Introduction

Parallel Computing

- Parallel Computing is an area of Computer Science included within the more general area of High Performance computing, which includes as other aspects that are not concerned with parallelism: cache techniques, data structure and algorithm specialization, I/O optimization, instruction reorganization, optimizing compilers.
- Parallel computing is a way as well to achieve high performance; always the goal is to achieve high performance. All aspects of high performance computing should be considered, not just parallelism.
- Parallel computing requires parallel architectures. There is no single architecture that suits *every* problem!
- Incorrect application of parallelism can hurt performance! Sometimes parallelism is not effective.
- As a result, there can exist a big gap between parallel computing theory and practice. In this subject the lectures will concentrate on theoretical aspects while the workshops will be very practical.

Parallel Algorithms

This subject is focused a lot on parallel algorithms.

- For every *sequential* algorithm that you have studied, you are now required to consider its parallel versions.
- We say *versions* because the exact parallel version of a given sequential algorithm that you use will depend on the architecture of the machine upon which it runs!
- Thus, the study of parallel algorithms cannot take place without specifying the parallel architecture, or *model*, that is being considered.
- As well, the best choice of parallel algorithm may depend on the number of instances of the problem that you wish to solve.

The above aspects make the study of parallel computing particularly challenging to students new to the area.

Applications

Many fields in science and engineering have computationally intensive problems that are intractable without the use of parallel computing.

- climate modeling (which consists of atmosphere model, ocean model, hurricane model, hydrological model and sea-ice model),
- plasma physics (to produce safe, clean and cost-effective energy from nuclear fusion),
- engineering design (of aircraft, ships, and vehicles),
- bio-informatics and computational biology,
- geophysical exploration and geoscience,
- astrophysics, material science and nanotechnology,
- defense (cracking cryptography code),
- computational fluid dynamics, computational physics, and
- big data processing.

Typically, without considering parallelism, programmers and software engineers are already faced with a large number of questions regarding the hardware, programming languages and system architectures that could or should be used to build a system.

Ideally, parallelism would be transparent to these considerations. However the use of parallelism is not transparent and does cause significant additional questions at all levels of a system.

One could consider sequential computing as a subset of parallel computing, i.e. the case when there is a single processor.

The breadth and depth of study is beyond a single subject, in many cases a researcher will opt to specialize in a particular problem domain for parallelization, e.g. genome wide studies.

Historical perspective

- By 1965 the IBM System/360 mainframe was well established as a large centralized computer system for many corporations. It could use up to one megabyte of 32-bit word memory and could store many programs in memory at the same time, with OS/360 options for time-sharing.
- In the later 1960s, semiconductor technology spawned the minicomputer era: small, fast and inexpensive machines, but still too difficult for end-users. Companies such as DEC, Prime and Data Central built minicomputers. Cray Research Corporation introduced the best cost/performance supercomputer, the Cray-1, in 1976.
- By the mid 1980s, personal computers (PCs) or desktops were common and local area networks (LANs) of powerful desktops and workstations began to replace mainframes and minis by 1990. The network of workstations was typically 10 times less cost, with comparable performance. In the 1990s, several supercomputers were built, using thousands of processors with a dedicated interconnection network.E.g., Sequent Symmetry, Intel iPSC, nCUBE, Intel Paragon, Thinking Machines (CM-2, CM-5), MasPar (MP) and Fujitsu (VPP500).

- In 2000 and beyond, expensive, specialized parallel machines are now largely replaced by clusters of workstations. With the Internet, high performance computing has moved from a processor-centric view to a network-centric view. This has lead to grid computing and very recently "Cloud Computing".

- 2010 to today, multi-core desktops, including GPUs (nVidia and ATI), many-core CPUs (Xeon Phi), promote a new era in parallel computing. Systems must take advantage of the available parallelism to be competitive.

Minicomputers are now largely called workstations or servers, as opposed to desktops. A cluster is a collection of stand-alone computers connected using some interconnection network, such as gigabyte Ethernet. Compute grids link together many HPC sites over the Internet, but would rather today be called a Cloud.

As of June 2008, the fastest supercomputer in a single installation is at the National Supercomputing Center in Wuxi, China, a Sunway TaihuLight – Sunway MPP, Sunway SW26010 260C 1.45GHz, 10,649,600 Cores, Rmax (TFlop/s) 93,014.6, Rpeak (TFlop/s) 125,435.9, Power (kW) 15,371.

These stats were taken from `www.top500.org`, a list of the "Top 500" supercomputers. Go to the site and see a wide variety of stats.

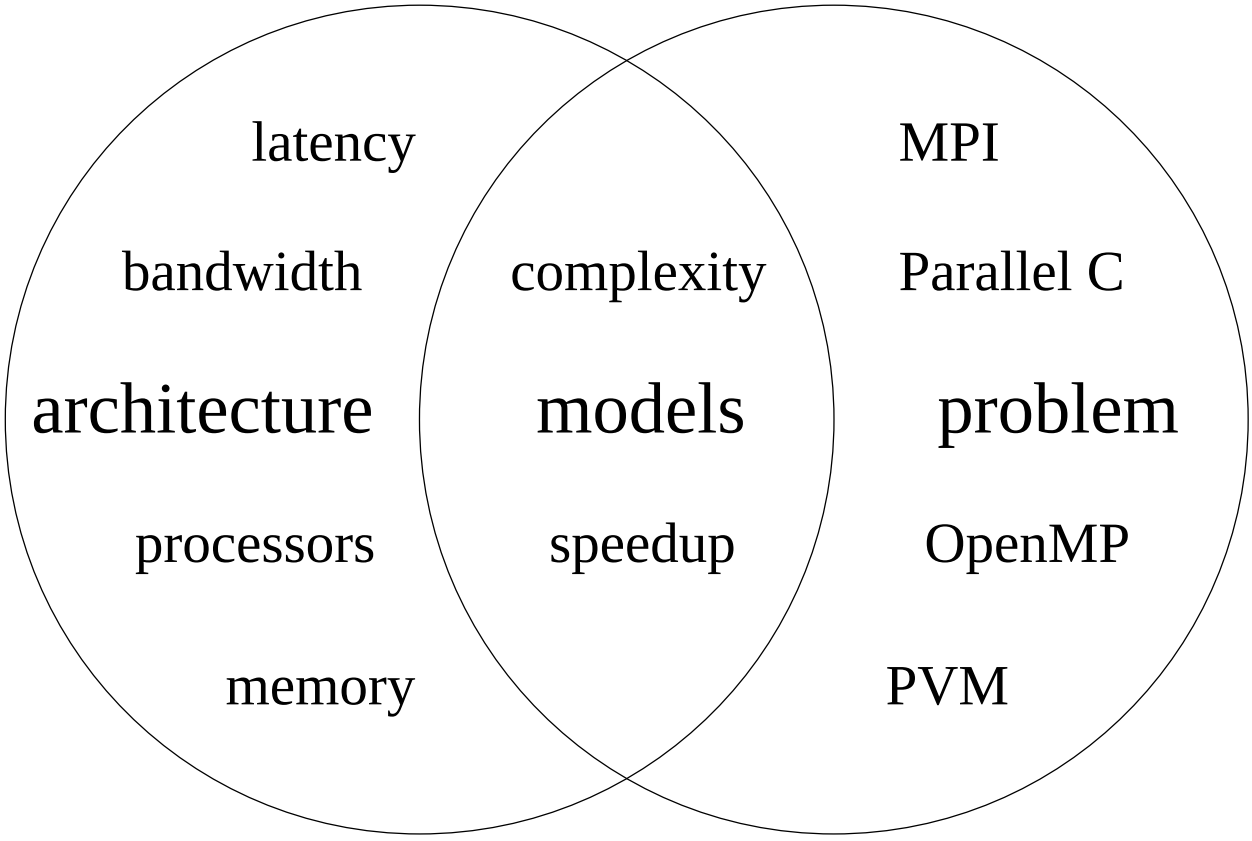

Figure 1: Overlapping aspects of parallel computing.

An architecture that is built without considering the problem to be solved is likely to exhibit less performance than it would otherwise.

A parallel program that does not consider the architecture that it is to execute on is likely to exhibit less performance than it would otherwise.

In between there are models that help us connect problems to be solved with architectures in order to achieve expected performance. Models allow programmers to predict what performance their program will have when it executes.

Some of the broader approaches to the study of parallel computing include:

- formalisms and mathematical logic;
- programming models, complexity and architectures;
- languages, compilers, explicit and implicit parallelism;
- environments, libraries and platforms; and
- problem domains and applications.

This subject will use these approaches, with more or less emphasis given to relevant and interesting aspects. Some aspects will span over a number of topics.

Parallel complexity analysis

The study of sequential algorithms goes hand-in-hand with complexity analysis as this becomes our basis for comparing one algorithm to another. It is similarly true for parallel algorithms and becomes more difficult as we additionally consider the *processor complexity* as well as computational steps and memory.

Following Krishnamurthy's guidelines, the three basic aims of complexity theory are to:

1. introduce a notation that specifies complexity;
2. choose a machine model that standardizes the measures; and
3. refine the measures for parallel computation.

We will start by seeing how parallelism is used to extend a sequential machine model and how this affects our complexity measures.

Common functions used in complexity analysis

Notation

For two functions $f(n)$ and $g(n)$, $f(n) = \mathcal{O}\big(g(n)\big)$ if there exists constants c and n_0 such that for all $n > n_0$, $|f(n)| \leq c|g(n)|$. For large n, the function f grows no faster than the function g.

E.g. is $n - \frac{\log n}{n} = \mathcal{O}(n)$? Yes, because for $c = 1$ and $n_0 = 1$ our definition is true: $n - \frac{\log n}{n} \leq n$ since $\frac{\log n}{n} > 0$ when $n > 1$.

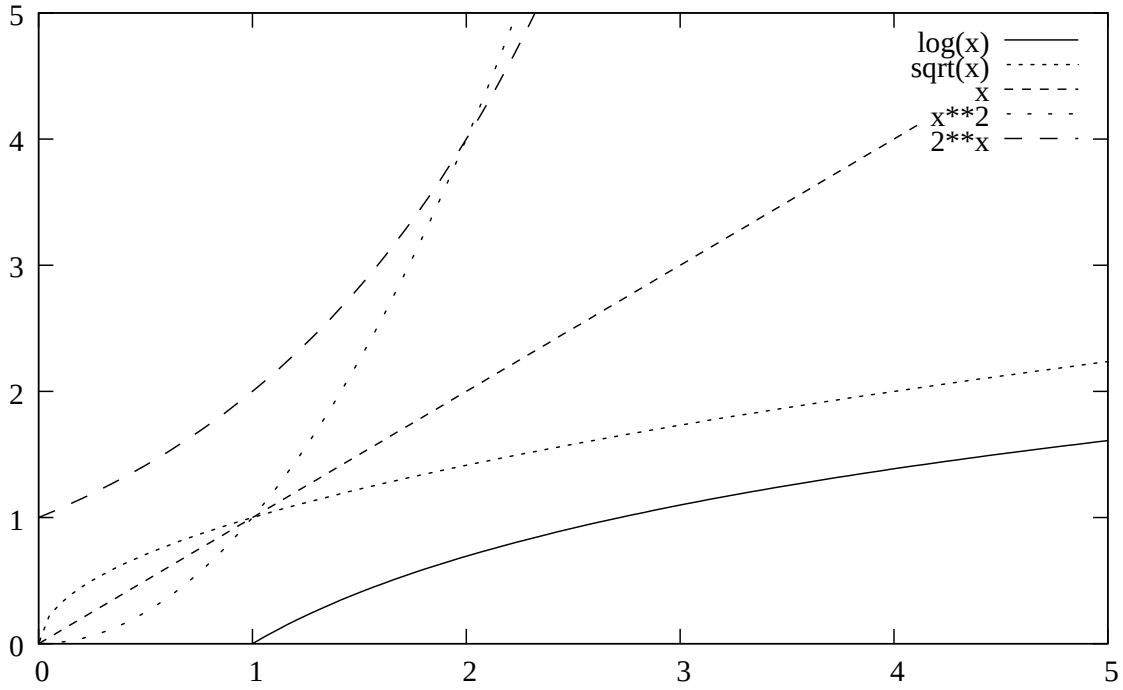

Figure 2: Some common functions of x at small values of x.

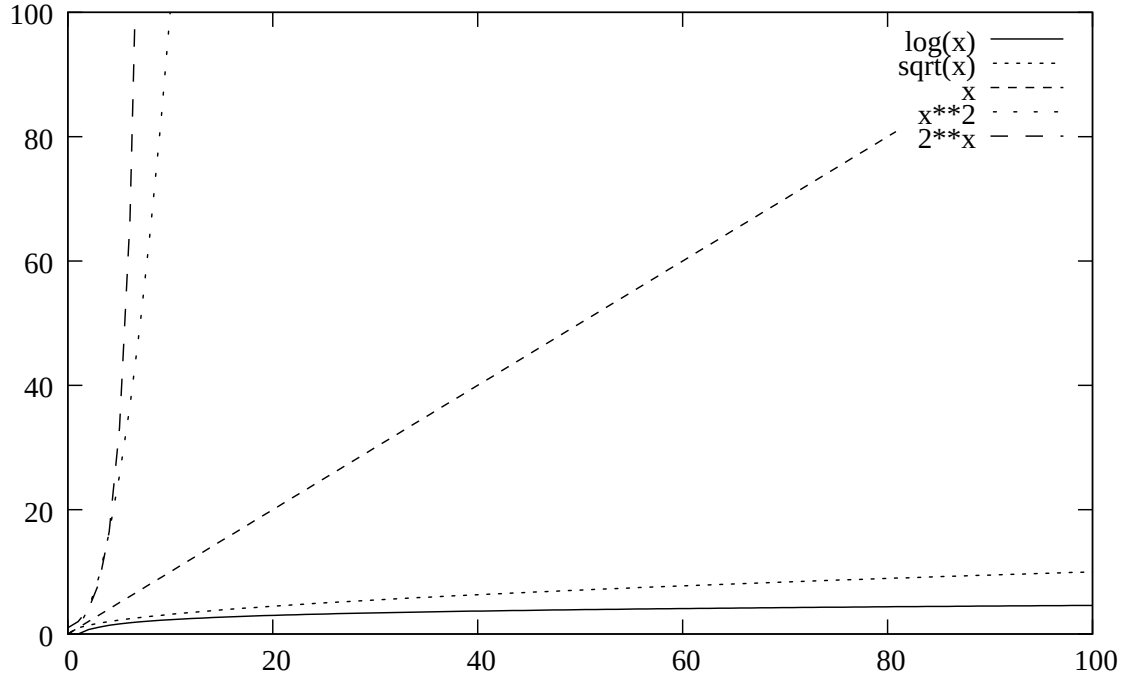

Figure 3: Some common functions of x at large values of x.

Exercise: show that $n^2 + n = \mathcal{O}(n^2)$.

Choose $c = 2$. Then:

$$\begin{aligned} n^2 + n &\leq 2\,n^2 \\ n &\leq n^2 \end{aligned}$$

which is true for all $n > n_0 = 1$.

Note that it is not true for $0 < n < 1$.

If $f(n) = \mathcal{O}\big(g(n)\big)$ then $g(n) = \Omega\big(f(n)\big)$ is similarly defined and means that for large n, the function g grows no slower than the function f.

E.g. is $n - \frac{\log n}{n} = \Omega(n)$? Yes, because for $c = 2$ and $n_0 = 0$ our definition is true:

$$\begin{aligned} n &\leq 2\left(n - \frac{\log n}{n}\right) \\ \frac{\log n}{n} &\leq n \\ \log n &\leq n^2 \end{aligned}$$

when $n > 0$.

If both $f(n) = \mathcal{O}\big(g(n)\big)$ and $f(n) = \Omega\big(g(n)\big)$ then $f(n) = \Theta\big(g(n)\big)$. For large n, the function f grows no slower and no faster than the function g.

E.g. is $n - \frac{\log n}{n} = \Theta(n)$? Yes, because it is both $\mathcal{O}(n)$ and $\Omega(n)$ from previous examples.

Exercises, test each of the following:

- $n \log n = \mathcal{O}(n)$?
- $n - \frac{1}{n} = \Omega(\sqrt{n})$?

For a constant c:

- $\mathcal{O}(1)$ is constant,
- $\mathcal{O}(\log n)$ is logarithmic,
- $\mathcal{O}((\log n)^c)$ is polylogarithmic,
- $\mathcal{O}(n)$ is linear,
- $\mathcal{O}(n \log n)$ is linearithmic,
- $\mathcal{O}(n^2)$ is quadratic,
- $\mathcal{O}(n^c)$ is polynomial or geometric,
- $\mathcal{O}(c^n)$ is exponential and
- $\mathcal{O}(n!)$ is factorial complexity.

For two functions $f(n)$ and $g(n)$, $f(n) = o\big(g(n)\big)$ if for any $\epsilon > 0$ there exists a constant n_0 such that $|f(n)| < \epsilon|g(n)|$. Thus $\lim_{n\to\infty} f(n)/g(n) = 0$. Similarly if $f(n) = o\big(g(n)\big)$ then $g(n) = \omega\big(f(n)\big)$ means that $\lim_{n\to\infty} g(n)/f(n) = \infty$.

RAM model

John von Neumann provided the *stored program concept*, referred to as the von Neumann Architecture or model of computation which leads us to the *random access machine* (RAM) model. There are three concepts:

- random access memory which stores information and is accessible independently of its content,
- a central processing unit that accesses the RAM using a Fetch-Decode-Execute-Writeback cycle and
- input/output devices.

The time taken to access the memory is constant over all addresses, each address stores the same amount of information and each decode and execute takes constant time.

Fig. 4 depicts the von Neumann Architecture. The acronym RAM is used for both random access memory and random access machine, the usage should be inferred from the context. RAM is sometimes called the primary storage. I/O is required for example to load a program into RAM from a secondary storage device or to store results. In most cases we do not depict the I/O because it is largely irrelevant to computation. Communication is depicted using hollow arrows; as we shall see in this subject, the communication mechanism is in general an interconnection network.

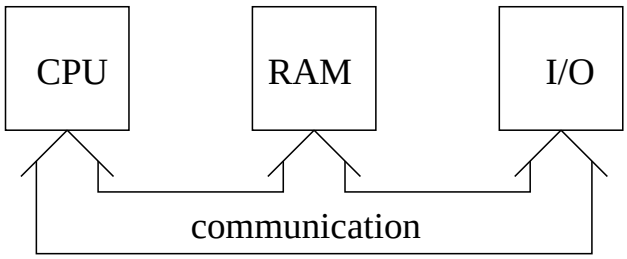

Figure 4: Concepts of the von Neumann Architecture

Computational models separate implementation from problem solving. Each operation in the example program translates to a constant number, i.e. independent of the problem size (n), of instructions.

Computational models allow programmers to develop expressive languages without having to consider implementation. Processor architects can continue to refine the implementation without needing to consider the programmer's choice of language. In both cases, it is understood how changes in the language or changes in the implementation will affect performance of the system.

Affects of changes become more difficult to understand as processor designs become more complicated. For example, caching can lead to significant improvements in performance and can be exploited by appropriate high level language constructs (though such architecture dependency can lead to further complications).

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    $0, -4(%ebp)
.L2:
        movl    -4(%ebp), %eax
        cmpl    8(%ebp), %eax
        jl      .L5
        jmp     .L1
.L5:
        movl    -4(%ebp), %eax
        leal    0(,%eax,4), %ecx
        movl    12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    %eax, (%edx,%ecx)
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     .L2
.L1:
        leave
        ret
```

```
void func(int n, int *a){
        int i;
        for(i=0;i<n;i++) a[i]=a[i]+1;
}
```

How to reduce complexity?

The RAM model can be used as a measure for the complexity of sequential algorithms. Optimal algorithms are known for many problems.

In general, architectural refinements (such as vector operations or multi-core) do not reduce the *complexity* of the algorithm. For a given computational model and algorithm with complexity of $\mathcal{O}(n)$, then architectural refinements *that are bounded in space* will not reduce the complexity to, e.g. $\mathcal{O}(\sqrt{n})$ or $\mathcal{O}(\log n)$.

To reduce complexity we need unbounded space, e.g. for a problem of size n we might ask for n processors; in general we may assume a function $p(n)$ processors. This begins the theory of parallel computing – such an unbounded machine is impossible to actually build.

There is an assumption that an unbounded amount of logic cannot reside in a bounded amount of space and nor can a bounded amount of logic execute an unbounded number of operations in a bounded time. These assumptions seem quite reasonable.

PRAM

The *parallel random access machine* or PRAM model of parallel computation is an idealization of a parallel architecture, proposed by Fortune and Wyllie in 1978. (See Karp and Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines", for a good overview of PRAM work.)

It can be described as consisting of p identical RAM processors, each with its own private memory, that share a single large memory.

The Fetch-Decode-Execute-Execute cycle is performed synchronously by the PRAM processors. In other words, in one unit of time, each processor can read one global or local memory location, execute a single RAM operation, and write into one global or local memory location.

The PRAM model is depicted in Fig. 5. I/O is omitted.

The PRAM model is very successful as a basis for parallel algorithm design.

Its power draws from the fact that the model ignores algorithmic complexity of machine connectivity and communication contention, data locality, synchronization, and reliability.

Implementations of a PRAM must address the above, additional aspects. In particular the PRAM model is generally classified into four sub-categories which relate to the use of shared memory:

EREW : exclusive read, exclusive write

CREW : concurrent read, exclusive write

CRCW : concurrent read, concurrent write

ERCW : exclusive read, concurrent write (shown only for completeness)

An EREW PRAM does not allow simultaneous access to a memory location for read or for write operations. A CREW PRAM allows simultaneous access for reading but not for writing. A CRCW PRAM allows simultaneous access for reading and for writing.

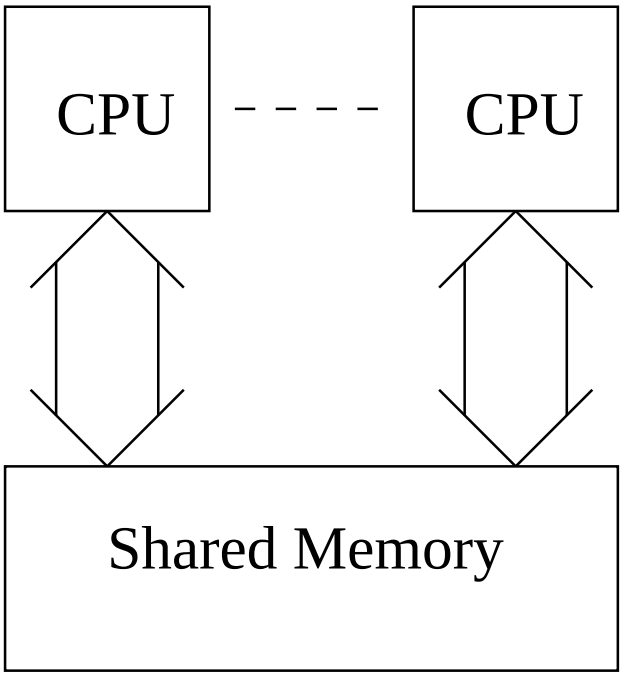

Figure 5: The PRAM model

Simultaneous writes require a further category that defines how write conflicts are resolved:

COMMON Value is written iff all processors write the same value (otherwise an error condition may be flagged).

ARBITRARY A processor is picked at random among the conflicting processors to write to the memory (randomness may or may not be usable); the algorithm should work regardless of which processor is picked.

PRIORITY The processor with the lowest identifier among the conflicting processors can write to the memory.

COMBINING A function of the conflicting values is written; this model requires defining the combining operation.

Power of PRAM model variations

The models are listed in increasing order of "power":

- any algorithm that runs on a EREW PRAM will run on a CREW PRAM,
- any algorithm that runs on a CREW PRAM will run on a COMMON CRCW PRAM, and so on.

However it is not true in the other direction. E.g. in general some problems can be solved on a CREW PRAM with a complexity that can not be obtained for the same problems on an EREW PRAM. This makes the CREW PRAM more powerful.

Having said this, the models do not differ much in their power, for example:

- Any algorithm for a CRCW PRAM in the PRIORITY model can be "simulated" by an EREW PRAM with the same number of processors and with the parallel time increased by only a factor of $\mathcal{O}(\log p)$.
- Any algorithm for a PRIORITY PRAM can be simulated by a COMMON PRAM with no loss in parallel time provided sufficiently many processors are available.

We will look at simulation later in this slide set. Basically when a PRAM simulates an algorithm from another PRAM, it substitutes some operations, that would not be permitted, with permitted operations that typically (necessarily) change the complexity of the algorithm.

Since the EREW PRAM has the weakest power, most researchers consider this when attempting to find optimal parallel algorithms: such algorithms will run without modification on other models.

Work and Size

Let S be a problem with input size n, best sequential running time $T(n)$, that can be solved on a PRAM by a parallel algorithm in $t(n)$ parallel steps, with $p(n)$ processors. The quantity

$$w(n) = t(n) \times p(n)$$

represents the *work* done by the parallel algorithm.

Any PRAM algorithm that performs work $w(n)$ can be converted into a sequential algorithm running in time $w(n)$ by having a single processor simulate each parallel step of the PRAM in $p(n)$ time units. Note that the definition of work assumes that all $p(n)$ processors are "busy" in every step, even though in many algorithms it may be that only a fraction of the $t(n)$ steps require $p(n)$ processors.

A related measure to consider is called *size*, which we will write as $size(n)$, that is the total number of operations that the parallel algorithm undertakes. Generally:

$$T(n) \leq size(n) \leq w(n).$$

Using fewer processors

Generally, if a parallel algorithm taking $t(n)$ time is known for p_1 processors then a parallel algorithm for $p_2 < p_1$ processors can be defined that increases $t(n)$ by a factor $\lceil \frac{p_1}{p_2} \rceil$. Each of the p_1 processors is assigned the work of up to $\lceil \frac{p_1}{p_2} \rceil$ processors.

The size of the new algorithm does not change. The work is now:

$$\left\lceil \frac{p_1}{p_2} \right\rceil \times t(n) \times p_2,$$

which, so long as p_2 divides p_1, is no worse.

Assuming that processor allocation is clear, it is relatively easy to convert an algorithm designed for a large network of processors into an equally efficient algorithm for a smaller network of processors. The reverse process is not so easy.

Hence, considering algorithms where the number of processors is equal to the size of the problem, e.g. that sort m numbers using $p = m$ processors is reasonable.

Brent's Principle

If an algorithm with size x operations can be performed in t time on a PRAM with sufficiently many processors, then it can be performed in time $t + \frac{(x-t)}{p}$ on a PRAM with p processors.

Intuitively, if the i-th step requires x_i operations then it can be simulated in time:

$$\left\lceil \frac{x_i}{p} \right\rceil \leq \frac{x_i}{p} + \frac{p-1}{p}.$$

The sum of x_i over t time steps is x and one arrives at Brent's result.

Brent's principle can be used to infer the best running time for a parallel algorithm that runs with a given size. But what do we mean by best running time? What can we hope to achieve with parallel computing?

Optimality

Let S be a problem of size n whose best sequential algorithm runs in time $T(n)$; in many cases $T(n) = \Omega(n)$ because the algorithm must look at all the input at least once.

A PRAM algorithm A for S, running in parallel time $t(n)$ with $p(n)$ processors is optimal if:

1. $t(n) = \mathcal{O}(\log^c n)$ for some constant $c > 0$; and
2. work $w(n) = p(n) \times t(n) = \mathcal{O}(T(n))$.

Note that:

- the parallel algorithm is doing the same work as the sequential algorithm, but achieves a high degree of parallelism,
- the definition suggests that a parallel algorithm is not optimal unless its runtime is polylogarithmic!

An *efficient* parallel algorithm for problem S is one for which the work $w(n) = T(n)\,\mathcal{O}(\log^c n)$ and $t(n) = \mathcal{O}(\log^c n)$.

Speedup

In not every case do we know the optimal parallel algorithm for a given problem, but our parallel algorithm is still efficient. If $T(n)$ is the best sequential complexity for a given problem and $t(n)$ is the parallel complexity of a given parallel algorithm then we have reduced the complexity of the problem by a factor of:

$$\frac{T(n)}{t(n)}.$$

This is called *speedup*. Usually speedup is considered in terms of the number of processors. E.g. let $t_p(n)$ be the parallel complexity using p processors then:

$$S(p) = \frac{T(n)}{t_p(n)}.$$

How does the speedup change with respect to the number of processors?

Efficiency

The *efficiency*, E, of a parallel algorithm is defined as

$$E = \frac{\text{sequential complexity}}{\text{parallel complexity} \times \text{processors}} = \frac{T(n)}{t_p(n) \times p},$$

or

$$\frac{S(p)}{p},$$

the speedup per processor. If $E = \mathcal{O}(1)$ then we say the parallel algorithm has optimal processor allocation (size equals work). But how large can p be while maintaining optimal processor allocation? If it can be

$$\Theta\left(\frac{T(n)}{\log^c n}\right)$$

then the algorithm is optimal by our definition of optimality.

Example of a naive parallel sort

Assume you are sorting n numbers with $p(n)$ processors. Let's say the numbers are distributed evenly over the $p(n)$ processors. Thus, each processor can sort (using a fast sequential algorithm) its $n/p(n)$ numbers in $\mathcal{O}((n/p(n)) \log(n/p(n)))$ time.

The sorted lists are then merged. In the first parallel step, $p(n)/2$ processors are used to merge lists of length $n/p(n)$. In the second step, $p(n)/4$ processors merge lists of length $2\,n/p(n)$, and so on. In the last step, 1 processor merges two lists of length $n/2$. The last step dominates and the merging takes $\mathcal{O}(n)$ time.

The total time is

$$\mathcal{O}(n) + \mathcal{O}\Big(\frac{n}{p(n)} \log \frac{n}{p(n)}\Big) = \mathcal{O}\Big(n + \frac{n}{p(n)} \log \frac{n}{p(n)}\Big)$$

and the speedup is at most $\log n$. The $\mathcal{O}(n)$ component cannot be avoided, no matter how many processors are used.

Try substituting $p(n) = \log n, p(n) = n, p(n) = n \log n, p(n) = n^2$ into the total time above.

Optimal sorting algorithms are known. We will discuss them later in the course.

Example

[From Leighton] Consider a problem of size M, and an M-processor algorithm that can solve the problem in M steps. Consider another algorithm that uses M^2 processors and that can solve the problem in $\sqrt{M}$ steps.

Now consider that we need to solve X instances of the problem and that we have a machine with M^2 processors. Should we use the first algorithm or the second algorithm?

The first algorithm requires $\lceil X/M \rceil M{=}\Theta(X + M)$ total steps.

The second algorithm requires $X\sqrt{M}$ total steps.

If $X < \sqrt{M}$ then we should use the second algorithm, otherwise we use the first algorithm!

Feasibility

But $T(n)$ can be a very high complexity function. So how big can we allow the number of processors to really be?

By feasible we tend to mean the ability to solve instances of the problem within our available resources. Generally, if the consumption of some resource grows exponentially to the problem size, then we can solve only small problem instances.

Thus, feasibility is provided when the growth rate for the resource is bounded by a polynomial of the problem size, i.e. in time $n^{\mathcal{O}(1)}$ or space $n^{\mathcal{O}(1)}$ for a problem of size n.

In parallel computations, a parallel algorithm is feasible if solutions to size n problems are found in $n^{\mathcal{O}(1)}$ time using $n^{\mathcal{O}(1)}$ processors.

In some cases it is cheaper to give more time for a problem to be solved than to invest in more processors to solve the problem in a shorter period of time. In some cases the cost of not having the solution in a given period of time is greater than the cost of additional processors.

Weather forecasting, market prediction and even drug design are examples of time limited problems. If the solutions take too long to compute, then there is little or no benefit.

Adding new processors or memory to a system is not as easy as adding time though. When a problem exhausts available memory, we are inclined to find solutions to the problem that use less memory (rather than just buying more memory, which may not be possible in any case). Similarly, considering processor requirements in terms of problem size allows us to examine the question of how big a problem can be solved in a given time with a given number of processors.

Informal definition of feasibility

We use the definition of Greenlaw and others:

- A problem is *feasible* if it can be solved by a parallel algorithm with worst case time and processor complexity $n^{\mathcal{O}(1)}$.
- A problem is *feasible highly parallel* if it can be solved by an algorithm with worst case time complexity $(\log n)^{\mathcal{O}(1)}$ and processor complexity $n^{\mathcal{O}(1)}$.
- A problem is *inherently sequential* if it is feasible but has no feasible highly parallel algorithm for its solution.

The feasible problem class is the same as the class P where the processor complexity is feasible. It is not known whether there are any inherently sequential problems.

Nick's Class

Nick's Class, NC, is the set of problems solvable in polylogarithmic time on a parallel computer with a polynomial number of processors. This class consists of the feasible highly parallel problems.

Similar to class P for sequential computers, NC is the class of tractable problems for parallel computers.

It is known that $NC \subset P$ (because a parallel computer can be simulated by a sequential computer).

It is not known whether $NC = P$; similar to the case for $P = NP$ it is suspected to be not true.

A basic P-complete problem is this: given a Turing machine, an input for that machine, and a number T (written in unary, e.g. 1 is 1, 2 is 11, 3 is 111 and so on), does that machine halt on that input within the first T steps? It is clear that this problem is P-complete. If we can parallelize a general simulation of a sequential computer, then we will be able to parallelize any program that runs on that computer. If this problem is in NC, then so is every other problem in P.

The Parallel Computation Thesis

Summing up issues of time and space is the *Parallel Computation Thesis*:

> "Whatever can be solved in polynomially bounded space in a TM using unlimited time can be solved in polynomially bounded time on a parallel machine using unlimited number of processors (or space) and conversely."

The thesis has not been proven (similarly to the Church-Turing thesis).

Parberry points out that we need to consider the notion of a "reasonable" parallel machine. Many would not call a parallel machine which runs for $t(n)$ steps and has $2^{2^{\mathcal{O}(t(n))}}$

processors "reasonable". A more acceptable number of processors would be $2^{\mathcal{O}(t(n))}$, which is achievable in a *lazy activation* model where initially only one processor is active, and in each time-step every active processor can activate an inactive one. At worst, the number of processors could be bounded by $2^{t(n)^{\mathcal{O}(1)}}$.

The reason for these considerations is that a (uniform) shared-memory machine with sufficiently many processors can compute any recursive function in constant time.

Speedup in wall-clock time

In practical settings we consider speedup in terms of wall-clock time:

$$S(p) = \frac{\text{exec. time using one processor}}{\text{exec. time using } p \text{ processors}} = \frac{t_s}{t_p}.$$

Here, p is the number of processors, t_s is the sequential execution time and t_p is the parallel execution time. In a practical case, time is in terms of "wall clock" time, but for PRAM algorithms we talk about complexity. They are two very different things.

Wall clock time includes all overheads; like network communication delays, cache contention, etc.

If $S(p) = \Theta(p)$ then we say that the parallel algorithm has achieved linear speedup. Speedup in terms of execution time can be superlinear if the parallel architecture benefits from additional memory or in cases where parallel searching can avoid otherwise lengthy sequential searching, due to pruning. From a computational complexity point of view, the speedup is rarely superlinear...

Maximum speedup

Amdahl's Law is one way of predicting the maximum achievable speedup for a given program. If t_s is the sequential running time and f is the fraction of this time that cannot be parallelized then the maximum speedup for p processors is:

$$\begin{aligned} S(p) = \frac{t_s}{f t_s + (1-f)t_s/p} &= \frac{p}{1 + (p-1)f} \\ &= \frac{1}{f} \text{ for large } p. \end{aligned}$$

Applied Amdahl's Law

Consider a case when the fraction of the problem that cannot be parallelized is a function of the problem size, $f(n)$. Then the speedup is $\frac{1}{f(n)}$ for large number of processors.

E.g., consider sorting where the elements to be sorted are read sequentially from disk. For a problem of size n the sequential run time is $\Theta(n \log n)$. If the sorting computations can be completely parallelized then the fraction that cannot be parallelized is:

$$f(n) = \Theta\left(\frac{n}{n \log n}\right) = \Theta\left(\frac{1}{\log n}\right).$$

For a large number of processors the speedup is limited to $\log n$.

Gustafson's Law

Gustafson's Law considers the total time of a parallel program using p processors as a serial part s and a parallel part r. Let $s + r = 1$ for algebraic convenience. The execution time for a single processor would be $s + p\,r$ so that the speedup factor becomes:

$$S(p) = \frac{s + p\,r}{s + r} = p + (1 - p)s.$$

Sub-optimal Parallel Add

Description: Returns the sum of n elements from array A.
Analysis: $\Theta(\log n)$
Processors: $\frac{n}{2}$

```
1: procedure SUMMATION^◇_EREW(A, n)
2:     j ← n/2
3:     while j ≥ 1 do
4:         for k ← 0 to j − 1 do in parallel
5:             processor k does
6:                 A[k] ← A[2 k] + A[2 k + 1]
7:         j ← j/2
8:     return A[0]
```

We have $t(n) = \Theta(\log n)$, $p(n) = \Theta(n)$, and so $w(n) = t(n)\,p(n) = \Theta(n \log n)$, which is sub-optimal work.

Can we do better?

The size of our sub-optimal parallel addition is n. The number of steps is $\log n$. Brent's Principle says there is a parallel algorithm with $p = \frac{n}{\log n}$ processors that can run in time:

$$t + \frac{(x - t)}{p} = \log n + \frac{n - \log n}{\frac{n}{\log n}} = \mathcal{O}(\log n),$$

which is no worse run time complexity than the sub-optimal algorithm, but fewer processors. However what is this algorithm?

Optimal Parallel Add

Description: Returns the sum of n elements from array A.
Analysis: $\Theta\big(\frac{n}{p} + \log p\big)$

1: **procedure** $\textsc{Summation}^{\bigstar}_{\text{EREW}}(A, n, p)$
2: **for** $i \leftarrow 0$ **to** $p - 1$ **do in parallel**
3: **processor** i **does**
4: $B[i] \leftarrow \sum_{k=0}^{\frac{n}{p}-1} A\left[i\frac{n}{p} + k\right]$ ▷ $\Theta\left(\frac{n}{p}\right)$ steps
5: **all processors do**
6: $\textsc{Summation}^{\diamondsuit}_{\text{EREW}}(B, p)$ ▷ $\Theta(\log p)$ steps
7: **return** $B[0]$

Let $p = \frac{n}{\log n}$. Then, the first part requires $\Theta(\log n)$ steps.
The remaining part requires steps:

$$\begin{aligned}\Theta\left(\log \frac{n}{\log n}\right) &= \Theta(\log n - \log\log n)\\ &= \Theta(\log n).\end{aligned}$$

Total run time is therefore $t(n) = \Theta(\log n)$, as before.
However $p(n) = \Theta\left(\frac{n}{\log n}\right)$ and therefore $w(n) = t(n)\,p(n) = \Theta(n)$.
This is optimal work.

COMMON CRCW PRAM

Consider the logical OR of n bits using n processors.

Description: Returns the logical OR of an n-bit binary array.
Analysis: $\mathcal{O}\left(\frac{n}{p}\right)$

1: **procedure** $\textsc{LogicalOR}^{\bigstar}_{\text{COMMON}}(A, n, p)$
2: $r \leftarrow 0$
3: **for** $i \leftarrow 0$ **to** $p - 1$ **do in parallel**
4: **processor** i **does**
5: **for** $j \leftarrow i\frac{n}{p}$ **to** $(i+1)\frac{n}{p} - 1$ **do**
6: **if** $A[j] = 1$ **then**
7: $r \leftarrow 1$ ▷ The only value written is 1
8: **return** r

This is done in $t(n) = \mathcal{O}(1)$ time using $p(n) = n$ processors. Work is $w(n) = \Theta(n)$ which is optimal since a sequential algorithm requires to check each bit.

What will be the time complexity on an EREW PRAM? Will the algorithm be optimal?

Description: Returns the maximum of n unique elements.
Analysis: $\mathcal{O}(1)$
Processors: n^2

1: **procedure** $\textsc{Maximum}_{\text{COMMON}}(A, n)$

```
2:    for k ← 0 to n − 1 do in parallel
3:        processor k does
4:            M[k] ← true
5:    for k ← 0 to n − 1 do in parallel
6:        for j ← 0 to n − 1 do in parallel
7:            processor (k, j) does
8:                if A[j] > A[k] then
9:                    M[k] ← false
10:   for k ← 0 to n − 1 do in parallel
11:       processor k does
12:           if M[k] = true then
13:               r ← A[k]
14:   return r
```

The first step is $\mathcal{O}(1)$ time, the second step is $\mathcal{O}(1)$ time and the third step is $\mathcal{O}(1)$ time; the total time is therefore $\mathcal{O}(1)$.

The number of processors is $p(n) = \Theta(n^2)$ and so the work $w(n) = \Theta(n^2)$ which is suboptimal since a sequential algorithm takes time $\Theta(n)$.

Consider finding the maximum of n^2 unique numbers using n^2 processors on a COMMON CRCW PRAM.

Description: Returns the maximum of n^2 unique elements.
Analysis: $\Theta(\log \log n)$
Processors: n^2

```
1: procedure FilterMaximum^◇_COMMON(A, n^2)
2:     if n = 2 then
3:         return max{A[0] ... A[3]}
4:     for k ← 0 to n − 1 do in parallel
5:         n processor array (k, 0) ... (k, n − 1) does
6:             M[k] ← FilterMaximum^◇_COMMON(A[k, :], n) ▷ A[k, :] = [A[k, 0] ... A[k, n − 1]]
7:     return Maximum_COMMON(M, n^2)
```

The last step uses max1 algorithm and takes $\mathcal{O}(1)$ time. The base case for the recursion takes $\mathcal{O}(1)$ time.

The first step is recursive. Each call of the recursion reduces the array size (and perfectly allocates processors over the reduced arrays).

After 1 step, each array has size $\sqrt{n} = n^{\frac{1}{2}}$.

After 2 steps, each array has size

$$\begin{aligned}\sqrt{\sqrt{n}} &= n^{\frac{1}{2^2}} \\ &= n^{\frac{1}{4}}.\end{aligned}$$

After j steps, each array has size $n^{\frac{1}{2^j}}$.

Let $n^{\frac{1}{2^j}} = 2$, then:

$$\begin{aligned}
\frac{1}{2^j} &= \log_n 2 \\
\Rightarrow 2^j &= \frac{\log n}{\log 2} \\
\Rightarrow j &= \log\left(\frac{\log n}{\log 2}\right) \\
&= \log\log(n) - \log\log(2) \\
&= \Theta(\log\log n)
\end{aligned}$$

This gives a total runtime $t(n) = \Theta(\log\log n)$ with $p(n) = n^2$ and work $w(n) = \Theta(n^2 \log\log n)$ which is efficient, since a sequential algorithm requires time $\Theta(n^2)$, and very close to optimal.

Again, what is the best EREW PRAM algorithm?

Element uniqueness: PRIORITY PRAM algorithm

Description: For an array A of n elements, drawn from the set $\{1 \dots n\}$, determine if all elements are distinctly different.

Analysis: $\mathcal{O}\left(\frac{n}{p}\right)$

```
1:  procedure ElementUniqueness*_PRIORITY(A, n, p)
2:      r ← true
3:      for i ← 0 to p − 1 do in parallel
4:          processor i does
5:              for j ← i n/p to (i + 1) n/p do
6:                  M[A[j]] ← i                    ▷ Only the lowest id will be written
7:      for i ← 0 to n − 1 do in parallel
8:          processor i does
9:              for j ← i n/p to (i + 1) n/p do
10:                 if M[A[j]] ≠ i then
11:                     r ← false                  ▷ The highest priority processor will write
12:     return r
```

This is a constant time algorithm, but requires a large array :-)

The best sequential algorithm is $\Theta(n \log n)$, that does not require any additional memory, or $\mathcal{O}(n)$ that requires infeasible memory. Therefore the PRIORITY PRAM algorithm is optimal.

Simulating concurrent reading and writing

In some cases, concurrent reading and/or concurrent writing is disallowed. In this case an EREW PRAM can simulate the missing concurrency with a penalty of $\mathcal{O}(\log n)$ time steps.

A simple technique to consider is called *broadcasting* (for concurrent read) or *reducing* (for concurrent write).

However, in general different subsets of processors in a PRAM will concurrently read-/write to different memory cells; so we need a more general approach.

Some additional memory is required to support the simulation.

REPLICATION$^{\diamondsuit}_{\text{EREW}}$ of x_0 to all $x_1 ... x_{n-1}$ in X of length n with n processors.

Description: If $X = [x_0 \dots x_{n-1}]$ then return $X = [x_0, x_1 \leftarrow x_0, \dots, x_{n-1} \leftarrow x_0]$.
Analysis: $\Theta(\log n)$
Processors: n

```
1: procedure REPLICATION◇EREW(X, n)
2:     for i ← 0 to log n − 1 do
3:         for j ← 2^i to 2^{i+1} − 1 do in parallel
4:             processor j does
5:                 X[j] ← X[j − 2^i]
6:     return X
```

REPLICATION$^{\bigstar}_{\text{EREW}}$ of x_0 to all $x_1 ... x_{n-1}$ in X of length n with p processors.

Description: If $X = [x_0 \dots x_{n-1}]$ then return $X = [x_0, x_1 \leftarrow x_0, \dots, x_{n-1} \leftarrow x_0]$.
Analysis: $\Theta\left(\frac{n}{p} + \log p\right)$

```
1: procedure REPLICATION★EREW(X, n, p)
2:     M[0] ← X[0]                                   ▷ Array M has size p
3:     all processors do
4:         REPLICATION◇EREW(M, p)                    ▷ Θ(log p) steps
5:     for i ← 0 to p − 1 do in parallel
6:         processor i does
7:             for j ← i n/p to (i + 1) n/p − 1 do   ▷ Θ(n/p) steps
8:                 X[j] ← M[i]
9:     return X
```

Simulating CR PRAMs on EREW PRAM

BROADCAST$^{\bigstar}_{\text{EREW}}$ x to p processors.

Description: The value x is intially in the local memory of the first processor in the array and the algorithm leads to all n processors having read x at least once.
Analysis: $\Theta(\log p)$

```
1: procedure BROADCAST★EREW(x, p)
2:     processor 0 does
3:         A[0] ← x                                  ▷ Temporary array of length p
4:     all processors do
```

5: REPLICATION$^{\diamondsuit}_{\text{EREW}}(A, p)$ ▷ No need for optimal replication

Simulating CW PRAMs on EREW PRAM

What about these?

- PRIORITY CRCW PRAM
- COMBINING CRCW PRAM

They are both more powerful than the EREW PRAM. The COMBINING CRCW PRAM is quite general, since the "combining" function can in theory be any function of the inputs, though usually it is considered a dyadic operator of some kind, e.g. addition.

Simulating PRIORITY PRAM on COMMON PRAM

Description: Given an n processor PRIORITY PRAM, simulate it on an n^2 processor COMMON PRAM.
Analysis: $\mathcal{O}(1)$
Processors: n^2

```
1: procedure SIMULATEPRIORITY_COMMON(W, n)
2:     for k ← 0 to n − 1 do in parallel
3:         processor k does
4:             M[k] ← true
5:     for k ← 0 to n − 1 do in parallel
6:         for j ← 0 to k − 1 do in parallel
7:             processor (k, j) does
8:                 if W[j] = W[k] then        ▷ W[i] is the address that PRIORITY processor i wants to write to
9:                     M[k] ← false
10:    for k ← 0 to n − 1 do in parallel
11:        processor k does
12:            if M[k] = true then
13:                PRIORITY processor k can write to W[k]
```

Simulating PRIORITY PRAM on EREW PRAM

Suppose we want to simulate a PRIORITY PRAM write with n processors, on an EREW PRAM with slowdown $\mathcal{O}(\log n)$. Assume we can sort n numbers on an EREW PRAM with n processors in $\mathcal{O}(\log n)$ time.

Description: Given an n processor PRIORITY PRAM, simulate it on an n processor EREW PRAM.

Analysis: $\Theta(\log n)$
Processors: n

```
1: procedure SimulatePriority_EREW(W, n)
2:     for k ← 0 to n − 1 do in parallel
3:         processor k does
4:             A[k] ← (k, W[k], false)          ▷ W[i] is the address that PRIORITY processor i wants to write to
5:     all processors do
6:         Cole'sMergeSort*_EREW(A, n)          ▷ Sort by address W[k] then id k, Θ(log n) steps
7:     for k ← 0 to n − 1 do in parallel
8:         processor 0 does
9:             (a, b, c) ← A[0]
10:            A[a] ← (a, b, true)              ▷ Lowest id for the smallest address always wins
11:        processor k > 0 does
12:            (a, b, c) ← A[k]
13:            (a′, b′, c′) ← A[k − 1]
14:            if b ≠ b′ then
15:                A[a] ← (a, b, true)
16:            else
17:                A[a] ← (a, b, false)
18:    for k ← 0 to n − 1 do in parallel
19:        processor k does
20:            if A[k] = (k, W[k], true) then
21:                PRIORITY processor k can write to W[k]
```