Analysing Parallel Algorithms

We begin by reviewing the standard framework for sequential algorithm analysis. We then consider the complications introduced by the introduction of parallelism and look at some proposed parallel frameworks.

Analysing Sequential Algorithms

The design and analysis of sequential algorithms is a well developed field, with a large body of commonly accepted results and techniques. This consensus is built upon the fact that the methodology and notation of *asymptotic* analysis (the so-called "big-O" notation) deliver results which are applicable across all sequential computers, programming languages, compilers and so on. This generality is achieved at the expense of a certain degree of blurring, in which constant factors and non-dominating terms in the analysis are simply ignored. In spite of this, the approach produces results which allow useful comparisons of the essential performance characteristics of different algorithms which are reflected in practice when implemented on real machines, in real languages through real compilers. For example, mergesort with its $\Theta(n \log n)$ run time is (in the worst case) an asymptotically better sorting algorithm than insertion sort ($\Theta\left(n^2\right)$) on any normal sequential machine (although the actual problem size at which the dominance becomes apparent *will* vary from implementation to implementation). Underpinning this work is the "Random Access Machine" (RAM) model which is an abstraction of the essential capabilities and cost characteristics which unite all sequential machines. The notation allows the description of "upper bounds" (with $O()$), "lower bounds" (with $\Omega()$) and "tight bounds" (with $\Theta()$) on the behaviour of functions representing the time or space requirements of an algorithm as its input problem size grows.

If you are unfamiliar with the big-O notation you should consult any text book on algorithms from the reserve section of the library (for example, Introduction to Algorithms by Cormen et al).

Analysing Parallel Algorithms

The sequential world benefits from a single universal abstract machine model (the RAM) which accurately (enough) characterizes all sequential computers and from a simple criterion of "better" for algorithm comparison ("less is better", usually of run time, and occasionally of memory space). Thinking parallel, we immediately encounter two complications.

Firstly, and fundamentally, there is no commonly agreed model of parallel computation. The diversity of proposed and implemented parallel architectures is such that it is not clear that such a model will ever emerge. Worse than this, the variations in architecture capabilities and associated costs mean that no such model can emerge, unless we are prepared to forgo certain tricks or shortcuts exploitable on one machine but not another. An algorithm designed in some abstract model of parallelism may have *asymptotically different* performance on two different architectures (rather than just the varying constant factors of different sequential machines).

Secondly, our notion of "better" even in the context of a single architecture must surely take into account the number of processors involved, as well as the run time. The trade-offs here will need careful consideration.

In this course we will not attempt to unify the irretrievably diverse. Thus we will have a small number of machine models and will design algorithms for our chosen problems for some or all of these. However, in doing so we still hope to emphasize common principles of design which transcend the differences in architecture. Equally, in some instances, we will exploit particular features of one model where that leads to a novel or particularly effective algorithm. Similarly, we will investigate notions of "better" as they have been traditionally defined in the context of each model. We will continue to employ the notation of asymptotic analysis, but note that we must be particularly wary of constant factors in the parallel case - a "constant factor" discrepancy of 32 in an asymptotically optimal algorithm on a 64 processor machine is a serious matter.

The PRAM Model

The Parallel RAM (PRAM) is a model (or rather a family of models) forming a natural generalisation of the RAM model beloved of the sequential algorithms community. Its main attraction is simplicity, allowing the algorithm designer to concentrate on the essence of a problem rather than architectural distractions. The price of simplicity is a questionable applicability to any realistic machines (in the sense that the cost of a PRAM algorithm when implemented on a more practical system may be rather different from that of its abstract cost, and worse, that the relative performance of two PRAM algorithms may even be reversed when implemented realistically).

The PRAM model allows some given number p of processors to access an m location shared memory (where for our purposes m will always be "large enough" and not of interest). Processors are synchronised for free whenever we like (to avoid worries about races) but can be executing different instructions during any one step (though typical PRAM algorithms tend not to exploit this significantly in practice). The EREW, CREW, ERCW and CRCW variants (with CRCW having its own sub-variants) determine the extent to which accesses to the shared memory can clash in any step, and the way in which clashes (if allowed) are resolved.

- In the "common" write variant, concurrent writes to the same variable are only allowed if all processors are trying to write the same value.
- In the "arbitrary" write variant, one of the written values is chosen randomly to be that which actually succeeds.
- In the "priority" write variant, the written value is the one from the processor with the highest priority (given some notion of priority, such as smallest or largest processor ID).
- In the "associative" write variant, all clashing write values are combined with some associative operator (such as "max", "min" or "+").

Each parallel step (one instruction in each processor) is charged one time unit (wherein lies the source of most arguments about realistic applicability). The run time of an algorithm is then modelled simply by the number

of such steps required and usually expressed as a function of problem size n and p (which may itself be expressed as a function of n).

For example, consider the problem of summing an array of n integers. With the CRCW-Associative PRAM we have a simple n processor single time step (or asymptotically $\Theta(1)$ time) algorithm - each processor writes a distinct array element to the "sum" location and the clash resolution mechanism (with + as the associative operator) does the rest. By contrast, in the EREW variant an obvious approach is to use $\frac{n}{2}$ processors to add distinct pairs in the first step, then $\frac{n}{4}$ of these processors to add distinct pairs of results in the second step, and so on. This process continues for $\Theta(\log n)$ steps until the final two sub-totals are summed into the intended sum location by a single processor.

As well as absolute speed, a significant focus interest concerns the design of "cost-efficient" or "cost-optimal" PRAM algorithms.

Definition 1 *The* **cost** *of a parallel algorithm is the product of its run time T_p and the number of processors used p. A parallel algorithm is* **cost optimal** *when its cost matches the run time of the best known sequential algorithm T_s for the same problem. The* **speed up** *S offered by a parallel algorithm is simply the ratio of the run time of the best known sequential algorithm to that of the parallel algorithm. Its* **efficiency** *E is the ratio of the speed up to the number of processors used (so a cost optimal parallel algorithm has speed up p and efficiency 1 (or $\Theta(1)$ asymptotically).*

For example, the sequential run time of (comparison based) sorting is known to be $\Theta(n \log n)$. A cost optimal parallel sorting algorithm might use $O(n)$ processors for $O(\log n)$ time, or $O\left(\frac{n}{logn}\right)$ processors for $O\left(\log^2 n\right)$ time. On the other hand, an $O\left(n^2\right)$ processor, constant time sorting algorithm would be faster than both of these (given enough processors) but not cost-optimal.

The significance of cost optimality is that it implies good scalability down to smaller sized machines. It is not difficult to see that a PRAM algorithm for say n^2 processors can be emulated on n processors with a corresponding slow-down of a factor of n (each abstract time step is emulated by n real time steps in which each processor plays the role of n imaginary processors). This

is called *round-robin* scheduling. Scaling down a cost optimal algorithm still produces a fast (if slower than the original) algorithm. Scaling down a non-optimal algorithm may soon produce a parallel algorithm which is slower than the best sequential run time (consider scaling down an n^3 processor constant time sorting algorithm to n processors).

The degree to which cost-optimality is missed impacts upon the range of problem and machine sizes over which an algorithm is useful (we will investigate this idea soon). Notice that the definition of speed-up makes it impossible to achieve greater than n fold speed-up with n processors (by a similar emulation). Systems which claim "super-linear" speed-up will be found on closer inspection to be either benefiting from different memory hierarchy performance in the two cases, or to be hitting data dependent instances of a problem in which a "better" sequential solution can indeed by formulated by emulation of a parallel one (such as in branch-and-bound search algorithms).

The notion of scaling PRAM algorithms down to smaller numbers of processors is captured precisely in a "Brent's Theorem".

A PRAM algorithm involving t time steps and performing a total of m operations, can be executed by p processors in no more than $t + \frac{(m-t)}{p}$ time steps.

Proof: Let s_i denote the number of computational operations performed by A at step i, where $1 \leq i \leq t$. By definition $\sum_{i=1}^{t} s_i = m$. Using p processors we can simulate step i in time $\lceil \frac{s_i}{p} \rceil$. The entire computation A can be performed with p processors in time

$$\begin{aligned} \sum_{i=1}^{t} \lceil \frac{s_i}{p} \rceil &\leq \sum_{i=1}^{t} \frac{s_i+p-1}{p} \\ &= \sum_{i=1}^{t} \frac{p}{p} + \sum_{i=1}^{t} \frac{s_i-1}{p} \\ &= t + \frac{m-t}{p} \end{aligned}$$

•

Notice that the theorem deals precisely with the number of operations rather than the cost. These will differ if some processors are idle during certain phases of an algorithm. Our simple summation algorithm is not cost optimal, but Brent's theorem tells us that a cost optimal execution exists

(though not necessarily how to express it as an algorithm). With a little more thought we can adapt our algorithm to produce an asymptotically optimal variant. The trick (which will be applicable in many situations), is to have a smaller number of processors each do some of the work sequentially and optimally, to improve the cost-efficiency to the extent that we can hide a less efficient second phase in the $O()$ notation. In this case, Brent tells us that we should work with $\Theta\left(\frac{n}{\log n}\right)$ processors. If each of these sums $\log n$ items sequentially (in $\Theta(\log n)$ time), and then co-operates in the original parallel summation approach (but now with fewer items and steps), then we still have a $\Theta(\log n)$ time algorithm, but one which is now cost optimal.

Strictly speaking, neither round-robin scheduling nor Brent's theorem apply to CRCW-associative PRAM algorithms, since breaking the work of what was a single steps across several steps can change the program's behaviour (for example, think about our single step summation algorithm). However, the techniques can be adapted to apply to even this most powerful model, with only a small constant-factor increase in time (and so no change asymptotically).

The choice of PRAM variant can have an impact on the run time which can be achieved for many problems. For example, the following CRCW-Associative (+) algorithm allows constant-time comparison based sorting of n items with n^2 processors. This is not possible in any non-concurrent-write variant (and could be argued to call into question the practicality of this model).

```
for i = 0 to n-1 do in parallel
  for j = 0 to n-1 do in parallel
    if (A[i]>A[j]) or (A[i]=A[j] and i>j) then
       wins[i] = 1;  /* exploiting concurrent writes */
    else
       wins[i] = 0;
for i = 0 to n-1 do in parallel
  A[wins[i]] = A[i]; /* writes to distinct locations */
```

Notice that the second clause in the conditional breaks ties between duplicated values, ensuring that each entry has a distinct number of wins.