hand, it has very many wires: $p(p-1)/2$.

EXERCISE: Prove that there are $p(p-1)/2$ wires in a complete network.

The hypercube is a good compromise, having low diameter and low number of wires $(p/2)\lg p$.

EXERCISE: Prove that there are $(p/2)\lg p$ wires in a hypercube.

EXERCISE: The hypercube is a special case of the equilateral grid, where the dimension $k = \lg p$. Show that the grid and hypercube formulas for wires and diameters agree in this case.

Finally, we consider the "power of parallelism" in general. Ie, how much advantage can be gained by the use of parallel processing?

The SPEEDUP of a parallel algorithm over a sequential algorithm, for the same computational problem, is defined as follows: For a problem of size n (i.e., n data items) and for p processors used in parallel, it is the function

$$S_p(n) = t(n) / T_p(n)$$

where $t(n)$ is the runtime for a sequential solution and $T_p(n)$ is that of a parallel solution. [We usually have in mind ‘‘good’’ algorithms, i.e., ones that run as fast as possible in the worst-case, both for $t(n)$ and for $T_p(n)$, although in practice we may not know precisely what those are.]

For instance, in the above summing example, the speedup is $(n-1)/\lg n$, where $p = n/2$. If we decide to buy 4 processors ($p=4$) then we want to get our money's worth. The speedup for addition of $2p = 8$ numbers as above will be $7/3 = 2.333$. This is not very good (only a slight saving in runtime) for a considerable cost of extra processors. But if we buy one million processors, and add two million numbers, the speedup is (roughly) 1,000,000/21, which is about 50,000 times faster than the sequential method. So if we were to routinely add millions of numbers, and if great speed were crucial, then it *might* be worth it to buy millions of processors. However, most fast modern sequential machines can add as many as one billion numbers in a second. Thus it would make sense to pay the expense of a parallel machine for addition only if it were necessary to get such sums in far less than one second.

SpeedUp Theorem: 1 <= S_p(n) <= p

Proof: Since we can always let p-1 of the processors do nothing, then we can have a parallel algorithm behave just like a sequential one (really using just one processor -- imagine the others are set to work doing irrelevant things, if you prefer that) and so T_p(n) is always at least as fast as t(n). But recall that ‘‘at least as fast as’’ means ‘‘at least as small as’’, when we are talking about runtime; so T_p(m) <= t(n), and thus 1 <= S_p(n). This proves one of the two inequalities.

For the other inequality, note that we can have a single processor do all the work of the p processors, but (in general) by taking more time: for each time-step used by the parallel algorithm with p processors, we can do the same with one processor in p steps. Thus in p times as long, a single processor can simply carry out sequentially all the activities of p processors. So a parallel machine with p processors can never be more than p times faster than a suitable sequential machine solving the same problem.

We define the EFFICIENCY of a parallel algorithm as the speedup divided by the number of processors. So the maximum efficiency is p/p = 1. Efficiency measures how well each processor is used (as opposed to having them doing little that is useful). So, for the example of adding two million numbers on a million processors, the efficiency is approximately 50,000/1,000,000 = 1/20, which is not particularly good.