CS170 – Spring 2010 – Lecture Notes - Jan 27

1 Parallelism and Divide and Conquer

1.1 Motivation for Parallelism

Parallelism is the execution of more than one operation at a time. This has been done for some operations in computers for a long time. Some examples are (1) propagating carry bits in the addition of integers, (2) memory accesses overlapping with arithmetic operations (so-called "prefetching"), and (3) pipelines for instruction execution that break a complicated process into phases (fetch the instruction, decode the instruction, fetch the operands from memory, execute the instruction, store the results in memory) and execute them in an assembly-line fashion, with phases of consecutive instructions being done in parallel by different parts of the assembly line.

What these kinds of parallel operations had in common was that they happened invisibly to the programmer, who could write conventional sequential (i.e. non-parallel) programs, and let the hardware worry about such parallelism automatically. This of course made it easy on the programmer, who got the benefit (speedup) of parallelism without even thinking about it. It also made it possible for computer hardware manufacturers to maintain the promise of Moore's Law, which had led users to expect their computers to double in speed every 18 months or so (without having to change any software!).

But computer hardware manufacturers can no longer continue on this path of doubling speed without the user having to change their programs. This is both because they can't make individual processors faster by running them faster (they would get too hot and break), and because they can't find enough operations in the user's programs to do automatically in parallel anymore. So manufacturers are changing their designs to build parallel computers, with multiple CPUs on a chip (multicore, GPUs, ...) that will have to be programmed in an explicitly parallel fashion. In other words, user programs will have to explicitly control multiple processors, finding enough work for them all to do.

This is a big change in the way computers have been programmed. Up until now, relatively few programmers have programmed parallel computers, which were built to solve the largest and most demanding computing problems in science, engineering, defense, and similar fields. (See www.top500.org for historical lists of the 500 largest computers in the world, and how many parallel processors they use. The world record in late 2009 is a 224,162 processor Cray XT5 at Oak Ridge National Lab, with a peak speed of 2.33 PFlops ("Peta Flops", or "Quadrillion

Floating Point Operations per second", where a "floating point operation" is the addition or multiplication of 2 real numbers). The benchmark used to rank these 500 fastest machines is how fast they can solve a large system of linear equations using Gaussian Elimination. It turns out that the inner loop of Gaussian Elimination spends most of its time doing matrix-multiplication, which we will also talk about in this class.)

Now, the expectation is that to keep up with Moore's Law, desktops and laptops will have 2 processors this year or next, 4 processors the year after that, 8 the year after that, and so on. This means that programmers will need to learn how to program them, and this means that you need to learn what algorithms will work in parallel.

You may well ask why compilers can't solve this problem, automatically converting a sequential program into one that uses multiple processors. Indeed, this has been an ongoing part of compiler research for many years, and works for certain kinds of programs, but not nearly enough.

To learn more about parallelism, there is a large research project at Berkeley called ParLab, about which you can learn more at parlab.eecs.berkeley.edu.

1.2 Models of Parallelism

To talk about parallel algorithms, we need a simple model of a parallel computer, what operations it can do in parallel, and how much time they take (so we can count the right operations in our O() analyses). There is a big variety in parallel computers, ranging from your laptop with a modest amount of parallelism, to graphical processing units (GPUs) designed to make graphics and games go as fast as possible, to clusters of dozens up to thousands of standard computers connected over a standard network (how Google works), to "supercomputers" like the Cray XT5, to the "true" largest computer in the world – all the computers connected to the Internet. The record so far in using the Internet is "SETI@Home", where people volunteer their computers to analyze radiotelescope data in the Search for ExtraTerrestrial Intelligence (see setiathome.berkeley.edu). In early 2010, over 1M people had volunteered their computers to this cause, and many others like protein folding for medical research, climate modeling to understand global warming, finding enormous prime numbers, and so on (see setiathome.berkeley.edu or boinc.berkeley.edu).

In this class we will concentrate on a simple model appropriate to the modest parallelism of laptops, the *shared memory model*. In this model there are multiple independent processors all running their own programs. But they all share the same physical memory, so that if 2 or more processors execute the instruction "load location 3", they get the same value from the same location in memory. If 2 processors execute the instructions "store the value 1 to location

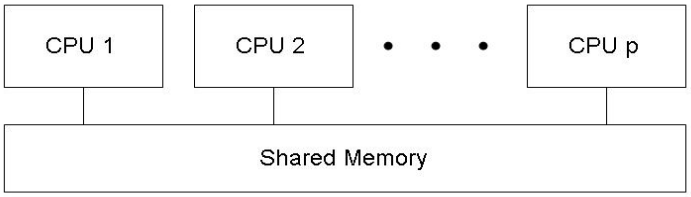

Figure 1: Shared Memory Parallel Computer

3" and "store the value 2 to location 3", then the memory will execute these in some order, and eventually location 3 will contain either 1 or 2 (rather than 1.5 or something else). (Like a sequential processor, each processor in the parallel computer will have its own local cache, so repeated reads of the same location are faster. As many processors load and store location 3, rather complicated hardware will make sure that the value of location 3 in the different caches is kept consistent, since otherwise very strange bugs would occur. How this hardware works is the topic of a different course.)

1.3 Reduction Operations

We begin with the simplest nontrivial operation: adding $2p$ numbers using p processors. We assume $p = 2^k$ for simplicity. Figure 2 shows the idea: At step 1, each processor adds a different pairs of numbers, "reducing" $2p$ addends to p. At step 2, $p/2$ processors add different pairs of numbers, reducing p addends to $p/2$. At step 3, $p/4$ processors produce $p/4$ addends. At step j, $p/2^{j-1}$ processors produce $p/2^{j-1}$ addends until at step $k = \log_2 p$, one processor produces the final sum. Thus we can add $2p$ numbers on p processors in $\log_2 p$ steps.
We use the the same binary tree pattern to solve the more general problem of adding n numbers on p processors: For simplicity we assume $p|n$ and $p = 2^k$.

1. Each processor independently and simultaneously adds n/p different inputs, yield a single sum. This is done sequentially by each processor, taking n/p additions.

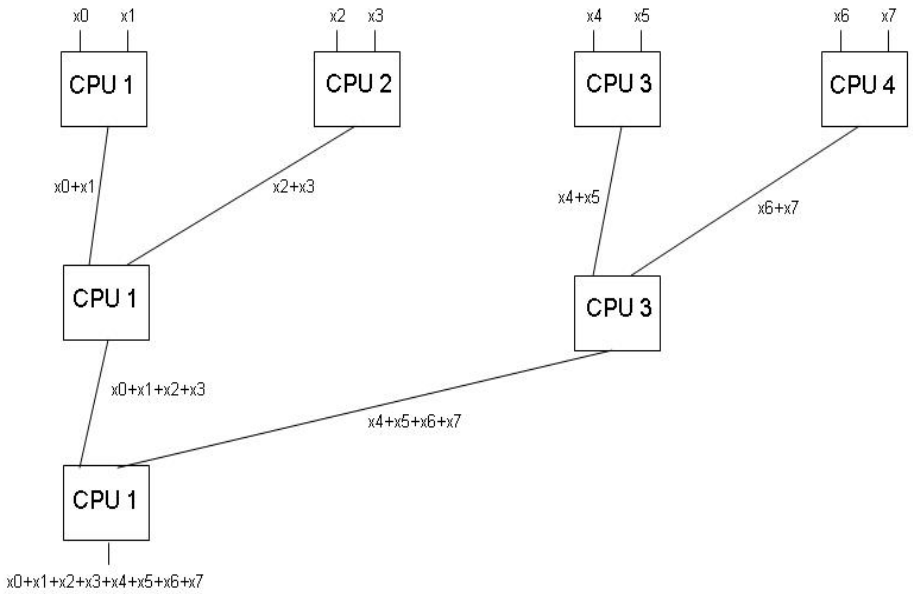

Figure 2: Adding 8 numbers on 4 Parallel Processors

2. $p/2$ processors reduce the resulting p sums to a single sum in another $(\log_2 p) - 1$ steps, using a binary tree as above.

Thus the total cost is $O(n/p + \log_2 p)$.
Note that the same algorithm works not just for adding n numbers, but for taking their product, maximum or minimum. Indeed, all that we ask for correctness is that

$$((x0 + x1) + x2) + x3 = (x0 + x1) + (x2 + x3)$$

i.e. that we can put the parentheses in differently (the left side corresponding to the sequential algorithm, and the right to the parallel). In other words, any *associative* operation can be done this way. Note that we do not need commutativity, i.e. it is not necessary that $x0 + x1 = x1 + x0$, so that associative operations like matrix multiplication (which is not commutative!) can be done this way too.
Google uses a version of this operation for much of its data processing work (to learn more, try googling "MapReduce").

1.4 Parallel Divide-and-Conquer

Now we show how divide and conquer algorithms may be naturally parallelized. We start with summing n numbers on p processors, just to illustrate the process of making a sequential divide-and-conquer program parallel. We assume n is a power of 2 for simplicity of presentation.

```
function sum(a(1..n)) ... sequential sum of n numbers
    if (n=1) return a(1)
    x = sum(a(1..n/2))
    y = sum(a(n/2+1..n))
    return (x+y)
```

The cost of this algorithm is T(n) = 2T(n/2) + 1 = O(n) as expected. Here is the parallel version:

```
function psum(a(1..n)) ... parallel sum of n numbers
    if (n=1) return a(1)
    do in parallel {
        x = psum(a(1..n/2))
        y = psum(a(n/2+1..n))
        }
    return (x+y)
```

The syntax "do in parallel" means that each of the following lines in braces {} will be executed on another processor, if available, and otherwise on the same processor. How this is implemented is not important for this class, but you can imagine a queue of tasks (subroutines to execute) sitting in memory, that idle processors examine to get work to do, and to which busy processors might add work to do. It leads to the same kind of binary tree of processors working on adding different parts of the input a(1..n) that we drew earlier.
The cost of parallel divide-and-conquer summation is then T(n) = T(n/2) + 1 (assuming we have enough processors (p $\geq$ n) to execute each pair of subroutine calls in parallel), which is T(n) = $\log_2$n as expected.
Now let's try quicksort:

```
function qsort(a(1..n)) ... sequential quicksort of n numbers
    if (n=1) return a(1)
    pick splitter s randomly
    [S_<, S_=, S_>] = divide(s,a(1..n))
    qsort(S_<, S_=)
    qsort(S_>)


function pqsort(a(1..n)) ... parallel quicksort of n numbers
    if (n=1) return a(1)
    pick splitter s randomly
    [S_<, S_=, S_>] = divide(s,a(1..n))
    do in parallel {
        pqsort(S_<, S_=)
        pqsort(S_>)
        }
```

The best we can hope for is a perfect parallel speedup, that p processors will sort p times as fast as 1. Unfortunately, pqsort doesn't come close, because the call to divide(s,a(1..n)) is still done on one processor and takes times $O(n)$.
To fix this, we need to divide differently.

1.5 Sample Sort

To sort faster, we will divide into p pieces, not 2, by taking a larger random sample of a(1..n) (not just a single random splitter s), and using this sample to split. The idea is that if we

pick a large enough random sample, and sort it on one processor in order to split it perfectly, this will do a good job of splitting all of a(1..n). Of course we can't pick too large a random sample, because then sorting it will be a bottleneck. So for now we let m denote the size of this random sample, and show how to choose it later.

function sample_sort(a(1..n)) ... parallel sort of n numbers

(1) processor 1 picks m random entries of a(1..n) and qsorts them

(2) from this sorted list, call it $s_1 \le s_2 \le \cdots \le s_m$,
processor 1 chooses p-1 splitters $s_{\frac{m}{p}}, s_{\frac{2m}{p}}, ..., s_{\frac{(p-1)m}{p}}$ which split $s(1..m)$ into p equal pieces and a(1..n) into p nearly equal pieces with high probability

(3) each processor divides its n/p entries of a(1..n) into p groups using the p-1 splitters, and stores each group together with with the corresponding groups from the other processors

(4) each processor qsorts one group of n/p values

Now we compute the cost of this algorithm. Steps (1) and (2) are done by processor 1 only, costing $O(mlog_2 m)$ and $O(p)$ respectively. Step (3) is done by all processors, costing $O(\frac{n}{p})$. Step (4) is done by all processors, costing $O(\frac{n}{p}\log_2 \frac{n}{p})$. Altogether the cost is $O(m \log_2 m + \frac{n}{p}\log_2 \frac{n}{p} + p)$. We want to choose m as large as possible to get good splitters, but we do not want it to be more expensive than other parts of the computation. Clearly, the choice $m = \frac{n}{p}$ is as large as we can make it, leading to a total cost of $O(\frac{n}{p}\log_2 \frac{n}{p} + p)$. Finally, we ask how many processors p we can productively use to sort. Since $\frac{n}{p}$ decreases as p increases, making p too large will not help. We again ask how large p can be without being dominant, and roughly this means $\frac{n}{p} \ge p$ or $n \ge p^2$ or $\sqrt{n} \ge p$. In practical situations, with thousands or millions of numbers to sort and many fewer processors, this is easily satisfied.