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.

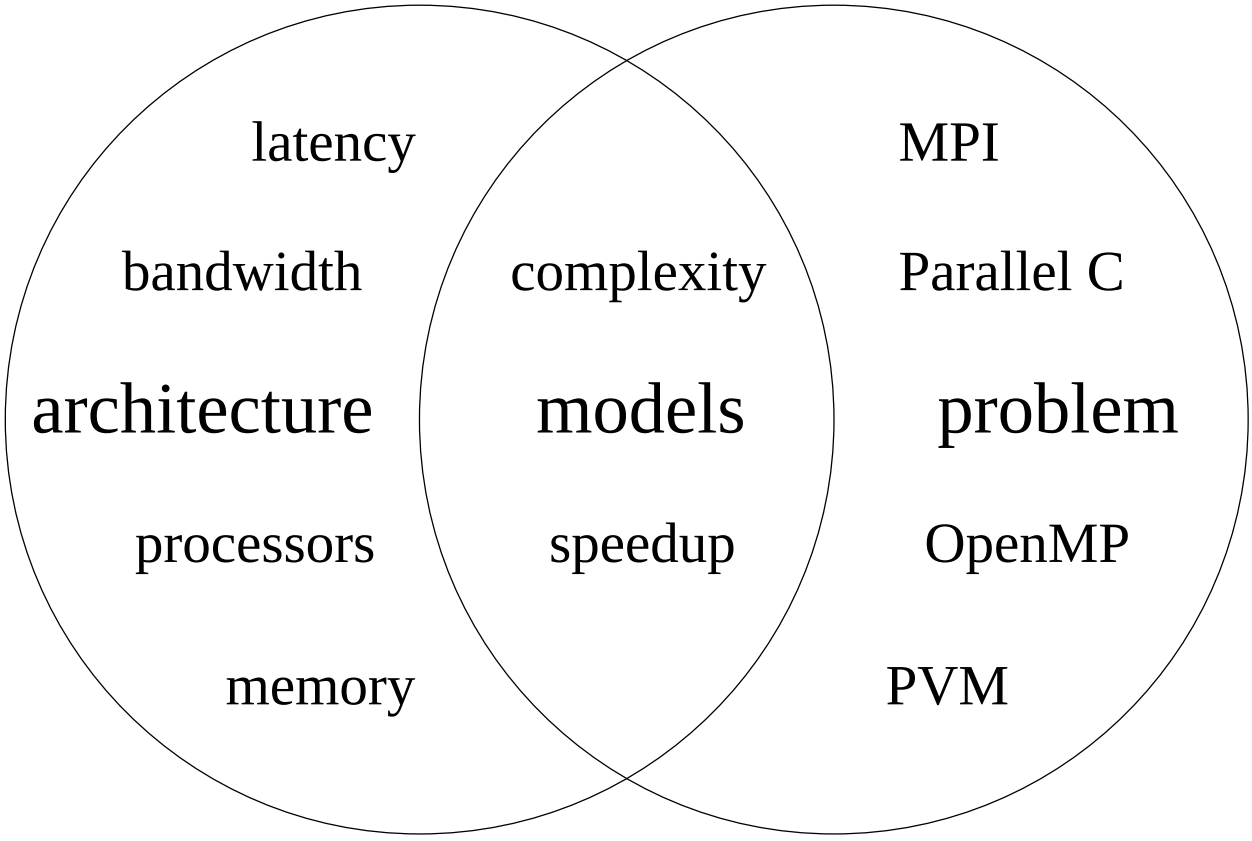


Figure 1: Overlapping aspects of parallel computing.

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 $n > 1$.

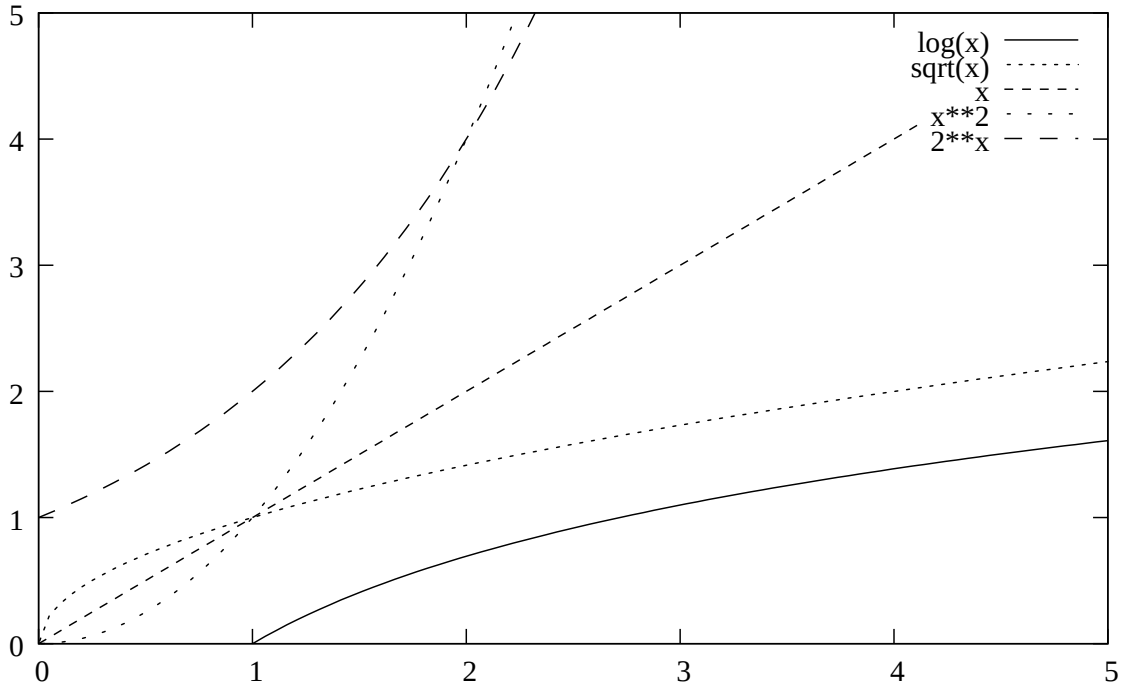


Figure 2: Some common functions of x at small values of x.

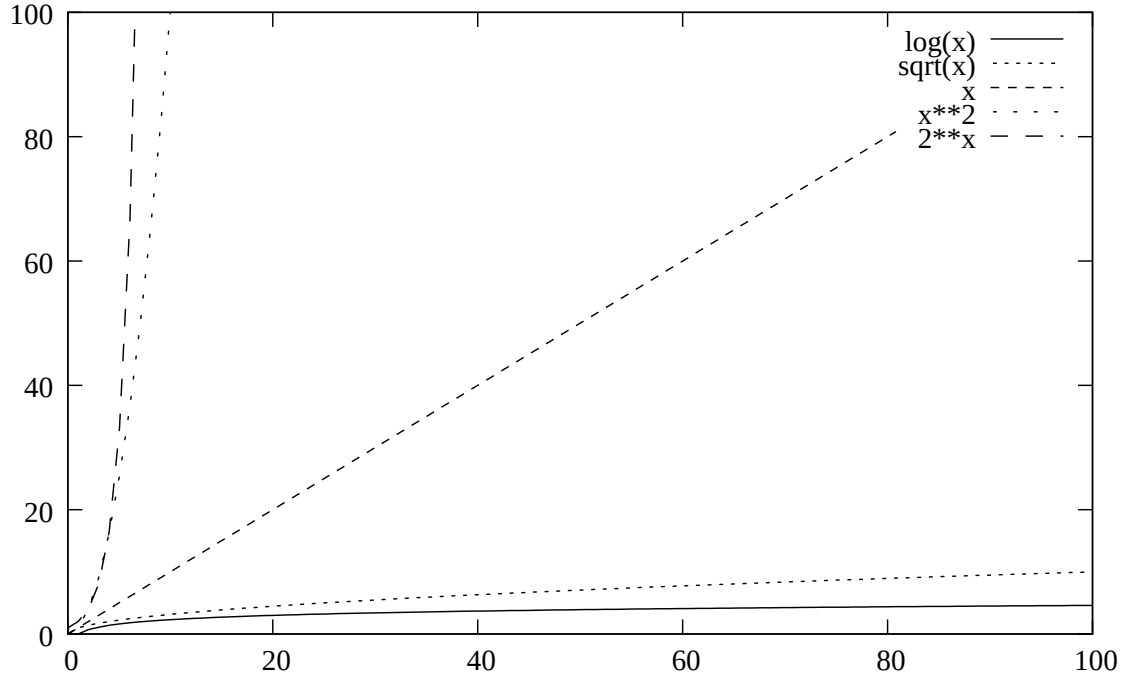


Figure 3: Some common functions of x at large values of x.

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.

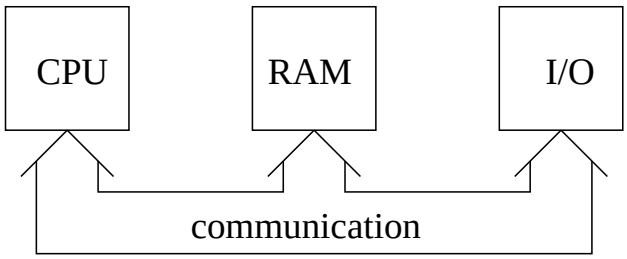


Figure 4: Concepts of the von Neumann Architecture

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.

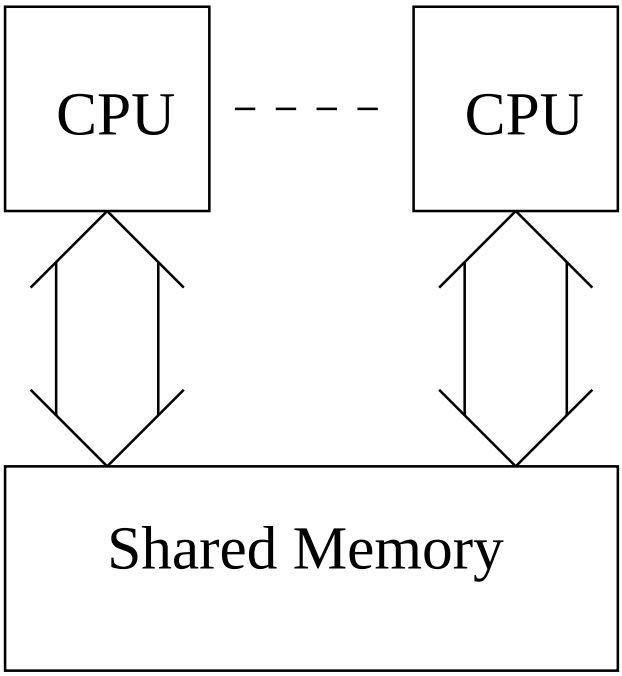


Figure 5: The PRAM model

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: $\Theta\left(\frac{n}{p} + \log p\right)$

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 $\{1 \ldots n\}$, determine if all elements are distinctly different.
Analysis: $\mathcal{O}(1)$
Processors: n

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

Parallel architectures and APIs

Parallel architectures

The PRAM model for parallel computation is algorithmically powerful and a direct implementation of a PRAM architecture is therefore desirable.

A criticism of the PRAM model is that in practice it is difficult to implement a machine with a potentially unbounded number of processors, that has constant access times to a shared memory across all processors.

In this section we consider possible architectural solutions for parallelism and the additional complexity that they exhibit. We can then revisit our models and provide refinements that include the architectural complexities.

PRAM considerations

The PRAM model considers N processors and M memory locations which are sometimes called *shared variables*.

While each processor may have its private memory, i.e. registers or cache, these are not shared with other processors.

A PRAM implementation must specify how the processors and memory locations are arranged so that the PRAM model is not violated, i.e. in a constant time step each of the processors can, in parallel, access a random memory location.

Flynn's classifications of parallel architectures

Flynn identifies four architectural classifications when considering the implementation of parallelism:

SISD : single instruction single data,

SIMD : single instruction multiple data,

MISD : multiple instruction single data,

MIMD : multiple instruction multiple data,

The definitions consider two fundamental streams, the *instruction* stream and the *data* stream.

All of the above parallelism can be simulated by a PRAM. The PRAM acts like a synchronous MIMD. But these architectures do not tell us how memory and processors are organized.

There are no machines widely accepted to be MISD.

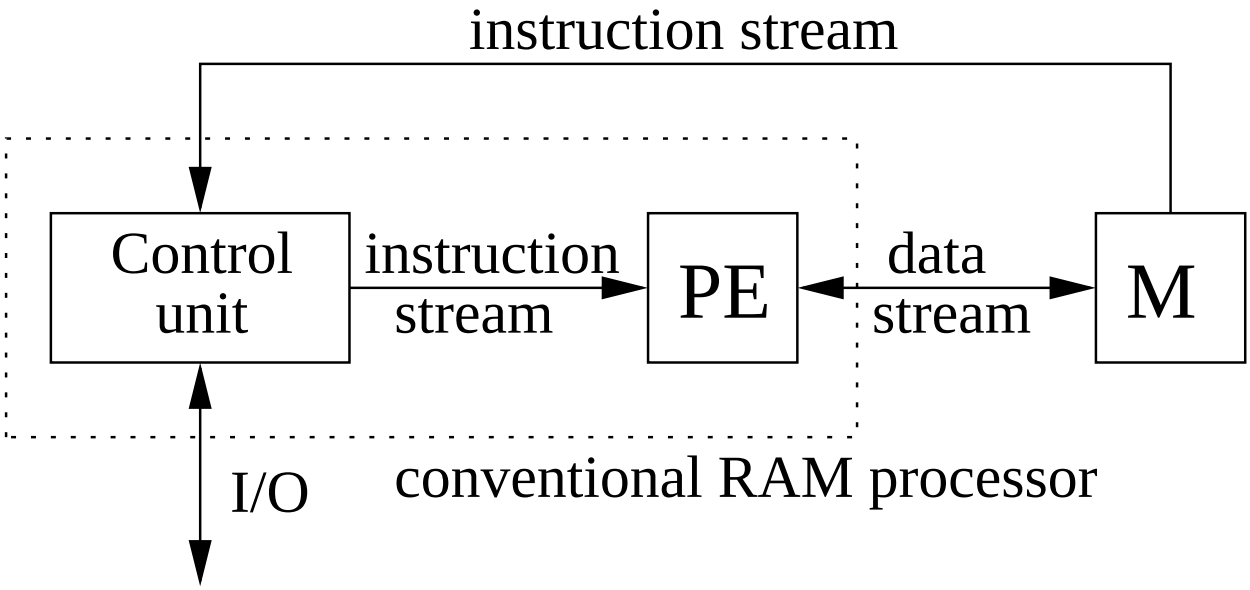


Figure 6: Abstract SISD

SIMD machine

For applications with lots of *data parallelism*, the most cost effective platforms are SIMD machines. In these machines, a single control unit broadcasts (micro-) instructions to many *processing elements* (each of which is a set of functional units with local storage) in parallel.

Until the rise of GPUs, the best known SIMD computer was the Connection Machine from Thinking Machines. The CM-2 model had 64k PEs, and even though each PE is only four bits wide, the machine could outperform many big Crays on some specially programmed problems.

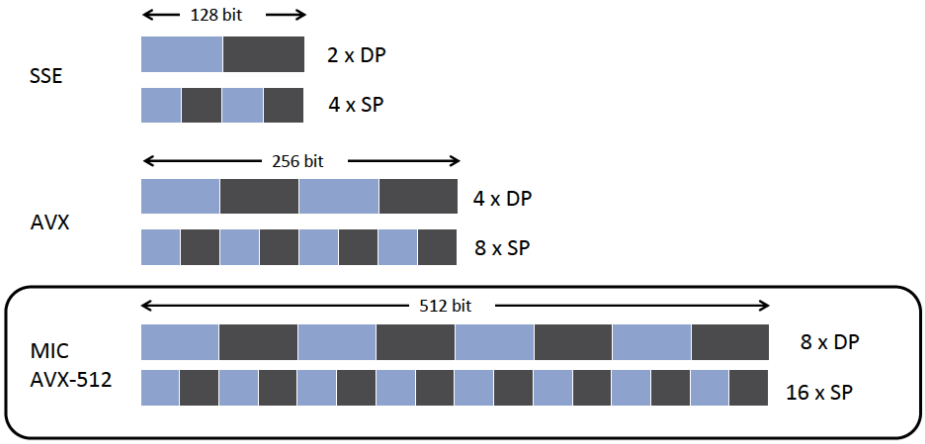


Figure 7: Abstract SIMD

MIMD machine

Most multiprocessors on the market today are shared memory MIMD machines. They are built out of standard processors and standard memory chips, interconnected by a fast bus (memory is interleaved).

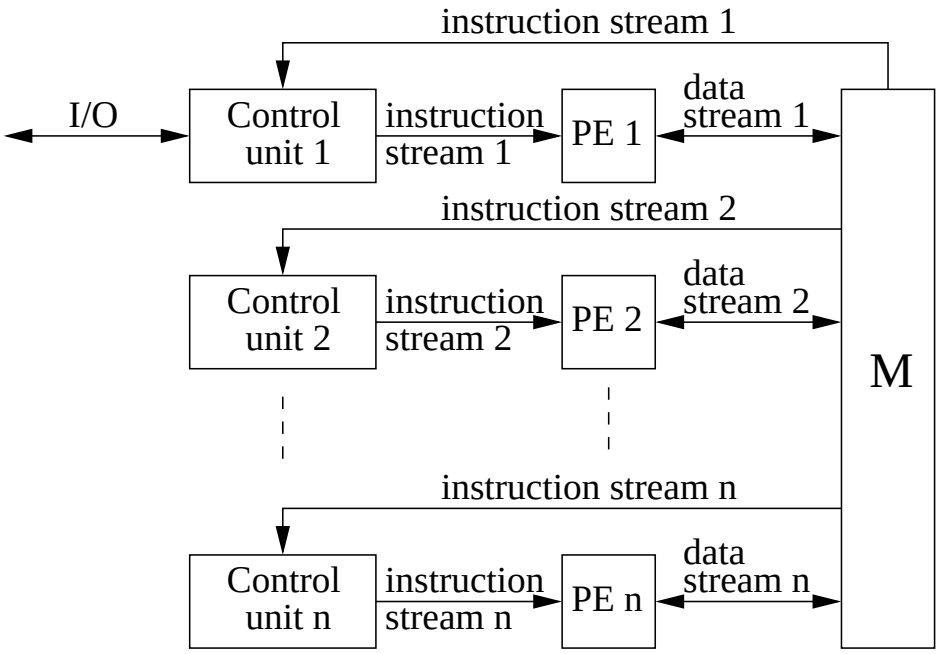


Figure 8: Abstract MIMD

The von Neumann architecture is classified as SISD. Providing more than one arithmetic logic unit (ALU) that can all operate in parallel on different inputs, providing the same operation, is an example of SIMD. This can be achieved by using multiple input busses in the CPU for each ALU that load data from multiple registers. The processor's control unit sends the same command to each of the ALUs to process the data and the results may be stored, again using multiple output busses.

Machines that provide vector operations are classified as SIMD. In this case a single instruction is simultaneously applied to a vector.

If the processor's control unit can send different instructions to each ALU in parallel then the architecture is MIMD. A superscalar architecture is also MIMD. In this case there are multiple execution units so that multiple instructions can be issued in parallel.

PowerPC processors that implement AltiVec execution units provide `vector` data types, e.g `vector float`, and C intrinsics to give programmers use of 128-bit vector operations. Using these vector operations can provide significant speed increases but can also lead to architecture dependent source code.

The MISD is rarely used and we wont make much mention of it.

For vector machines, the size of the vector is proportional to the parallelism. This is an example of *spatial parallelism*.

Pipelining exploits *temporal parallelism* within a single instruction stream. More pipeline stages generally leads to more parallelism, to a limit.

Schwartz's parallel machine classes

Schwartz defines two general approaches to arranging processors and memory.

Paracomputers separate the memory from the processors. The memory is shared and processors communicate via the shared memory. The PRAM module is closely approximated as a paracomputer in the sense that each memory location is equally accessible to each processor.

Ultracomputers distribute the memory over the processors (leading to modules). A processor can access the memory on its module in constant time but accessing memory on remote modules can take longer.

Alternate terminology is

- paracomputers = *shared memory multiprocessor*
- ultracomputer = *distributed memory multiprocessor*

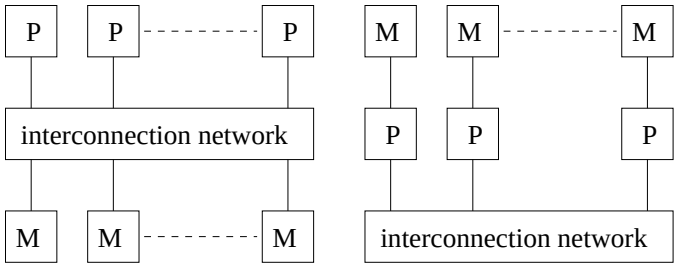


Figure 9: Shared (left) and distributed memory (right).

Shared versus distributed memory

In a shared memory system, processors communicate by reading and writing messages to a globally known memory address. The hardware ensures that all processors have the same access to the memory, i.e. using a *single address space*. For this reason it is often called a *symmetric multiprocessor* (SMP) machine.

In a distributed memory system, processors communicate by sending messages to each other. The hardware is only responsible for delivering the messages. There is no single address space. For this reason it is often called a *message passing* machine.

A *distributed shared memory* (DSM) system has distributed memory but a single address space.

Programming an SMP machine is easier than programming a message passing machine. However a message passing machine can scale up with less cost.

A SMP machine closely approximates the PRAM model. Programming a SMP machine is easier than a message passing machine because programmers can maintain the same data structure principles as would be used in sequential algorithms, e.g. a list that is processed by a single processor can be processed by two processors without significantly changing where or how the list is represented. On a message passing machine, programmers need to be concerned with *where* the data is and *when* to move data between processors.

SMPs can have trouble with cache consistency between different processors. The machine must usually ensure that all memory addresses are consistently seen by the processors, i.e. when a single address space is being used. Message passing machines do not have consistency problems because there is no shared memory that needs to be consistently represented, i.e. every processor has its own address space.

Message passing machines can be readily scaled up in size, in part because there is no need to maintain global consistency. The programmer explicitly maintains global consistency if required, by using various communication patterns, studied later in this chapter. Since the hardware is only required to communicate messages between the processors, it is less complex and cheaper to build. A view believed by many is that the Internet is a general purpose communication network that can be used as the basis for message passing machines.

A DSM architecture consists of a message passing machine with hardware and/or middleware that provides a consistent shared memory addressing scheme for programmers. This gives ease of programming plus scalability.

Popular SMP machines use Intel's Xeon processors, AMD's Athlon MP, Athlon 64 X2, and Opteron 200 series processors. These are basically entry level servers and workstations.

An example message passing machine includes the IBM Scalable Power Series (IBM POWERparallel 3 and SP 3).

Uniformity of shared memory access

Depending on the interconnection network, a shared memory machine can be either:

UMA : uniform memory access - all processors have equal access time to any given memory location.

NUMA : non-uniform memory access - typically exhibited by DSM architecture, memory locations incur different access delays depending on which processor accesses them.

COMA : the memory consists only of the collective cache contents of the processors and data migrates to the requesting processor.

UMA and cache issues

NUMA

Simultaneous memory access

Recall the abstract models of memory:

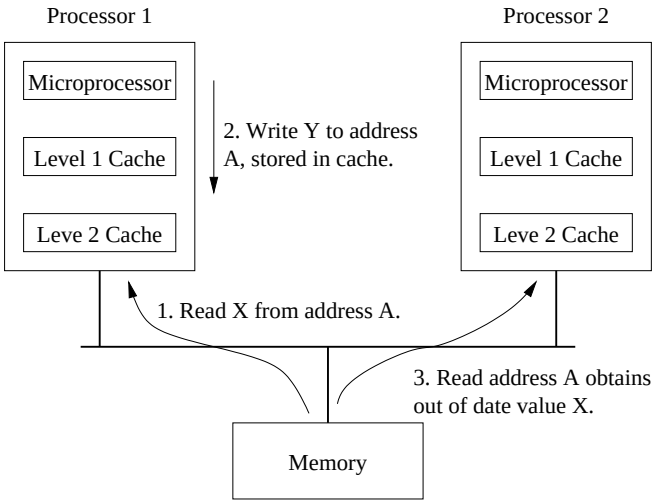


Figure 10: Caches improve performance but multi-processor machines need protocols to keep cache coherency.

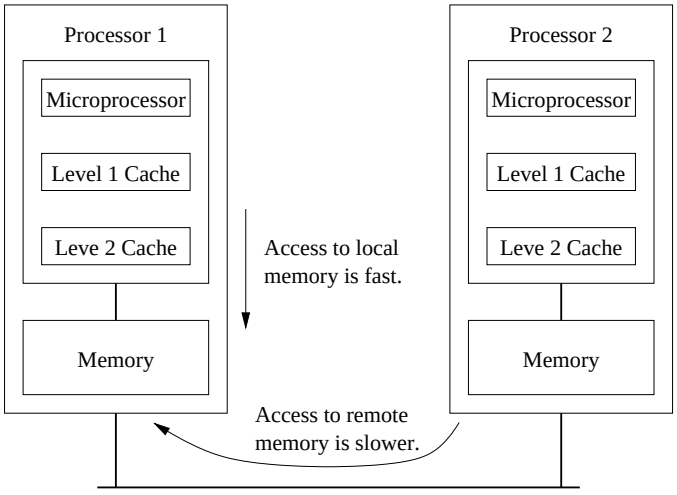


Figure 11: Distributed memory leads to non-uniform memory access. Cache problems exist for single address space systems.

- EREW (exclusive-read-exclusive-write
- CRCW (concurrent-read-concurrent-write) etc.

In these, access to one address is independent of access to any other.
Real memory is none of these.
Memory has pages (virtual memory) and cache (fast copies of recently used memory).
Memory also comes in *modules*.

- There is a long delay between requesting data and receiving it
- After a read, a module must "recharge" before it can be read again

Addressing memory banks

These limitations affect which addresses each memory bank should contain

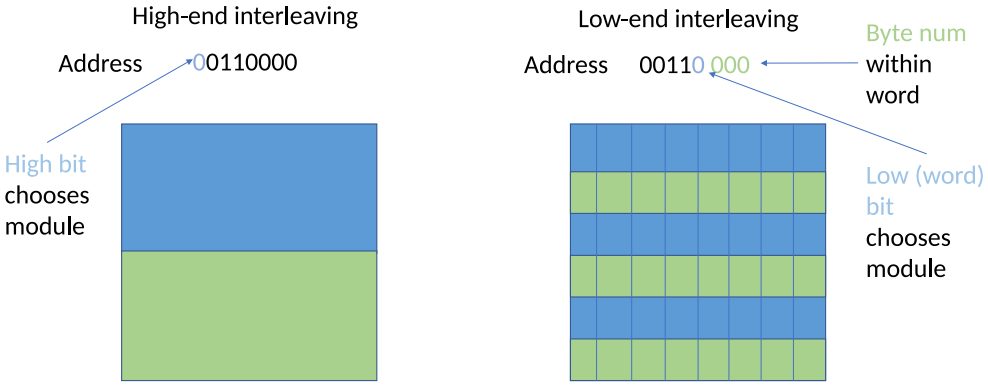


Figure 12: High-end (left) and low-end (right) interleaving

Coprocessors

Coprocessors are common these days, e.g. GPUs and Xeon Phi. They provide very low cost parallelism but architecturally they present more challenges.

Implicit versus Explicit

Models of parallel processing can be considered in a number of different ways (Skillicorn, *ACM Computing Survey*, Vol. 30, No. 2, 1998):

parallelism implicit/explicit: is a description of the parallel algorithm explicitly required?

decomposition implicit/explicit: are the pieces of the parallel program explicitly defined?

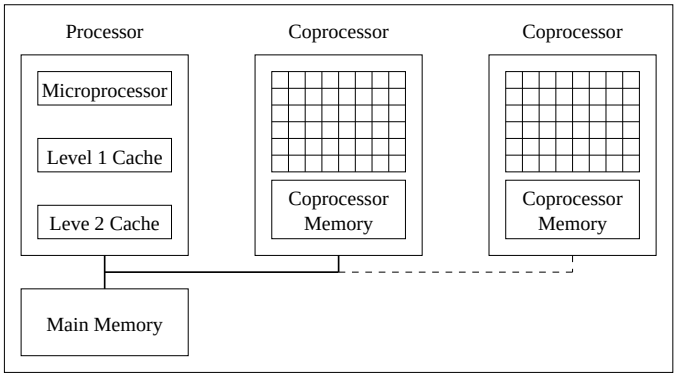


Figure 13: The coprocessor has its own memory and processing elements. Programs and data need to be transferred to and from the main memory and coprocessor memory.

mapping implicit/explicit: are the pieces of the parallel program required to be explicitly mapped to different processors?

communication implicit/explicit: is the communication between pieces of the parallel program required to be explicitly defined?

Threaded model for parallelism

The threaded model for parallelism is suitable when running on a single machine, i.e. single process with a single address space, or a distributed shared memory architecture that implicitly distributes threads (very rare). Note that coprocessors are usually treated as a separate machine, although this is increasingly being blurred with new developments (e.g., Java transparently running on GPUs).

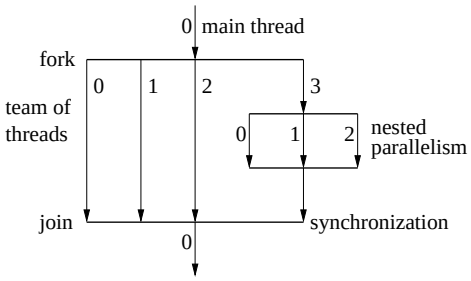


Figure 14: All processes start with a single, main thred, and create new threads as needed.

Process model for parallelism

The process model for parallelism is suitable when running on multiple machines, i.e. when there is no shared memory. Again, usually the process model is not transparent to coproces-

sors. The task model usually requires explicit decomposition and communication. Sometimes we use the word *task* to mean a process in a multi-process program.

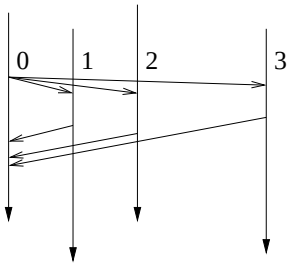


Figure 15: Processes run on individual machines and communicate by sending messages to each other. Some global knowledge, such as process number is usually available.

Hybrid model for parallelism

The hybrid model involves both multiple processes and multiple threads. Making use of a coprocessor can be considered a hybrid model in most cases.

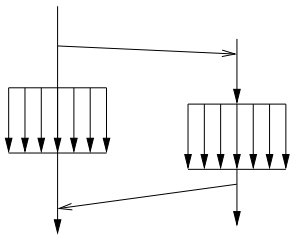


Figure 16: Two processes with each process making use of threads internally.

Since the thread model is more efficienct than the process model on a multicore machine, a distributed memory architecture where each machine has multiple cores would use the hybrid model, running a single process on each machine and having each process use multiple threads.

OpenMP

OpenMP, or Open Multi-Processing, is an API that supports shared memory multiprocessing programming in C, and other languages, across a wide range of platform. It is parallelism-explicit and mostly implicit in all other ways.

E.g. each iteration of this loop is assigned to a thread pool:

```
int main(int argc, char **argv)
{
    int a[100000];
```

```
    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

Java 1.8+ parallelism

While OpenMP is the mainstay in utilizing a multicore machine, Java 1.8 and above provides "parallelism" (its actually more akin to concurrency), in much the same way, via streams and lambda functions:

```
int[] a = new int[100000];

IntStream.range(0, 100000)
    .parallel()
    .forEach(i ->
        {
            a[i]=2*i;
        }
);
```

Similarly to OpenMP, this is parallelism-explicit and mostly implicit in all other ways.

omp4j

The *omp4j* project, `http://www.omp4j.org/`, provides OpenMP for Java, using an *omp4j* compiler rather than *javac*.

```
int[] a = new int[100000];

// omp parallel for
for (int i = 0; i < 100000; i++) {
    a[i]=2*i;
}
```

In this case the comment before the loop tells *omp4j* that the loop should be parallelized.

OpenMPI

OpenMPI, or Open Message Passing Interface, is for distributed memory machines where parallelism, decomposition and message passing are explicitly defined. Mapping can sometimes be defined but is usually implicit. E.g., usually the first process (master or process id 0) sends data to the other processes and receives processed data back:

```
if(my_id == 0) { // master
  for(an_id = 1; an_id < num_procs; an_id++) {
    start_row = an_id*num_rows_per_process;
    MPI_Send( &num_rows_to_send, 1, MPI_INT,
      an_id, send_data_tag, MPI_COMM_WORLD);
    MPI_Send( &array1[start_row], num_rows_per_process,
      MPI_FLOAT, an_id, send_data_tag, MPI_COMM_WORLD);
  for(an_id = 1; an_id < num_procs; an_id++) {
    MPI_Recv( &array2, num_rows_returned, MPI_FLOAT,
      MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
} else { // slave process

}
```

OpenCL

Open Computing Language is a framework for writing programs across heterogeneous platforms, e.g. consisting of multicore processors and coprocessors (GPUs, digital signal processors, etc).

```
// Multiplies A*x, leaving the result in y.
// A(i,j) is at A[i*ncols+j].
__kernel void matvec(__global const float *A,
                     __global const float *x,
                     uint ncols, __global float *y)
{
    size_t i = get_global_id(0);
    __global float const *a = &A[i*ncols];
    float sum = 0.f;
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```

SPMD and MPMD

The terms *Single Program Multiple Data* (SPMD) and *Multiple Program Multiple Data* (MPMD) in this subject will be used to categorize the way in which the parallel program is written, not the architecture, i.e. do not become confused with Flynn's Classifications.

For example, usually an OpenMP or OpenMPI program is written as a single program, and each instance (task or thread) of the program, operates on a different portion of the data, determined by its task identifier or rank. This is a SPMD approach.

Whereas in MPMD, different programs are written, e.g. perhaps in OpenMPI there is a separate program for the master as to the slave tasks. Still the same division of data is determined at the slaves by their task or rank identifier. Or indeed the MPMD approach may consist of many different programs, perhaps in a pipeline fashion, to achieve parallelism.

OpenMP

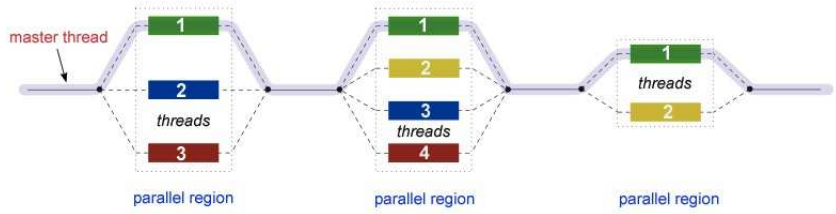
Overview

- What is OpenMP?
- Compiler Directives
- Run-time Library Routines
- Environment Variables
- C/C++ general code structure
- OpenMP Directives
  - SCHEDULE (Dynamic vs Static)
  - ORDERED clause
  - COLLAPSE clause
- Synchronization and sharing variables
- Memory Consistency and flush
- Data environment

What is OpenMP?

- The OpenMP is an Application Programming Interface (API) for parallel computing on multiprocessor *shared memory* machines.
- It *abstracts* the different thread implementations used in differing systems and *optimizes* thread usage according to the kinds of parallel operations that are required.
- The OpenMP API uses the *fork-join model* of parallel execution and is best suited to *large array-based applications*, although it can also be used for solving general parallel programs.

- One of three programming frameworks we will consider, in addition to the Message Passing Interface (MPI) and OpenCL for GPUs



OpenMP Programming Model

- Thread Based Parallelism
- Explicit Parallelism
- Fork-Join Model: All OpenMP programs begin as a single process: the master thread. The master thread then creates a team of parallel threads at parallel region. Then the team threads synchronize and terminate.

Isn't this model restrictive?

- POSIX threads (`pthreads`) are much more flexible
  - No need to keep joining
  - Ideal when each thread has a well-defined, different task
  - e.g., one running the GUI, one performing calculation, one loading / saving
  - (Ever noticed that you can't scroll with a document when you are choosing a filename to save as?)
- Flexibility requires the programmer to manage more details
  - Locks for accessing common data
  - Synchronization between cooperating threads
- Like the "rails" in Ruby on Rails
- If your program is suited to this model, the model helps.

OpenMP API Overview

- The OpenMP API is comprised of three distinct components. As of version 4.0:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables

- The application developer decides how to employ these components. In the simplest case, only a few of them are needed.

[fragile]

Compiler Directives

- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads
- Compiler directives have syntax like the following for C/C++:

```
#pragma omp parallel default(shared) private(beta)
```

[fragile]

Run-time Library Routines

- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Setting and querying nested parallelism
  - Querying if in a parallel region, and at what nesting level
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution
- Note that for C/C++, you usually need to include the `<omp.h>` header file.

```
int omp_get_num_threads(void);
```

OpenMP Run-time Library Routines

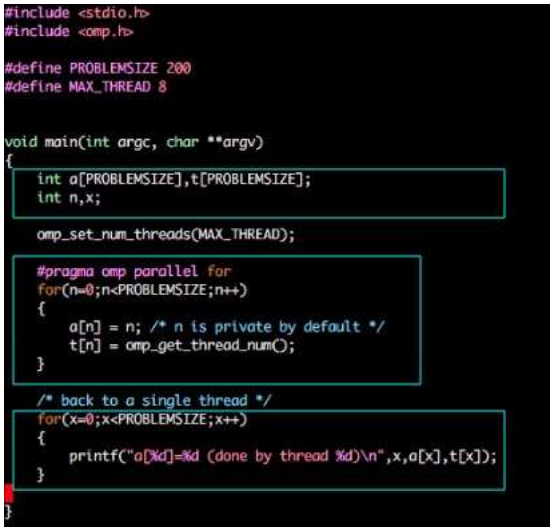
Routine	Purpose
OMP_SET_NUM_THREADS	Sets the number of threads that will be used in the next parallel region
OMP_GET_NUM_THREADS	Returns the number of threads that are currently in the team executing the parallel region from which it is called
OMP_GET_THREAD_LIMIT	Returns the maximum number of OpenMP threads available to a program
OMP_GET_THREAD_NUM	Returns the thread number of the thread, within the team, making this call.
OMP_GET_NUM_PROCS	Returns the number of processors that are available to the program
OMP_GET_WTIME	Provides a portable wall clock timing routine
OMP_GET_SCHEDULE	Returns the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_SET_SCHEDULE	Sets the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive

[fragile]

C/C++ general code structure

```
#include <omp.h>
main ()
{
int var1, var2, var3;
/* Serial code . . . */
/* Beginning of parallel section. Fork a team of threads. */
/* Specify variable scoping */
#pragma omp parallel private(var1, var2) shared(var3)
{
        /* Parallel section executed by all threads . */
        /* Other OpenMP directives . */
        /* Run-time Library calls . */
        /* All threads join master thread and disband */
}
/* Resume serial code . . . */
}
```

C/C++ general code structure



[fragile]

A simple OpemMP program – example1.c

```
#include <stdio.h>
#define SIZE 10

int main ()
{
    double value[SIZE];
    int i;

    #pragma omp parallel for num_threads (10)
    for (i = 0; i < SIZE; i++) {
        value[i] = i;
        printf ("%g\n", value[i]);
    }
}
```

[fragile]

Compiling and running the program

- `$ ssh` *your_login_id*`@spartan.hpc.unimelb.edu.au`
- `$ gcc -fopenmp example1.c -o example1`
- `$ sinteractive` *# brief shell on compute node for testing*
- `$ ./example1`
- Output: 1 0 7 6 8 5 9 4 3 2

[fragile]

Running in parallel on Spartan

- Create a SLURM file telling the system how to run your job, such as

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --time=0-12:00:00

# Load required modules
module load Python/3.5.2-intel-2016.u3

# Launch multiple process python code
echo "Searching␣for␣mentions"
time mpiexec -n 8 python3 twitter_search_541635.py -i /data/projects/COM
echo "Searching␣for␣topics"
time mpiexec -n 8 python3 twitter_search_541635.py -i /data/projects/COM
```

- `$ slurm` ⟨*your_slurm_file.slurm*⟩
- This schedules your job to execute, rather than running it now
- The wait can be frustratingly long (days if the system is busy)

(`https://dashboard.hpc.unimelb.edu.au/getting_started/`) [fragile]

Environment variables

- Did you notice that example.c specified the number of threads?
- You may want to debug your code on your own computer, and run it on Spartan
- You can specify the number of threads on each platform without changing the code
- Remove `numthreads (10)` from the code
- Set the environment variable `export OMP_NUM_THREADS=10`
- = number of threads created when the master thread hits this spot
- Other environment variables:

OMP_DYNAMIC	Adjust number of thread dynamically
OMP_NESTED	Allow non-master threads to spawn their own
OMP_THREAD_LIMIT	Max number of simultaneous threads for the v
OMP_STACKSIZE	Memory of each thread for local variables

[fragile]

Environment Variables

- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop interations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy
- Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which shell you use. E.g. for sh/bash:

```
export OMP_NUM_THREADS=8
```

[fragile]

OpenMP Directives

PARALLEL Region Construct

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

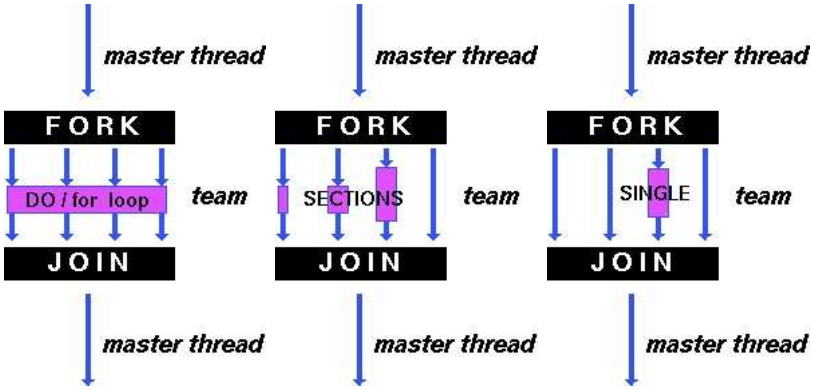
```
#pragma omp parallel [clause [clause ...]]
structured_block
```

where `clause` is one of

```
if (scalar_expression)
private (list) shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
num_threads (integer-expression)
```

OpenMP Directives

- Work-Sharing Constructs
  - A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
  - Work-sharing constructs do not launch new threads



  - There is an *implied barrier* at the end of a work sharing construct, but no implied barrier upon entry to a work-sharing construct.
    * No thread may process the instruction after the construct until all have finished the construct.
    * Common synchronisation construct. (See also Rendezvous in Plan 9, mutual exclusion etc.)

- Do / for
  SECTIONS
  SINGLE

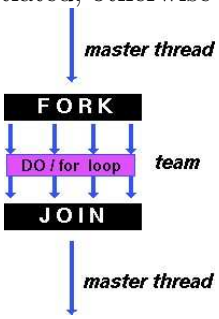
OpenMP Directives

Work-Sharing Constructs [fragile]

OpenMP Directives

Work-Sharing Constructs - DO / for Directive

specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes serially on a



single processor.

```
#pragma omp for [clause ...]
```

OpenMP Directives

Work-Sharing Constructs - DO / for Clauses

- SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. Types are STATIC, DYNAMIC, GUIDED, RUNTIME or AUTO
- nowait: If specified, then threads do not synchronize at the end of the parallel loop.
- ORDERED: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- COLLAPSE: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause.

SCHEDULE

- STATIC: Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
- DYNAMIC: Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
- GUIDED: Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.
- RUNTIME: The scheduling decision is deferred until runtime by the environment variable `OMP_SCHEDULE`.
- AUTO: The scheduling decision is delegated to the compiler and/or runtime system.

Code Example and Exercise

Go to directory `.../openmp/`
check out examples `helloworld.c` `multiply.c` `s_example.c`

1. Use run-time library routine to print out runtime of the program. Using `omp_get_wtime()`
2. Change schedule construct on `s_example.c`. E.g. static, dynamic. Change chunk size
3. Compare the runtime of static, dynamic schedule.
4. Write your own parallel version of a serial program `pi_serial.c` (under the same directory)

[fragile]

Dynamic vs Static Example - Static

```
#define THREADS 4
#define N 16

int main ( ) {
  int i;
  #pragma omp parallel for schedule(static)
        num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* wait for i seconds */
    sleep(i);
    printf("Thread␣%d␣has␣completed␣iteration␣%d.\n",
        omp_get_thread_num( ), i);
  }
  printf("All␣done!\n");
  return 0;
}
```

[fragile]

Dynamic vs Static Example - Dynamic

```
#define THREADS 4
#define N 16

int main ( ) {
  int i;
  #pragma omp parallel for schedule(dynamic)
        num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* wait for i seconds */
    sleep(i);
    printf("Thread␣%d␣has␣completed␣iteration␣%d.\n",
        omp_get_thread_num( ), i);
  }
  printf("All␣done!\n");
  return 0;
}
```

Dynamic vs Static

- Dynamic scheduling is better when the iterations may take very different amounts of time.
- However, there is some overhead to dynamic scheduling.
- After each iteration, the threads must stop and receive a new value of the loop variable to use for its next iteration.

OpenMP Directives

Work-Sharing Constructs - DO / for Clauses

- SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. Types are STATIC, DYNAMIC, GUIDED, RUNTIME or AUTO
- nowait: If specified, then threads do not synchronize at the end of the parallel loop.
- ORDERED: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- COLLAPSE: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause.

[fragile]

ORDERED clause

The ordered construct is used with loops and is useful for printing results in order.

```
#pragma omp parallel for ordered
for (i=0;i<10;i++)
{ a[i]=func(i);
    #pragma omp ordered
        print("a[%d] is %d",i,a[i]);
}
```

In the example, the array elements will be printed in order, rather than arbitrarily according to which threads get to the print statement first. [fragile]

COLLAPSE clause

Use the OpenMP collapse-clause to increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread.

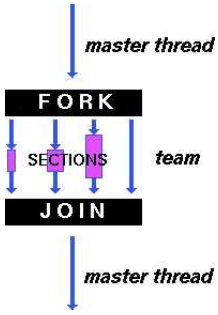
```
#pragma omp parallel for collapse(2)
    for (i = 0; i < imax; i++) {
        for (j = 0; j < jmax; j++) {
            a[ j + jmax*i] = 1.;
        }
  }
```

Especially useful when imax is smaller than number of threads. [fragile]

OpenMP Directives

Work-Sharing Constructs - SECTIONS Directive

specifies that the enclosed section(s) of code are to be divided among the threads in the team.




```
#pragma omp sections [nowait]
    #pragma omp section
      structured_block
```

[fragile]

OpenMP Directives

Work-Sharing Constructs - SECTIONS Directive

Parallel sections can be used to specify arbitrary work in different threads.

```
#pragma omp parallel sections
{
        #pragma omp section
        a[5]=a[5]*2;
        #pragma omp section
        a[1]=a[1]*2;
        #pragma omp section
        a[8]=a[8]*2;
}
```

[fragile]

OpenMP Directives

Work-Sharing Constructs - SINGLE Directive

specifies that the enclosed code is to be executed by only one thread in the team.

```
#pragma omp single [nowait]
      structured_block
```

[fragile]

OpenMP Directives

Work-Sharing Constructs - SINGLE Directive

```
#include <stdio.h>
#include <omp.h>

int main() {
   #pragma omp parallel num_threads(2)
   {
      char ch;
      #pragma omp single
      // Only a single thread can read the input.
      scanf("%c",&ch);

      // Multiple threads compute the results.
      printf("compute␣results\n");
    }
   return 0;
}
```

Synchronization and sharing variables

- Synchronization
    1. critical
    2. barrier
    3. atomic
    4. flush
    5. Example - parallel pi program

- Data environment
    1. stack and heap
    2. shared
    3. private
    4. firstprivate
    5. lastprivate
    6. reduction - parallel pi program
- Dijkstra Shortest-Path Algorithm

[fragile]

Synchronization:critical

A critical region applies to all current threads in the program, not just the threads in the current team. It restricts executions of the critical block to one thread at a time.

```
#pragma omp parallel shared(X,Y)
{
    #pragma omp critical
        Y=func(++X,Y);
}
```

[fragile]

Synchronization:barrier

The barrier directive has no statement associated with it. The barrier ensures that all threads in the team have reach the barrier before any thread is allowed to continue beyond the barrier.

```
#pragma omp parallel
{
        /* some work is done */
        #pragma omp barrier
        /* all previous work is done,
        now continue */
}
```

[fragile]

Synchronization:atomic

The atomic directive assures that updates to a variable are free from conflict. For example:

```
#pragma omp parallel for shared(a,b)
for(i=0;i<n;i++)
{
    #pragma omp atomic
        b+=a[i];
}
```

The kinds of updates are limited to a common arithmetic. A flush for the variable b is implicit on entry to and exit from the atomic region.

[fragile]

Memory Consistency and flush

- The compiler may store x in a register, and update the memory holding x only at certain points.
- In between such updates, since the memory location for x is not written to, the cache will be unaware of the new value, which thus will not be visible to other threads. This is important for shared variables.
- OpenMP takes a relaxed consistency approach, meaning that it forces updates to memory (flushes) at all synchronization points

[fragile]

Memory Consistency and flush

If two threads wish to communicate via a shared memory variable then the following must be ensured by the programmer:

- The value is written to the variable by the first thread.
- The variable is flushed by the first thread.
- The variable is flushed by the second thread.
- The value is read from the variable by the second thread.

[fragile]

Incorrect example of flush

```
Assuming a=b=0:
thread 1                    thread 2
b=1                         a=1
flush(b)                    flush(a)
flush(a)                    flush(b)
if (a==0) then              if (b==0) then
  critical section             critical section
```

The critical section, which should only be executed by one of the threads, is not safe. The compiler can reorder the flush(b) in thread 1 to be after the critical section since the variable b is not used in the condition (and assuming it is not used in the critical section itself). Similarly for thread 2. [fragile]

Correct example of flush

```
Assuming a=b=0:
thread 1                    thread 2
b=1                         a=1
flush(a,b)                  flush(a,b)
if (a==0) then              if (b==0) then
  critical section             critical section
```

The flush statements cannot be reordered and only one thread could enter the critical region. In this example, neither thread may be able to enter the critical region. [fragile]

Data environment: stack and heap

There are two types of memory to store your data, you can store your data in either stack or heap.

- Local variables are stored in stack. Global variables, static local variables, or memory allocated using malloc are in heap. The stack is always reserved in a LIFO (last in first out) order, stack allocated values are "deleted" once you leave the scope.
- The heap size is much larger than the size of stack, therefore it is good to storage large variables in heap, e.g. use char *a = malloc(1024) instead of char a[1024]
- In a multi-threaded situation each thread will have its own completely independent stack, so the threads will copy the stack variables but they will share the heap variables.

[fragile]

Data environment:shared and private

- Variables in shared context are visible to all threads running in associated parallel regions.
- Variables in private context are hidden from other threads. Each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads.

[fragile]

Data environment:shared and private

```
int E1;                        /* shared static      */
void main (argvc,...) {        /* argvc is shared    */
   int i;                      /* shared automatic   */
   void *p = malloc(...);      /*memory allocated by*/
          /* malloc is accessible by all threads*/
   #pragma omp parallel private (p)
   {
        int b;                 /* private automatic*/
        static int s;          /* shared static*/
        #pragma omp for
        for (i = 0;...)
        {          /* i is private here because it*/
                   /* is the iteration variable */
          b = 1;   /* b is still private here! */
        }
   }
 }
```

[fragile]

Data environment:Firstprivate Clause

Firstprivate is a special case of private.
It initializes each private copy with the corresponding value from the master thread

```
int main(void)
{
  int i = 0;
  int x;
  x=44;
  #pragma omp parallel for firstprivate(x)
  for (int i = 0; i < 10; i++) {
        x += i;
         printf("%d\n", x);
  }
  printf("x␣is␣%d\n", x);
}
```

[fragile]

Data environment:Lastprivate Clause

Lastprivate passes the value of a private from the last iteration to a global variable.

```
int main(void)
{
  int i = 0;
  int x;
  x=44;
```

```
  #pragma omp parallel for firstprivate(x)
        lastprivate(x)
  for (int i = 0; i < 10; i++) {
        x += i;
        printf("%d\n", x);
  }
  printf("x is %d\n", x);
}
```

[fragile]

Data environment:reduction

A private copy for each list variable is created and initialized for each thread.
At the end of the reduction, the reduction operator is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

```
main(int argc, char *argv[])  {
int   i, n, chunk;
float a[100], b[100], result;
/* Some initializations */

#pragma omp parallel for default(shared) \
  private(i) reduction(+:result)

  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
  printf("Final result= %f\n",result);
}
```

The Algorithm

Dijkstra algorithm is for finding the shortest paths from vertex 0 to the other vertices in an N-vertex undirected graph. [fragile]

Pseudocode

```
1  Done = {0} # vertices checked so far
2  NewDone = None # currently checked vertex
3  NonDone = {1,2,...,N-1} # vertices not checked
4  for J = 0 to N-1 Dist[J] = G(0,J) # initialize
5
6  for Step = 1 to N-1
7  find J so Dist[J] is min among all J in NonDone
8  transfer J from NonDone to Done
9  NewDone = J
```

```
10 for K = 1 to N-1
11  if K is in NonDone
12   # check if there is a shorter path from
13   # 0 to K through NewDone than our best so far
14   Dist[K]=min(Dist[K],Dist[NewDone]+G[NewDone,K])
```

Pseudocode

At each iteration, the algorithm finds the closest vertex J to 0 among all those not yet processed, and then updates the list of minimum distances to each vertex from 0 by considering paths that go through J. Two obvious potential candidate part of the algorithm for parallelization are the find J and for K lines.

Some fine tuning

```
#pragma omp parallel
{ int startv,endv, // start, end vertices for my thread
      step, // whole procedure goes nv steps
      mymv, // vertex which attains the min value in my chunk
      me = omp_get_thread_num();
  unsigned mymd; // min value found by this thread
  #pragma omp single
  { nth = omp_get_num_threads();  chunk = nv/nth; }
  startv = me * chunk; endv = startv + chunk - 1;
  for (step = 0; step < nv; step++) {
      #pragma omp single
      { md = largeint; mv = 0; }
      findmymin(startv,endv,&mymd,&mymv);
      // update overall min if mine is smaller
      #pragma omp critical
      { if (mymd < md)
          { md = mymd; mv = mymv; } }
      #pragma omp barrier
      // mark new vertex as done
      #pragma omp single
      { notdone[mv] = 0; }
     // now update my section of mind
     updatemind(startv,endv);
    #pragma omp barrier
    }
    }
```

```
Int *mymins; //(mymd,mymv) for
                //each thread;
```

```
mymins = malloc(2*nth*sizeof(int));
findmymin(startv,endv,&mymd,&mymv);
mymins[2*me] = mymd;
mymins[2*me+1] = mymv;
#pragma omp barrier
// mark new vertex as done
#pragma omp single
{ notdone[mv] = 0;
  for (i = 1; i < nth; i++)
    if (mymins[2*i] < md) {
        md = mymins[2*i];
        mv = mymins[2*i+1];
    }
  }
// now update my section of mind
updatemind(startv,endv);
```

Communication

Communication

Distributed memory architectures require message passing and this in turn leads to communication patterns. There are common communication patterns that support widely used parallel algorithms.

While parallel algorithm complexity is a governed by the machine's architecture upon which it executes, there are further aspects that can be considered, how messages are transmitted through the machine and what impact this can have on performance. These considerations need to address the communication patterns that are required by parallel algorithms.

Shared memory parallel programs don't explicitly use communication patterns though the underlying machine may, in the case of a distributed shared memory architecture.

A *message* is defined as a logic unit or packet of information. Messages may be instructions, data or control signals such as for synchronization.

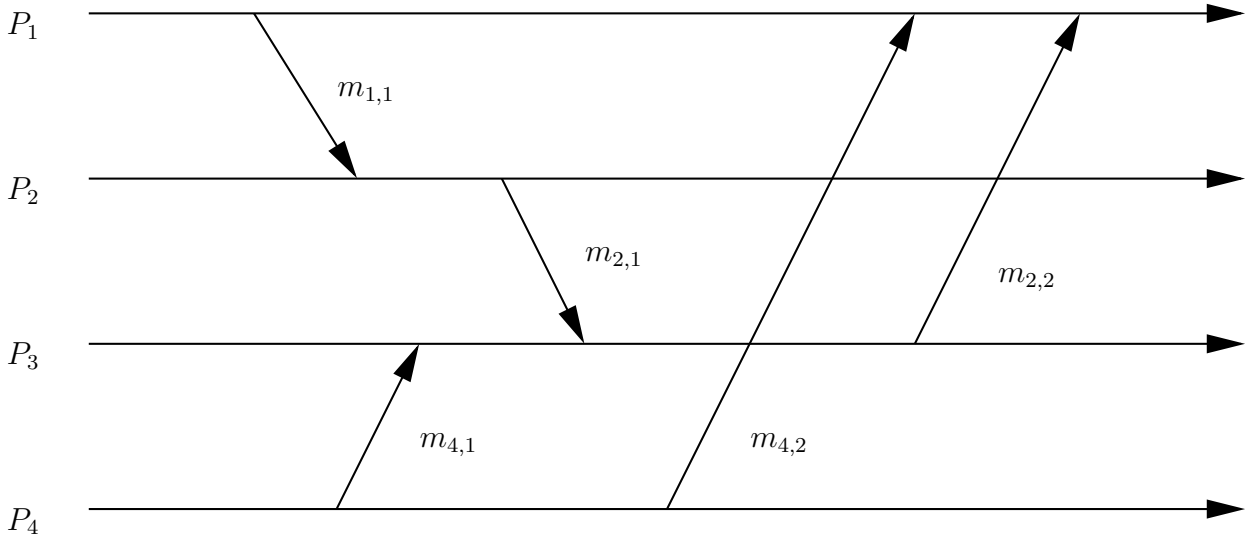


Figure 17: Communication between 4 processes or tasks.

Time to send a message

The time to send a message can be broken into a fixed tartup time, t_f, and a time per byte, t_b. The time to send the message is then

$$t_m = t_f + t_b l$$

where l is the number of bytes in the message.

The time t_f includes the time for the communication subsystem to prepare to send the message. This may include the time for the receiver to indicate that it is ready to receive the data, depending on the send/receive model being used.

- e.g., TPC's SYN + SYN/ACK + ACK 3-way handshake

For efficient message passing it is preferable to have $t_b\, l >> t_f$.

The *iPSC* message passing computer has $t_s = 4100\mu s$ and $t_b = 2.8\mu s/byte$. A different model, the *iPSC/860* has $t_s = 160\mu s$ and $t_b = 0.36\mu s/bytes$. The *CM-5* has $t_s = 86\mu s$ and $t_b = 0.12\mu s/byte$.

Total parallel run time

The parallel run time, t_p, is a combination of computation time, t_{comp}, and communication time, t_{comm}:

$$t_p = t_{comp} + t_{comm}.$$

For this, both times need to be in the same units. We usually convert t_p into seconds but sometimes we can convert t_{comm} into an equivalent number of computational steps.

The speedup can then be in the form:

$$S = \frac{t_s}{t_p} = \frac{t_s}{t_{comp} + t_{comm}}.$$

Delay hiding

- Communication and computation can happen at the same time
- Part of the art of parallel programming is making sure that all processes have enough data to keep them busy.
- It sometimes helps to forward results incrementally as they are generated
  - May incur extra fixed set-up costs, t_f

Granularity

The total time spent for computations divided by the total time spent communicating is the *granularity* of a task.

$$\text{granularity} = \frac{\text{computation time}}{\text{communication time}}$$

If the granularity is unity then the task spends half of its time in computations and half of its time communicating. Increasing the granularity can be done by decreasing the distribution, i.e. combining tasks into a single task; but this leads to decreased parallelism. A single task spends all of its time in computation and no communication time - apart from the initial communication to obtain the input parameters and the final communication to provide the result.

The time for a floating-point operation on the *iSPC* is $25\mu s$. Therefore 1,000,000 operations takes $25s$. If it sends a single message of about $8.9MB$ then the task has granularity of 1. For the *CM-5*, the time for a floating-point operation is $0.33\mu s$. Then 1,000,000 operations takes $0.33s$. If the same task on the *CM-5* sends an $8.9MB$ message, then the task granularity is about 0.3, i.e. about 23% of the time is spent doing the computation and the remaining 77% is communication.

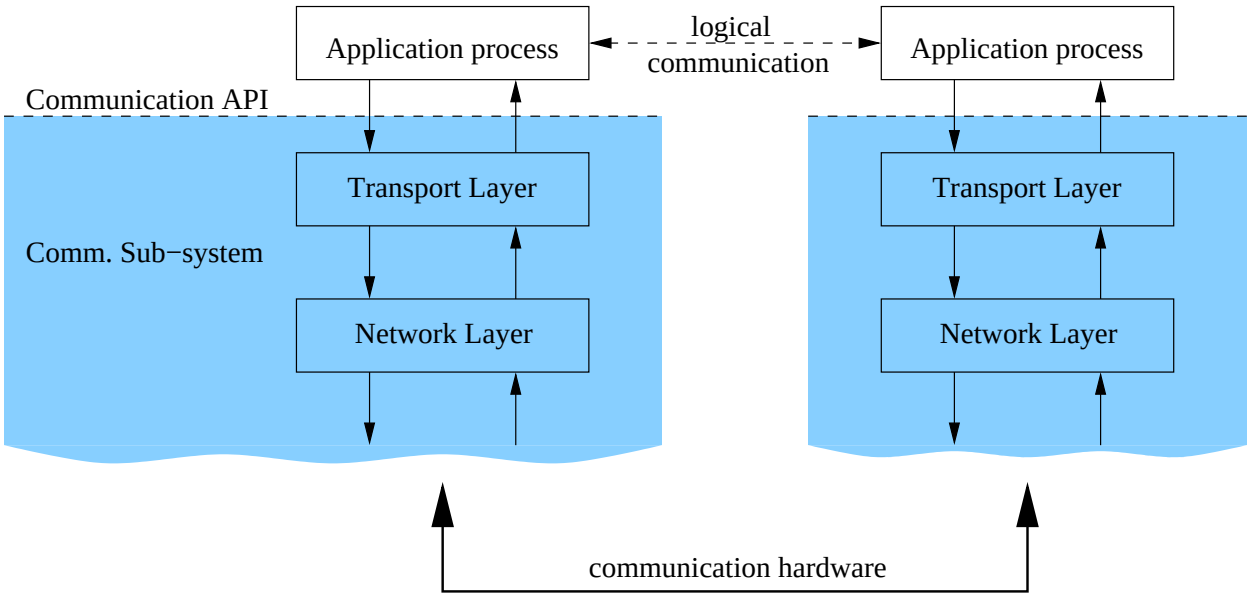


Figure 18: Generalized communication between two application processes.

Communication primitives

Figure 18 is using the *Open Systems Interconnection* (OSI) reference model, with the top three layers (Application, Presentation and Session) compressed into a single Application Layer. The predominant Internet Transport Layer interface uses IP/TCP/UDP or Internet Protocol, Transmission Control Protocol and Unreliable Datagram Protocol. TCP and UDP provide a connection-oriented and a connectionless communication service respectively. The `socket` interface to TCP/UDP, introduced by Berkeley Special Distribution (BSD) UNIX, is widely used.

A number of implementations of message passing systems use a TCP communication subsystem, e.g. cluster computing often uses this. In some cases, such as Myrinet, an alternative sub-system is provided. However the communication primitives remain the same. The primitive help to ensure that parallel programs are portable across different message passing systems.

At a higher level of abstraction the communication API is completely removed from the program logic, for example with *remote method invocation* (RMI) all objects register with the communication subsystem and then may invoke each others methods without regard to communication that must take place.

High level abstractions are not a "free lunch" – usually they come with a performance or functionality penalty since the application process no longer has access to communication specifics that may otherwise be used to optimize the application's individual communication needs.

Java RMI uses a registry server at each host in the network. The registry allows Java object to register their name. Objects are located through the registry.

The basic communication APIs consists of `send()` and `receive()` primitives. A sender uses `send()` to send data to a receiver that uses `receive()` to receive the data.

Details of the primitive arguments will vary, but the following arguments are common:

- destination identifier,

- message type,
- data type and contents.

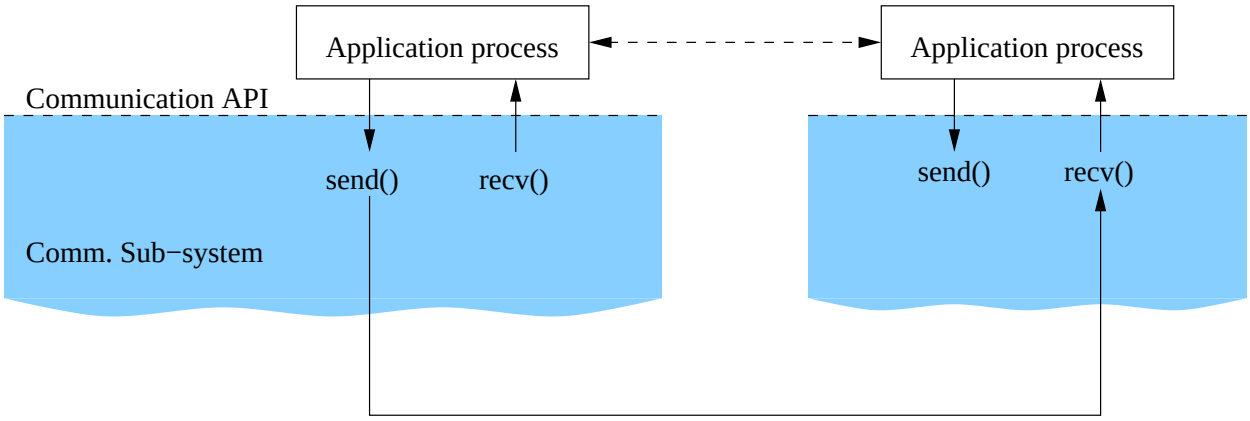


Figure 19: Send and receive primitives.

In some cases the `receive()` function is abbreviated as `recv()`.

The destination identifier needs to be known to the sender, either as a consequence of program design logic or using a dynamic lookup service. There may be a single destination or multiple destinations such a multicast or broadcast. Sometimes the word *groupcast* is used to represent a send to all members of a group. The message type allows for a different communication processes to take place which will be discussed further in. The data type and contents allows for different kinds of data to be represented in the communication.

Note that the send/receive paradigm is representative of a connectionless service.

Function calls associated with sending and receiving data are not usually provided as separate threads. This means that the caller will *block* until the function returns.

In a general sense functions are called *blocking* or *non-blocking*. A blocking function will not return until the service has been completed. A non-blocking function will return immediately, without requiring the service to be completed.

There are various assumptions made when deciding when a service has been "completed". In general, the application itself may block if there are insufficient resources to continue.

Figure 20 shows a blocking send and receive scenario. The sender calls `send()` (1). The send function sends the data (2). When the receiver's `recv()` returns (3), then it signals the sender's `send()` to return (4) which returns to the application process (5).

Note that the receiver's `recv()` is also blocking. The sender will block until the receiver calls `recv()` to receive the data. If the receiver calls `recv()` before the data has been sent then the receiver will block until the data arrives from the sender.

If the receiving subsystem does not buffer the data then the order of operations will be different. In this case, the data cannot be sent until the receiver has called `recv()`.

If the receiver doesn't call `recv()` then the sender will block for ever. If the receiver calls `recv()` but there is no sender then the receiver will block for ever.

The send and receive buffers are used to implement the appropriate protocol for the communication, e.g. reliable transmission will require a message to be retransmitted if it was received in error.

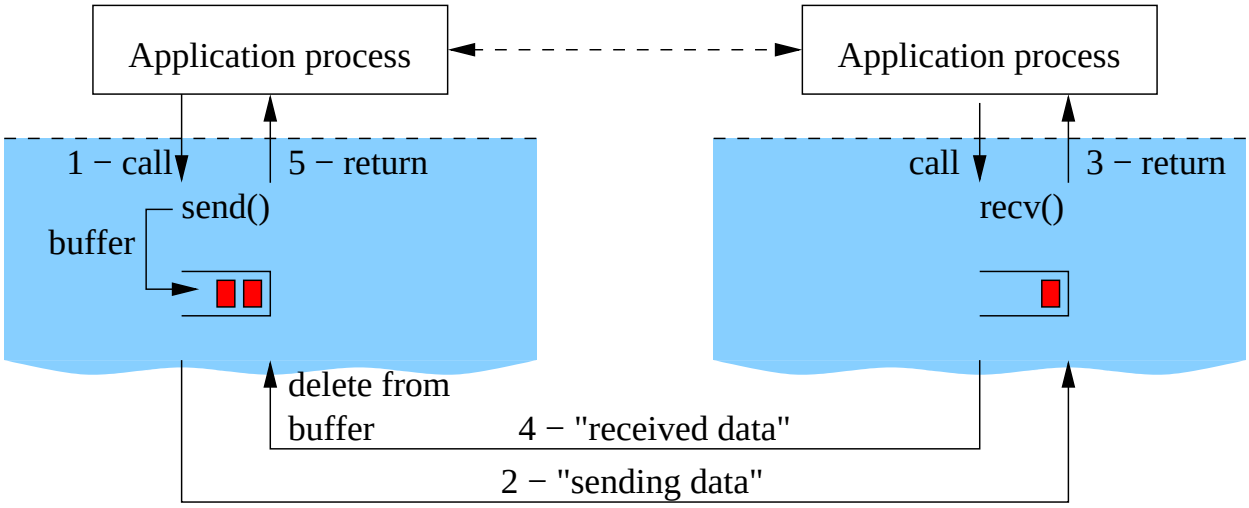


Figure 20: Blocking send.

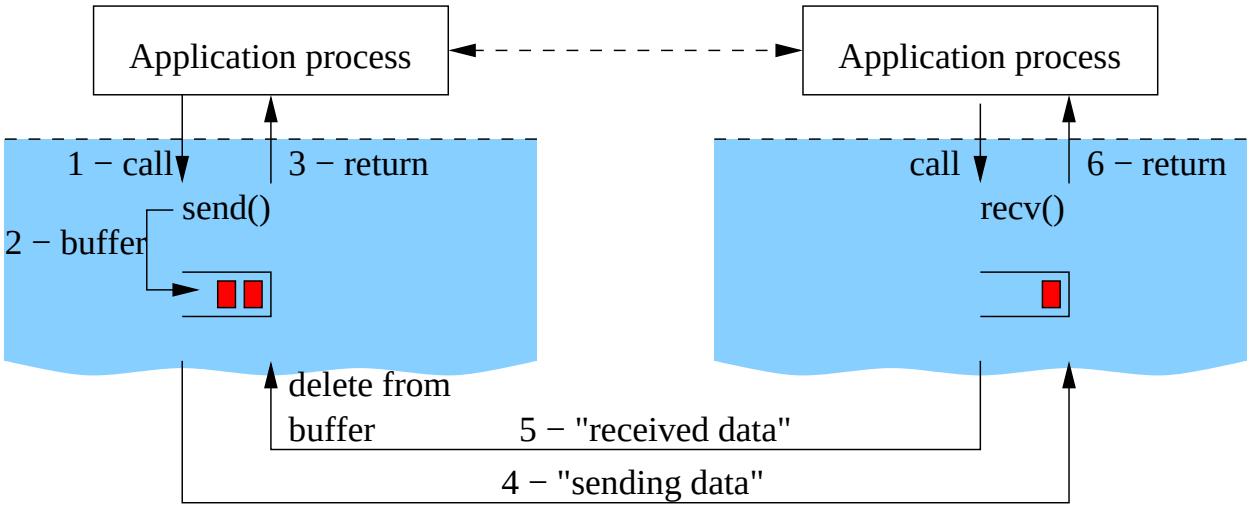


Figure 21: Local-blocking send.

The `send()` may return immediately after buffering the message as depicted in Figure 21. In this case the message has not yet been transmitted to the sender. This is called *local-blocking* because the `send()` function will not return immediately if the buffer is full.

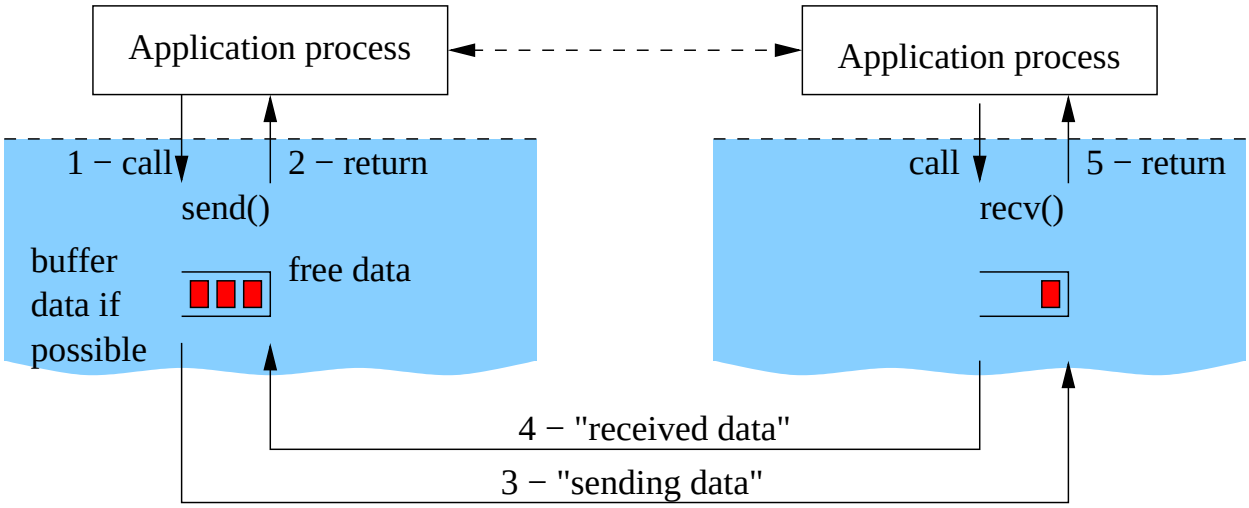


Figure 22: Non-blocking send.

Figure 22 depicts a non-blocking send. In this case the function is always "guaranteed" to return immediately. Here it is assumed that there is always enough resources available for the `send()` function to use. For example the message must be buffered by the application process and the `send()` function may simply record a pointer to the message.

The `recv()` is also either blocking or non-blocking. A blocking call will not return until a message has been received. A non-blocking call will return immediately with either a message (if one was buffered and waiting to be received) or no message.

There may be several messages waiting in the receive buffer to be received. The `recv()` function will usually receive one message per call.

Use of a non-blocking `recv()` function is similar to polling a device for I/O. It wastes CPU cycles if called too many times without a message being received.

The significant difference between blocking and both local-blocking and non-blocking is that the sender can be **sure** that the receiver has received the data when a blocking send is used. This is useful for loose synchronization.

Blocking and non-blocking connote *synchronous* and *asynchronous* communication.

The communication API needs to stipulate various axioms or quality of service about the message passing service provided. For example, if message A is sent and subsequently message B is sent to a receiver with non-blocking sends, is the receiver guaranteed to receive A before B?

The unicast sends a single data item from a single node to a single node.

The broadcast sends a single data item from a single node to all nodes.

The gather sends a single data item from each node to the same, single node.

The scatter sends each element from an array of data to a different node.

The gather/broadcast sends a data item from each node to every node.

The gather/scatter sends each element from arrays at every node to a different node.

Table 1: Communication patterns

communication	source	destination
unicast	one node	one node
broadcast	one node; one item	all nodes; item per node
gather	all nodes; item per node	one node; N items
scatter	one node; N items	all nodes; item per node
gather/broadcast	all nodes; item per node	all nodes; N items per node
gather/scatter	all nodes; N items per node	all nodes; N items per node
reduce	all nodes; arithmetic combining	one node; one item
reduce/broadcast	all nodes; arithmetic combining	all nodes; same item in each
prefix	all nodes; arithmetic combining	all nodes; different item in each

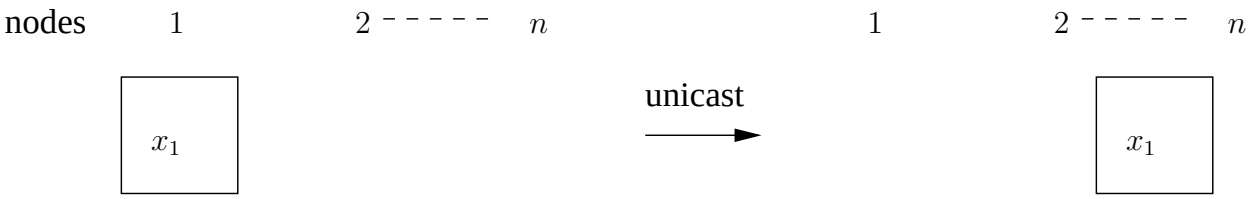


Figure 23: Unicast.

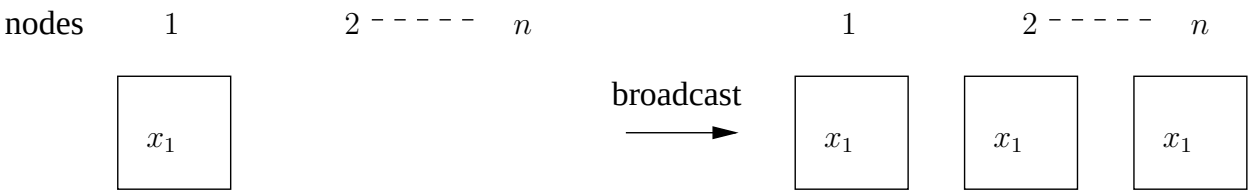


Figure 24: Broadcast.

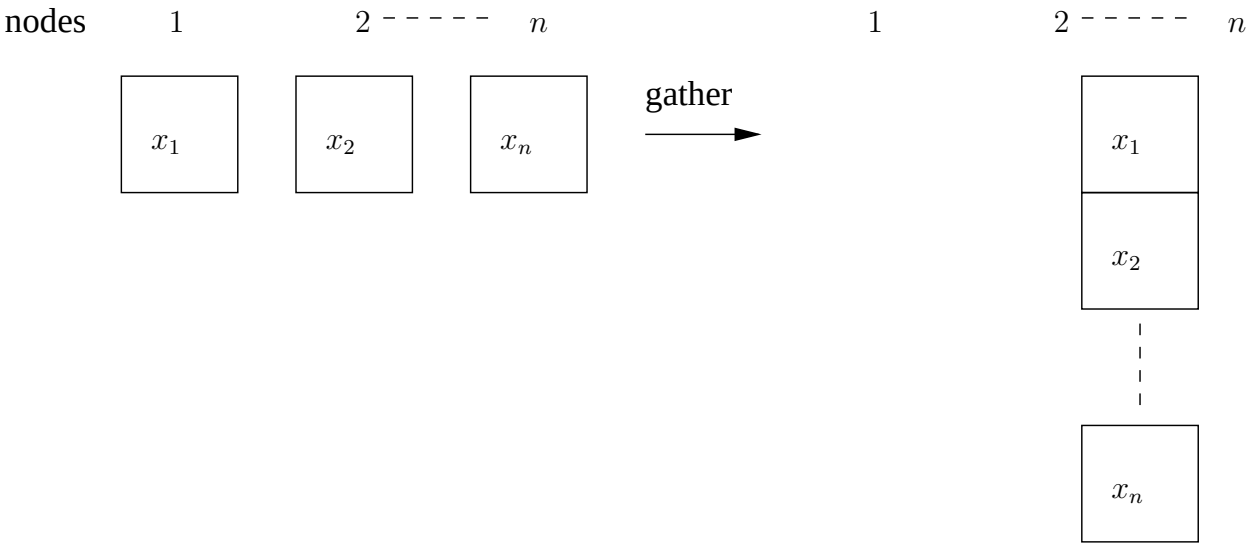


Figure 25: Gather.

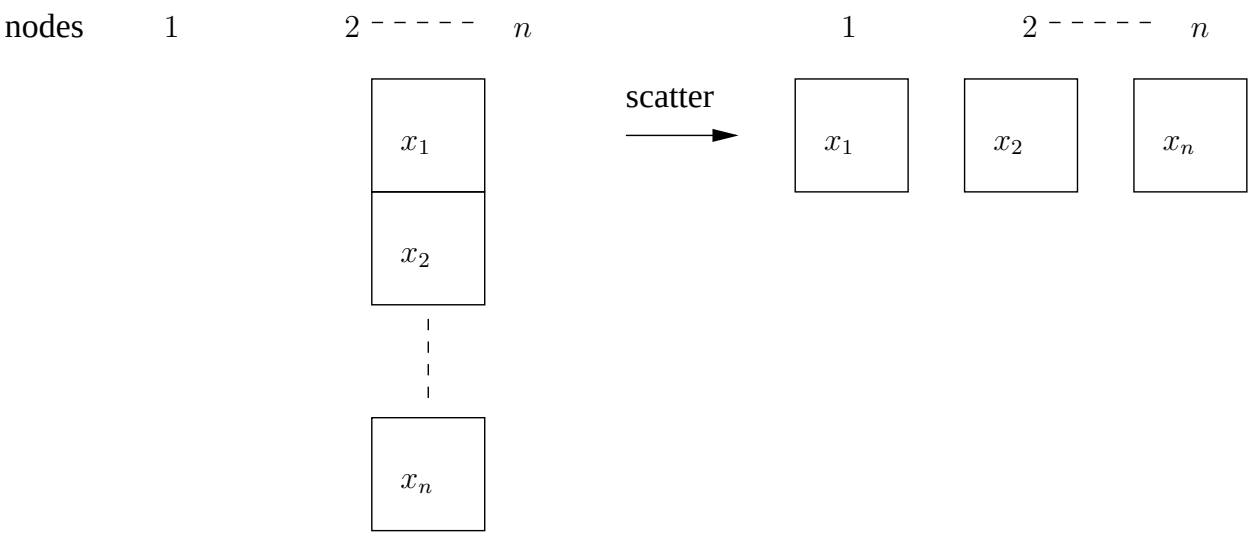


Figure 26: Scatter.

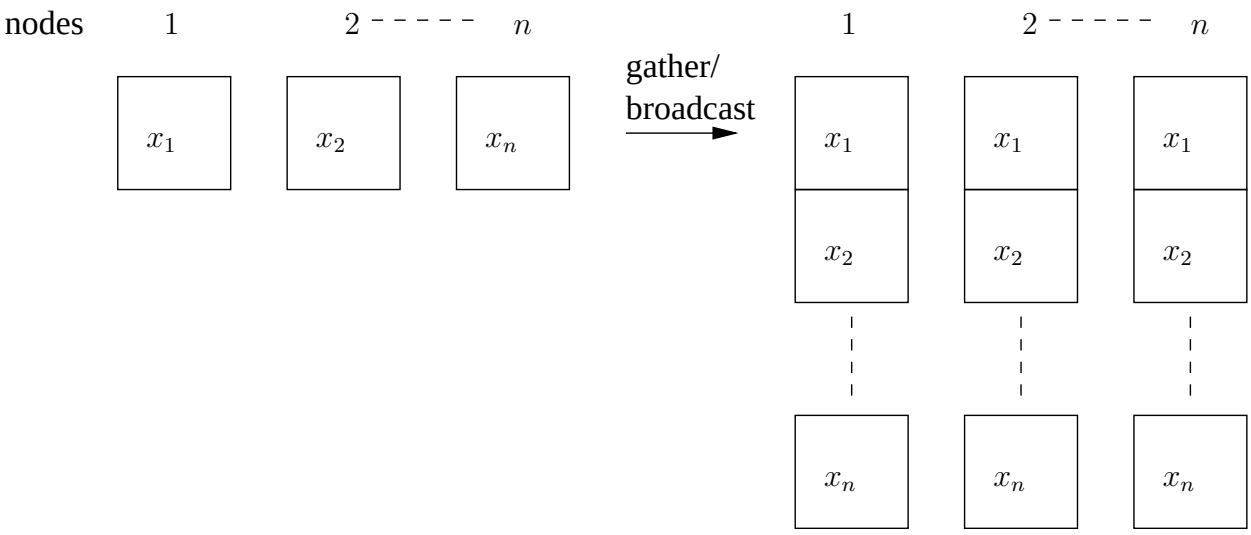


Figure 27: Gather/broadcast.

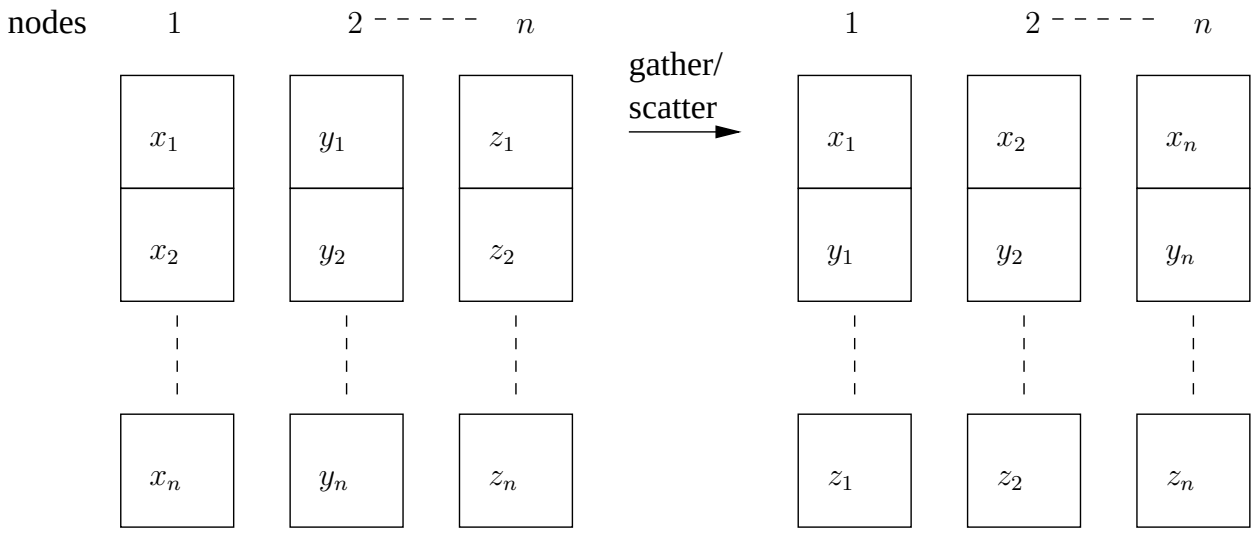


Figure 28: Scatter/gather.

The reduce sends the result of a mathematical function over the data items from each node to a single node.

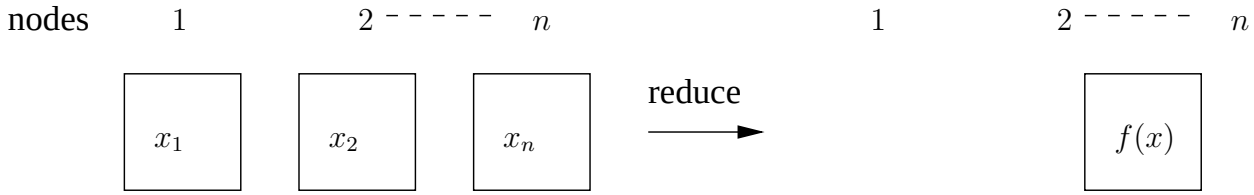


Figure 29: Reduce.

The reduce sends the result of a mathematical function over the data items from each node to all nodes.

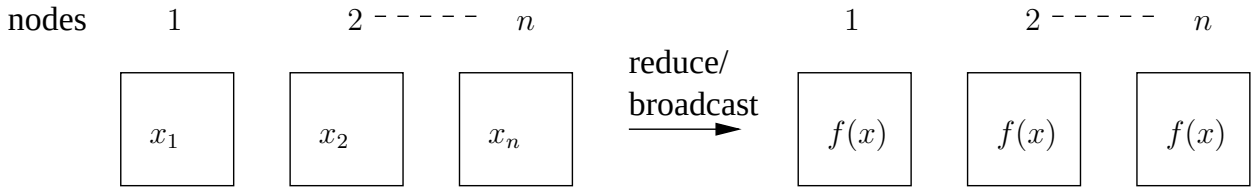


Figure 30: Reduce/broadcast.

The prefix sends the result of a different mathematical function over the data items to each of the nodes.

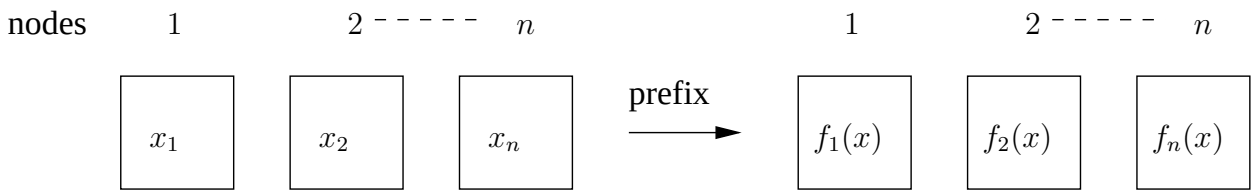


Figure 31: Prefix.

MPI Tutorial
with contributions from Xinyi Xu

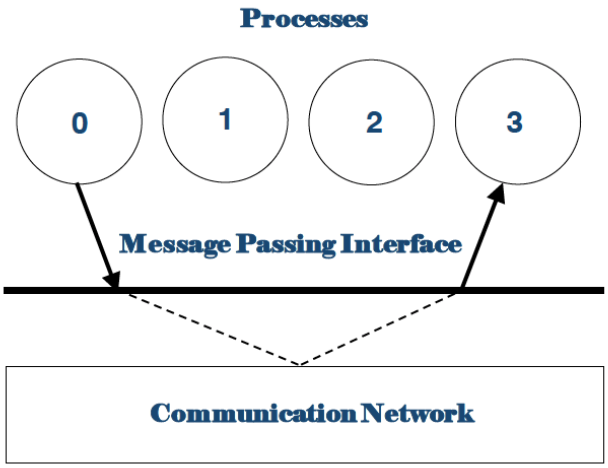
Overview

- MPI Introduction
  - Message Passing Model
  - Communication modes
  - MPI basics
- Point-to-Point Communications
  - Sending a message
  - Receiving a message
  - ping example
- Collective Communications
  - Barrier Synchronization
  - Broadcast, scatter, gather
  - Reduce - average example

Message Passing Model

- The message passing model is based on the notion of processes
- In the message passing model, parallelism is achieved by having many processes co-operate on the same task
- Each process has access only to its own data
- Processes communicate with each other by sending and receiving messages

Parallel Paradigm



Reference: http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/

Messages

- A message transfers a number of data items of a certain type from the memory of one process to the memory of another process

- A message typically contains
  - the ID of the sending processor
  - the ID of the receiving processor
  - the type of the data items
  - the number of data items
  - the data itself
  - a message type identifier

Communication modes

- Sending a message can either be synchronous or asynchronous
  - More options in the standard `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`
- A synchronous send is not completed until the message has started to be received
- An asynchronous send completes as soon as the message has gone

- Receives are usually synchronous - the receiving process must wait until the message arrives
  - Why?

What is MPI

- First message-passing interface standard.
- Message Passing Interface document produced in 1993
- MPI is a library of functions/subroutine calls
- MPI is not a language, there is no such thing as an MPI compiler
  - CUDA refers to an MPI compiler. It means something like `mpicxx`
  - This is just a wrapper for a C++ compiler that gives the paths to the MPI library and headers

Initializing and exiting MPI

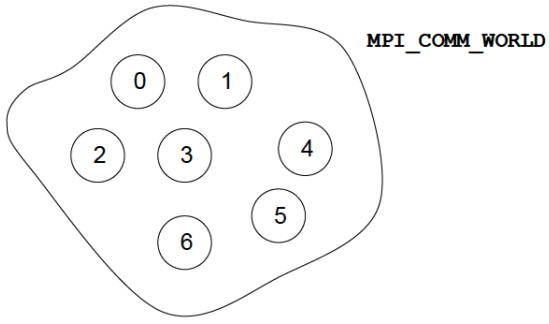
```
int MPI_Init(int *argc, char ***argv)
```

- Initializes the MPI execution environment.
- Must be the first MPI procedure called.
- Note the extra `*` in front of `argc` and `**argv`.

```
int MPI_Finalize()
```

- Terminates the MPI execution environment.
- Must be the last MPI procedure called, no other MPI routines may be called after it.

Communicators



- MPI uses *communicators* to define which collection of processes may communicate with each other.
- Most MPI routines require you to specify a communicator as an argument.
- `MPI_COMM_WORLD` is the predefined communicator that includes all of your MPI processes.

Reference:http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/

Rank and Size

How do you identify which process within a communicator you are?

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

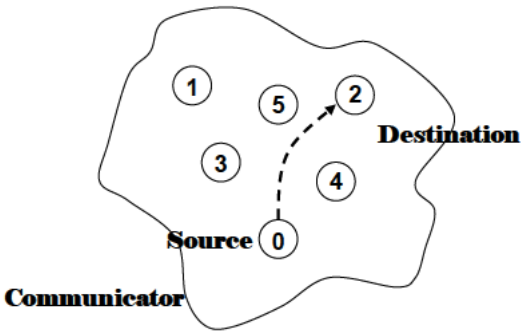
Numbering is always 0, 1, 2, ..., N-1.

- Can't compare with other communicators

How many processes are contained within a communicator?

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

Point-to-Point Communication



- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator.
- Destination process is identified by its rank in the communicator.

Point-to-Point Communication

- Sender calls a SEND routine
- Receiver calls a RECEIVE routine
- Data goes into the receive buffer
- Metadata describing message also transferred

Sending a message

Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse.

```
int MPI_Send(void *buf, int count,
                        MPI_Datatype datatype,
                        int dest, int tag,
                        MPI_Comm comm)
```

E.g. send data from rank 1 to rank 3

```
int x;
...
if (rank == 1)
MPI_Send(&x, 1, MPI_INT, /*dest=*/3, /*tag=*/0,
                  MPI_COMM_WORLD);
```

Receiving a message

Receive a message and block until the requested data is available in the application buffer in the receiving task.

```
int MPI_Recv(void *buf, int count,
                        MPI_Datatype datatype,
                        int source, int tag,
                        MPI_Comm comm,
                        MPI_Status *status)
```

Status indicates the source of the message, the tag of the message, and actual number of bytes received
E.g. Receive data from rank 1 on rank 3

```
int y;
...
if (rank == 3)
MPI_Recv(&y, 1, MPI_INT, /*src=*/1, /*tag=*/0,
                  MPI_COMM_WORLD, &status);
```

For a communication to succeed

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message types must match.
- Receiver's buffer must be large enough.
  - Need not hold the whole message, if the receiving process is extracting while the sender is sending

Collective Communications

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples
  - Barrier synchronisation.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

Barrier Synchronization

Synchronization operation. Creates a barrier synchronization in a group.

```
int MPI_Barrier (MPI_Comm comm)
```

- When reaching the `MPI_Barrier` call, each task blocks until all tasks in the group reach the same `MPI_Barrier` call.
- Used less than in shared-memory synchronization, because we're not waiting for data structures in memory to become ready.
- Useful if we are sharing an OS resource that is not controlled by MPI
- e.g., wanting to write output in the "correct" order `stackoverflow.com/questions/13305814/when-do-i-need-to-use-mpi-barrier`

Broadcast

Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

```
int MPI_Bcast (void *buffer, int count,
                        MPI_Datatype datatype,
                        int root,
                        MPI_Comm comm)
```

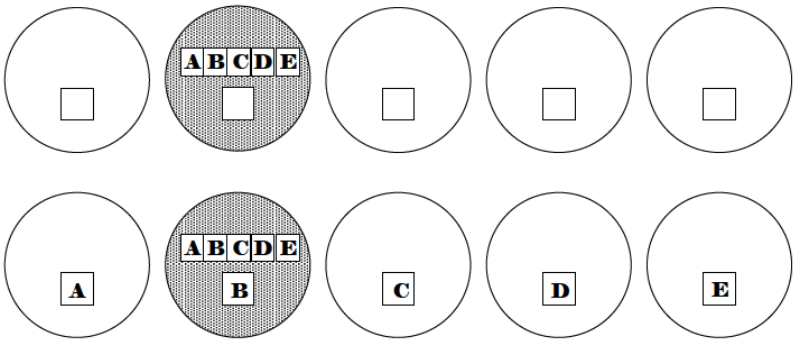
E.g.

```
MPI_Bcast(overallmin,2,MPI_INT,0, MPI_COMM_WORLD);
```

All the nodes in the group execute this line. The only difference is the action; most nodes participate by receiving, while node 0 participates by sending. [fragile]

Scatter

It breaks long data into chunks which it parcels out to individual nodes (including itself).



Reference:http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/

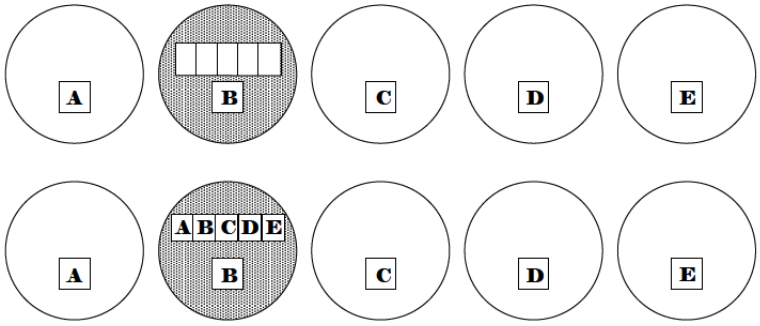
Scatter

```
int MPI_Scatter(void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcount,
                MPI_Datatype recvtype, int root,
                MPI_Comm comm)
```

Examples: `scatter.c`
(found at `.../mpi/scatter.c`)

Gather

Stringing everything together in node order and depositing it all in the program running at Node root.



Reference:http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/

Gather

```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

- all nodes participate in a gather operation
- each node (including Node root) contributes `sendcount` MPI integers
- from a location `sendbuf`
- Node root then receives `sendcount` items sent from each node

```
MPI_Allgather (&sendbuf,sendcount,sendtype,
        &recvbuf,recvcount,recvtype,comm)
```

- places the result at all nodes, not just one.

Reduce

Applies a reduction operation on all tasks in the group and places the result in one task.

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
        int count, MPI_Datatype datatype,
        MPI_Op op, int root, MPI_Comm comm)
```

E.g.

```
MPI_Reduce(mysum,overallsum,1,MPI_INT,MPI_SUM,
        0,MPI_COMM_WORLD);
```

- type of reduce operation is `MPI_SUM` (sum value)

- Each node contributes a value to be checked, from a location mysum
- type of the pair is `MPI_INT`
- The overall sum value will be computed by combining all of these values at node 0, where they will be placed at a location overallsum

Example found at `.../mpi/reduce_avg.c`

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
           MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
}
```

Overview

- MPI modes (Ssend, Bsend and Send)
- Meaning and use of communicator
- Multithreading

Modes

- `MPI_Send` (standard Send)
  - may be implemented as synchronous or asynchronous send
  - this may cause a lot of confusion
  - also the most efficient, since it gives the most flexibility to the system
- `MPI_Ssend` (Synchronous Send)
  - guaranteed to be synchronous
  - routine will not return until message has been delivered

- `MPI_Bsend` (Buffered Send)
  - guaranteed to be asynchronous
  - routine returns before the message is delivered
  - system copies data into a (user-supplied) buffer and sends it later on

Synchronous send

Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

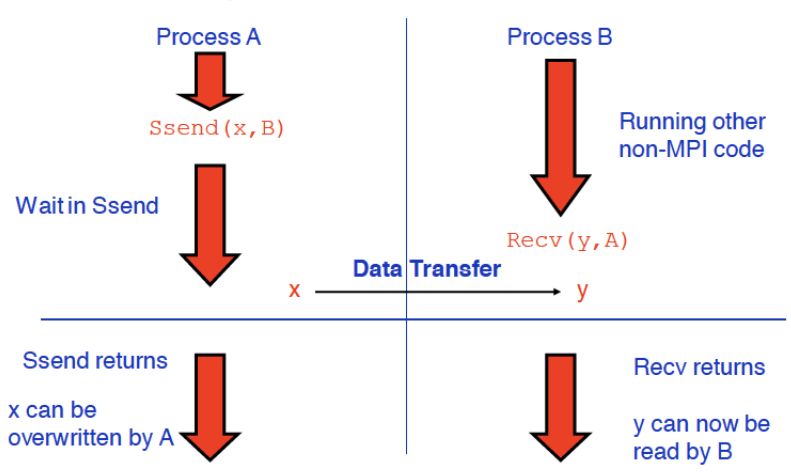
```
int MPI_Ssend(void *buf, int count,
              MPI_Datatype datatype,
              int dest, int tag,
              MPI_Comm comm)
```

Asynchronous send

This routine is a buffered mode send, routine returns before the message is delivered.

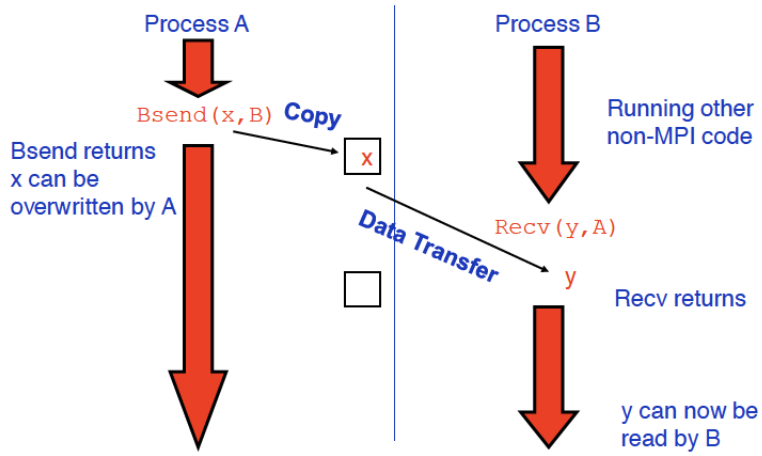
```
int MPI_Bsend(
              void *buf, int count,
              MPI_Datatype datatype,
              int dest, int tag,
              MPI_Comm comm )
```

MPI_Ssend



reference:http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/

MPI_Bsend



reference: http://archer.ac.uk/training/course-material/2014/07/MPI_Edi/

MPI_Send

MPI_Send tries to solve these problems

- buffer space is provided by the system
- Send will normally be asynchronous (like Bsend)
- if buffer is full, Send becomes synchronous (like Ssend)

Message Matching(I)

```
Rank 0:
        Ssend(msg1, dest=1, tag=1)
        Ssend(msg2, dest=1, tag=2)
Rank 1:
        Recv(buf1, src=0, tag=1)
        Recv(buf2, src=0, tag=2)
```

- buf1 = msg1; buf2 = msg2
- Sends and receives correctly matched

Message Matching(II)

```
Rank 0:
        Ssend(msg1, dest=1, tag=1)
        Ssend(msg2, dest=1, tag=2)
Rank 1:
        Recv(buf2, src=0, tag=2)
        Recv(buf1, src=0, tag=1)
```

- Deadlock (due to synchronous send)

- Sends and receives incorrectly matched

Message Matching(III)

```
Rank 0:
        Bsend(msg1, dest=1, tag=1)
        Bsend(msg2, dest=1, tag=1)
Rank 1:
        Recv(buf1, src=0, tag=1)
        Recv(buf2, src=0, tag=1)
```

- buf1 = msg1; buf2 = msg2

- Messages have same tags but matched in order

Message Matching(IV)

```
Rank 0:
        Bsend(msg1, dest=1, tag=1)
        Bsend(msg2, dest=1, tag=2)
Rank 1:
        Recv(buf2, src=0, tag=2)
        Recv(buf1, src=0, tag=1)
```

- buf1 = msg1; buf2 = msg2

- Do not have to receive messages in order!

Message Matching(V)

```
Rank 0:
        Bsend(msg1, dest=1, tag=1)
        Bsend(msg2, dest=1, tag=2)
```

```
Rank 1:
        Recv(buf1, src=0, tag=MPI_ANY_TAG)
        Recv(buf2, src=0, tag=MPI_ANY_TAG)
```

- buf1 = msg1; buf2 = msg2

- Messages guaranteed to match in send order
  examine status to find out the actual tag values

Uses of Communicator

- Can split `MPI_COMM_WORLD` into pieces
  - each process has a new rank within each sub-communicator
  - guarantees messages from the different pieces do not interact

```
MPI_Comm_split(
        MPI_Comm comm,
        int color,
        int key,
        MPI_Comm* newcomm)
```

Example of using multiple communicators

```
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
       world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

Uses of Communicator

- Can make a copy of `MPI_COMM_WORLD`
  e.g. call the `MPI_Comm_dup` routine
  containing all the same processes but in a new communicator

- Enables processes to communicate with each other safely within a piece of code
  guaranteed that messages cannot be received by other code

Multithreading

MPI libraries vary in their level of thread support:

- `MPI_THREAD_SINGLE` - Level 0: Only one thread will execute.
- `MPI_THREAD_FUNNELED` - Level 1: The process may be multi-threaded, but only the main thread will make MPI calls - all MPI calls are funneled to the main thread.
- `MPI_THREAD_SERIALIZED` - Level 2: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time. That is, calls are not made concurrently from two distinct threads as all MPI calls are serialized.
- `MPI_THREAD_MULTIPLE` - Level 3: Multiple threads may call MPI with no restrictions.

SINGLE

There are no threads in the system
E.g., there are no OpenMP parallel regions

```
int main(int argc, char ** argv)
{
 int buf[100];
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 for (i = 0; i < 100; i++)
  compute(buf[i]);
 /* Do MPI stuff */
 MPI_Finalize();
 return 0;
}
```

FUNNELED

All MPI calls are made by the master thread * Outside the OpenMP parallel regions * In OpenMP master regions

```
int main(int argc, char ** argv)
{
 int buf[100], provided;
 MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED,
         &provided);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 #pragma omp parallel for
  for (i = 0; i < 100; i++) {
   compute(buf[i]);
  }
 /* Do MPI stuff */
 MPI_Finalize();
 return 0;
}
```

SERIALIZED

Only one thread can make MPI calls at a time
Protected by OpenMP critical regions

```
int main(int argc, char ** argv)
{
 int buf[100], provided;
 MPI_Init_thread(&argc, &argv,
        MPI_THREAD_SERIALIZED, &provided);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 #pragma omp parallel for
  for (i = 0; i < 100; i++) {
   compute(buf[i]);
   #pragma omp critical
     /* Do MPI stuff */
 }
 MPI_Finalize();
 return 0;
}
```

MULTIPLE

Any thread can make MPI calls any time

```
int main(int argc, char ** argv)
{
 int buf[100], provided;
 MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 #pragma omp parallel for
```

```
  for (i = 0; i < 100; i++) {
   compute(buf[i]);
   /* Do MPI stuff */
 }
 MPI_Finalize();
 return 0;
}
```

Compile and run

- Use `mpicc -fopenmp [myprogram.c] -o myprogram` to compile
- Parallel jobs using MPI can be run using mpiexec or mpirun
- Mpiexec uses the task manager library of PBS to spawn copies of the executable on the nodes in a PBS allocation.

```
mpiexec [-n X] [-pernode]
      [-cpus-per-proc #perproc] [program]
```

where `X` is number of copies of `program`,
`[-pernode]` launches one process each node or `[-npernode #pernode]` `[-cpu-per-proc #perproc]` controls the number of cores per process.

Overview

- Topologies
  1. Cartesian
- MPI Derived Types
  - Vectors
  - Structs
  - Others

The what and why of topologies

- Specifying which processes are "neighbours" of each other
- New communicators representing neighbours
- Convenient process naming
- Naming scheme to fit the communication pattern

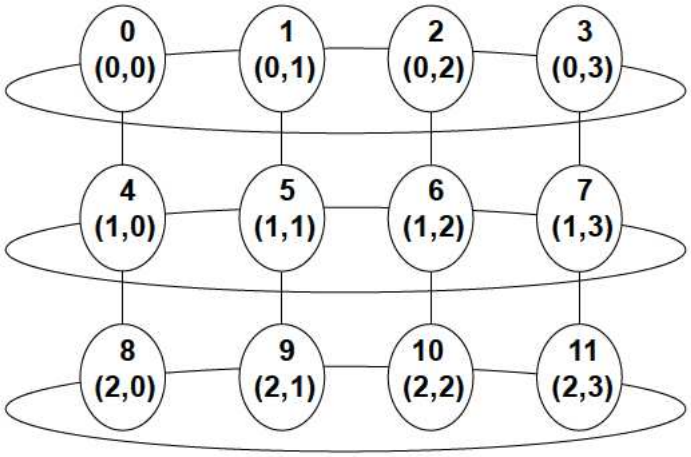
- Simplifies writing of code.
- Can allow MPI to optimise communications.

`http://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-topo.html`

How to use topologies

- Creating a topology produces a new communicator.
- MPI provides "mapping functions".
- Mapping functions compute processor ranks, based on the topology naming scheme.

Example



A 2-dimensional Cylinder

Topology types

- Cartesian topologies
  - each process is connected to its neighbours in a virtual grid.
  - boundaries can be cyclic, or not.
  - optionally re-order ranks to allow MPI implementation to optimise for underlying network interconnectivity.
  - processes are identified by cartesian coordinates.

Creating a Cartesian Virtual Topology

```
int MPI_Cart_create(MPI_Comm comm_old,
        int ndims, int *dims, int *periods,
        int reorder, MPI_Comm *comm_cart)
```

Balanced Processor Distribution

```
int MPI_Dims_create(int nnodes, int ndims,
        int *dims)
```

Cartesian Mapping Functions

- Call tries to set dimensions as close to each other as possible

dims before the call	function call	dims on return
(0, 0)	MPI_DIMS_CREATE(6, 2, dims)	(3, 2)
(0, 0)	MPI_DIMS_CREATE(7, 2, dims)	(7, 1)
(0, 3, 0)	MPI_DIMS_CREATE(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	MPI_DIMS_CREATE(7, 3, dims)	erroneous call

- Non zero values in dims sets the number of processors required in that direction.

Cartesian Mapping Functions

Mapping process grid coordinates to ranks

```
int MPI_Cart_rank(MPI_Comm comm,
        int *coords, int *rank)
```

Mapping ranks to process grid coordinates

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
        int maxdims, int *coords)
```

Cartesian Mapping Functions

Computing ranks of my neighbouring processes

```
int MPI_Cart_shift(MPI_Comm comm,
        int direction, int disp,
        int *rank_source, int *rank_dest)
```

```
int MPI_Sendrecv_replace(void* buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)

e.g.,
MPI_SendRecv_replace(A, 1, MPI_REAL, dest, 0,
        source, 0, comm, status, ierr)
```

Non-existent ranks

Computing ranks of my neighbouring processes

- What if you ask for the rank of a non-existent process? or look off the edge of a non-periodic grid?
- MPI returns a NULL processor - rank is `MPI_PROC_NULL`

Derived Types

- MPI Derived Types
  - Vectors
  - Structs
  - Others

Motivation

- Send / Recv calls need a datatype argument
- What about types defined by a program?
  e.g., structures (in C)
- Send / Recv calls take a count parameter
  what about data that isnt contiguous in memory?
  e.g., subsections of 2D arrays

Approach

- Can define new types in MPI
  - User calls setup routines to describe new datatype to MPI
  - MPI returns a new datatype handle

  - Store this value in a variable, eg `MPI_MY_NEWTYPE`

- Derived types have same status as pre-defined
  - Can use in any message-passing call
- Some care needed for reduction operations
  - User must also define a new `MPI_Op` appropriate to the new datatype to tell MPI how to combine them

Defining types

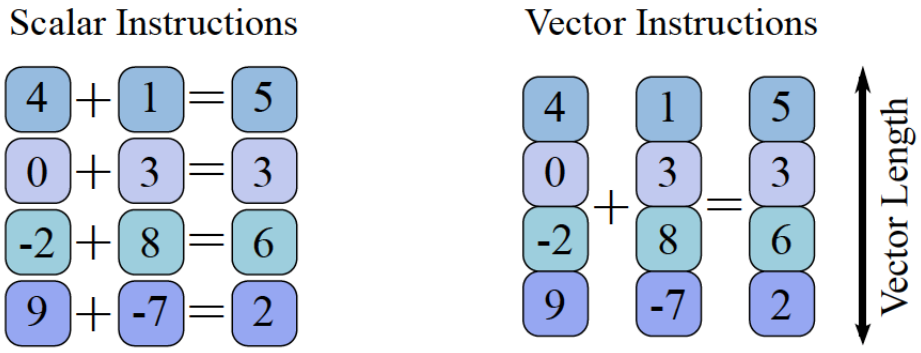
- All derived types stored by MPI as a list of basic types and displacements (in bytes)
  - for a structure, types may be different
  - for an array subsection, types will be the same
- User can define new derived types in terms of both basic types and other derived types

Contiguous Data

The simplest derived datatype consists of a number of contiguous items of the same datatype.

```
int MPI_Type_contiguous(int count,
        MPI_Datatype oldtype,
        MPI_Datatype *newtype)
```

Vector Datatype Example



- count = 2
- stride = 5
- blocklength = 3

What is a vector type?

Why is a pattern with blocks and gaps useful?

A vector type corresponds to a subsection of a 2D array

Think about how arrays are stored in memory

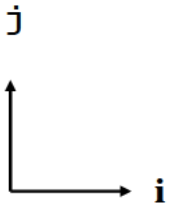
Array Layout In Memory

C: x[16]

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

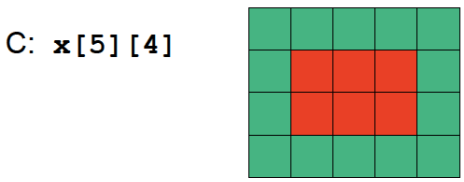
C: x[4][4]

4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13



Data is contiguous in memory

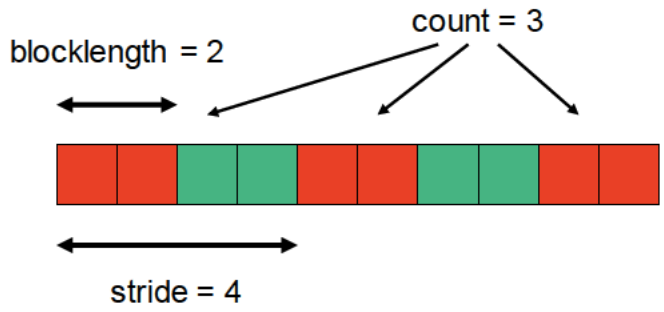
C example



A 3 X 2 subsection of a 5 x 4 array

- three blocks of two elements separated by gaps of two

Equivalent Vector Datatypes



[fragile]

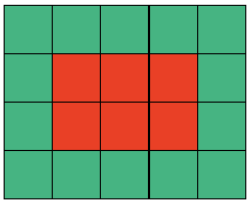
Constructing a Vector Datatype

```
int MPI_Type_vector (int count,
        int blocklength, int stride,
        MPI_Datatype oldtype,
        MPI_Datatype *newtype)
```

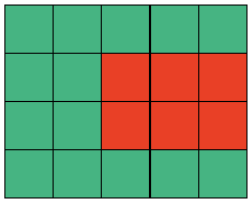
- Have defined a 3x2 subsection of a 5x4 array
  - but not defined WHICH subsection
  - is it the bottom left-hand corner? top-right?
- Data that is sent depends on what buffer you pass to the send routines
  - pass the address of the first element that should be sent

Constructing a Vector Datatype

```
MPI_Ssend(&x[1][1], 1, vector3x2, ...);
```



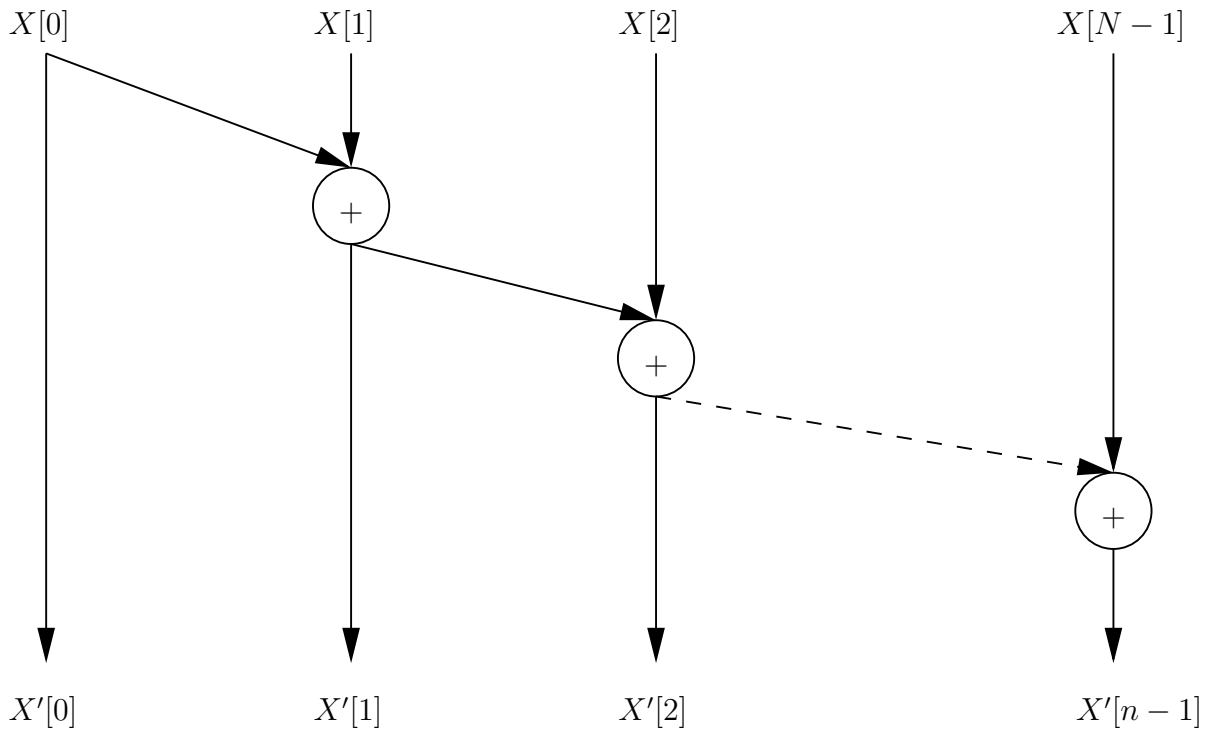
```
MPI_Ssend(&x[2][1], 1, vector3x2, ...);
```



Committing a Datatype

Once a datatype has been constructed, it needs to be committed before it is used in a message-passing call.
This is done using

```
MPI_TYPE_COMMIT
C:
int MPI_Type_commit (MPI_Datatype *datatype)
```



Prefix Sum

Prefix problem

The prefix problem requires computing the prefix sum (or other dyadic operation) on an array of elements:

```
int X[n];
```

⋮

```
for(i=1;i<n;i++)

    X[i] = X[i-1] + X[i];
```

The data dependency suggests flow dependence in this computation.

Consider computing the starting page number in a book for each chapter, knowing the size of the chapters. Similarly if the days of the year are numbered, then consider finding which days start each month, knowing the size of the months.

- The sequential prefix problem has a *size* of $n-1$ because there are $n-1$ operations in total.
- The depth of the sequential prefix problem is $n-1$ because it requires $n-1$ time steps to complete.

- It is possible to reduce the depth of the prefix problem, but only at the expense of increasing the size.

Upper/Lower parallel prefix algorithm

The upper/lower algorithm uses a *divide and conquer* approach to recursively divide the problem into smaller sub-problems. The results of the sub-problems are then combined to give the final output.

Divide and conquer is a fundamental approach for obtaining parallelism.

The upper/lower algorithm divides the lower half of the elements from the upper half of the elements. The prefix sum is independently computed on the lower and upper halves by recursively applying the divide and conquer. The output from the highest element of the lower half is added to each output of the upper half.

Description: For array $X = [x_0 \dots x_{n-1}]$ returns an array $S = [s_0 \dots s_{n-1}]$ of sums where $s_i = \sum_{j=0}^{i} x_j$.
Analysis: $\Theta(\log n)$
Processors: n

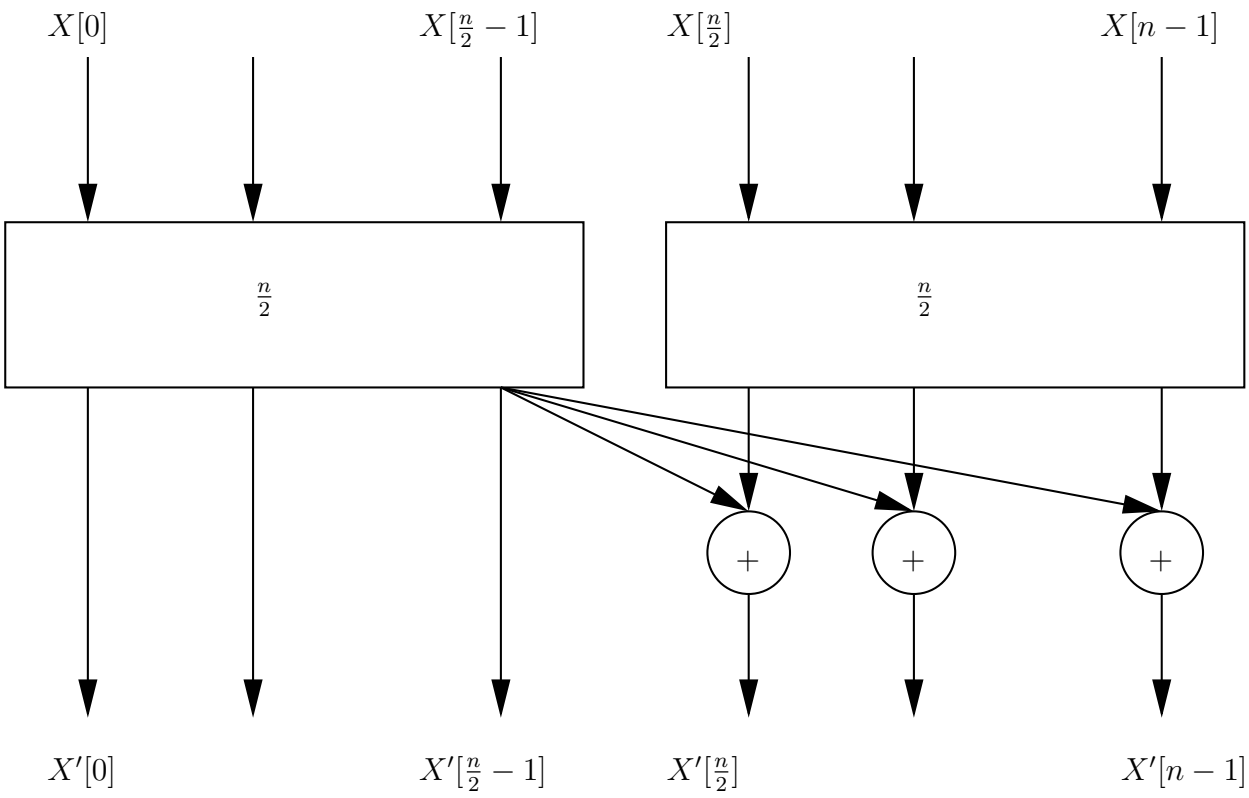
1: **procedure** $\textsc{UpperLower}^{\diamondsuit}_{\textsc{crew}}(X, n)$
2: **if** $n = 1$ **then**
3: $S[0] \leftarrow X[0]$
4: **return** S
5: **for** $i \leftarrow 0$ **to** $n-1$ **do in parallel**
6: $\frac{n}{2}$ **processor array** $0 \dots \frac{n}{2} - 1$ **does**
7: $S\left[0 \dots \frac{n}{2} - 1\right] \leftarrow \textsc{UpperLower}^{\diamondsuit}_{\textsc{crew}}(X\left[0 \dots \frac{n}{2} - 1\right], \frac{n}{2})$
8: $\frac{n}{2}$ **processor array** $\frac{n}{2} \dots n-1$ **does**
9: $S\left[\frac{n}{2} \dots n - 1\right] \leftarrow \textsc{UpperLower}^{\diamondsuit}_{\textsc{crew}}(X\left[\frac{n}{2} \dots n - 1\right], \frac{n}{2})$
10: **for** $i \leftarrow \frac{n}{2}$ **to** $n-1$ **do in parallel**
11: **processor** i **does**
12: $S[i] \leftarrow S[i] + S\left[\frac{n}{2} - 1\right]$ ▷ Concurrent read requirement
13: **return** S

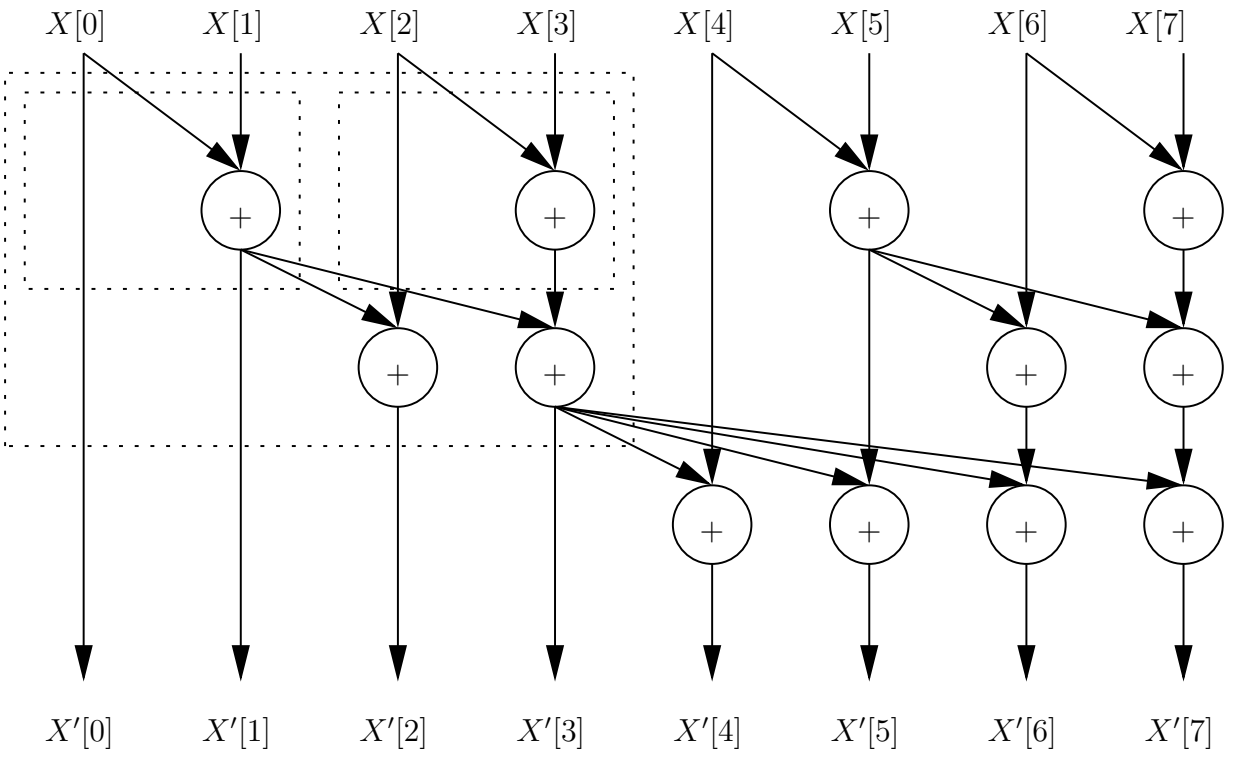
The size of the upper/lower algorithm is greater than $n-1$ because if the two halves use the minimum of roughly $n/2-1$ operations then the total number of operations is at least $n-2+\left\lfloor \frac{n}{2} \right\rfloor$.

Let $P^{ul}(n)$ denote the upper/lower prefix algorithm on n inputs.

Theorem 0.1 *Let $S(n)$ be the size of $P^{ul}(n)$, then for $n = 2^t$, $S(2^t) = t2^{t-1} = \frac{n}{2}\log_2 n$.*

Proof For the base case when $t = 1$, $S(2) = 1 \times 2^{1-1} = 1$ which is true. Assume that $S\left(2^i\right) = i2^{i-1}$


$X[0]$
$X[\frac{n}{2}-1]$
$X[\frac{n}{2}]$
$X[n-1]$
$\frac{n}{2}$
$\frac{n}{2}$
+
+
+
$X'[0]$
$X'[\frac{n}{2}-1]$
$X'[\frac{n}{2}]$
$X'[n-1]$


$X[0]$
$X[1]$
$X[2]$
$X[3]$
$X[4]$
$X[5]$
$X[6]$
$X[7]$
$X'[0]$
$X'[1]$
$X'[2]$
$X'[3]$
$X'[4]$
$X'[5]$
$X'[6]$
$X'[7]$

for $i > 1$. Now for $n = 2^{i+1}$,

$$\begin{aligned} S\left(2^{i+1}\right) &= S\left(2^i\right) + S\left(2^i\right) + 2^i \\ &= 2\,S\left(2^i\right) + 2^i \\ &= i2^i + 2^i \\ &= (i+1)2^i. \end{aligned}$$

Substituting $i' = i + 1$ yields $S\left(2^{i'}\right) = i'2^{i'-1}$ which by inductive hypothesis is true and this completes the proof. ■

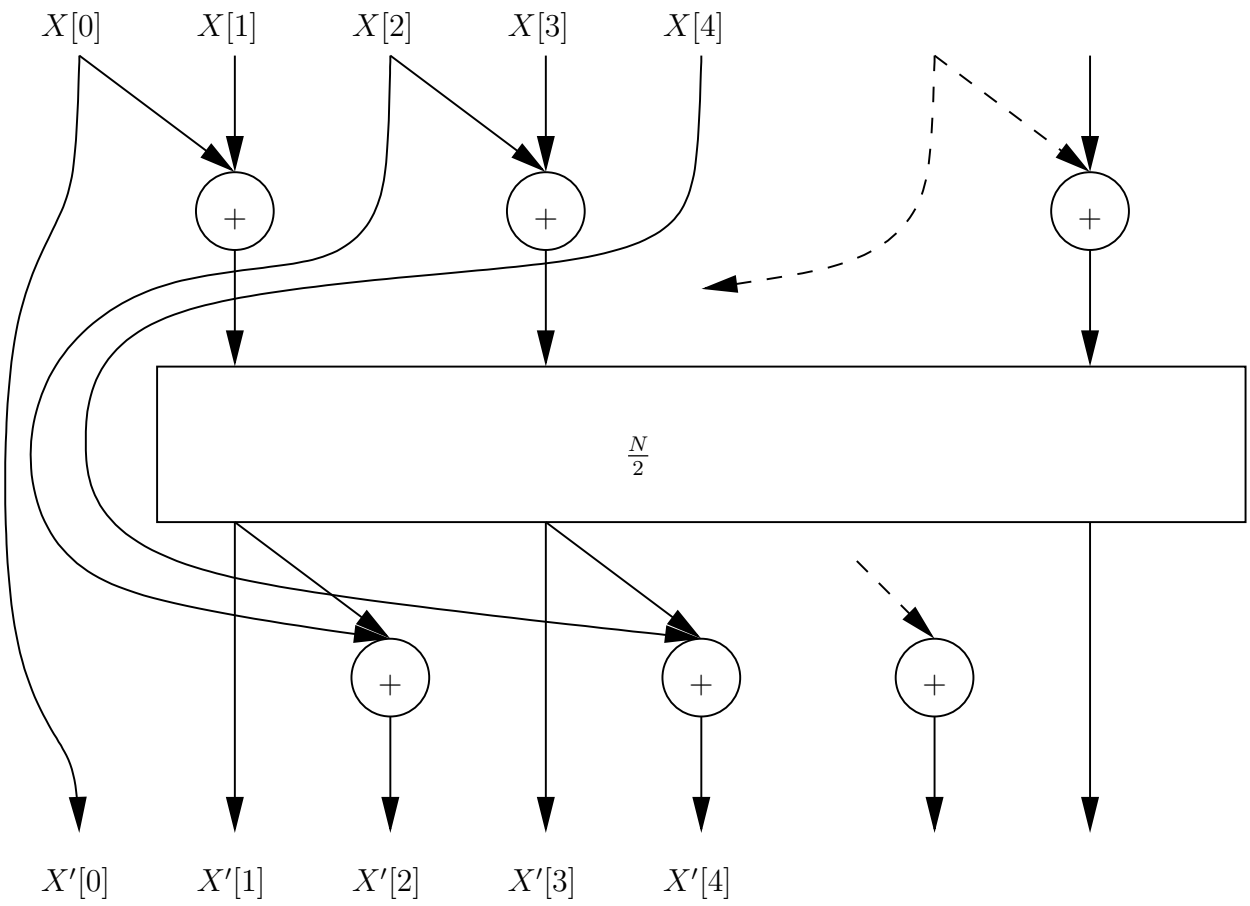
- If n is not a power of two then one prove bounds (using padding), e.g. that $(t-1)2^{t-2} \leq S(n) \leq t2^{t-1}$ for $2^{t-1} < n < 2^t$ and $t > 0$.
- Similarly the depth of $P^{ul}(n)$ for $n = 2^t$ is shown to be $t = \log_2 n$.
- If as many as $n/2$ processors are available then the operation can be completed in $\log_2 n$ steps. Furthermore, using $p = \frac{n}{2}$ processors:
  - speedup is $\frac{n-1}{\log_2 n}$.
  - efficiency is $\frac{2(n-1)}{n \log_2 n} \times 100\%$
  - work is $\frac{n}{2} \log_2 n$ compared to $(n-1)$ for sequential algorithm, which is not optimal, but efficient.

Odd/Even parallel prefix algorithm

- The odd/even parallel prefix algorithm, P^{oe}, also uses a divide and conquer approach.
- The algorithm divides n inputs into groups whose indices are odd and even, respectively.
- The recursion continues to halve the number of inputs until reaching a base case.

Description: For array $X = [x_0 \ldots x_{n-1}]$ returns an array $S = [s_0 \ldots s_{n-1}]$ of sums where $s_i = \sum_{j=0}^{i} x_j$.
Analysis: $\Theta(\log n)$
Processors: n

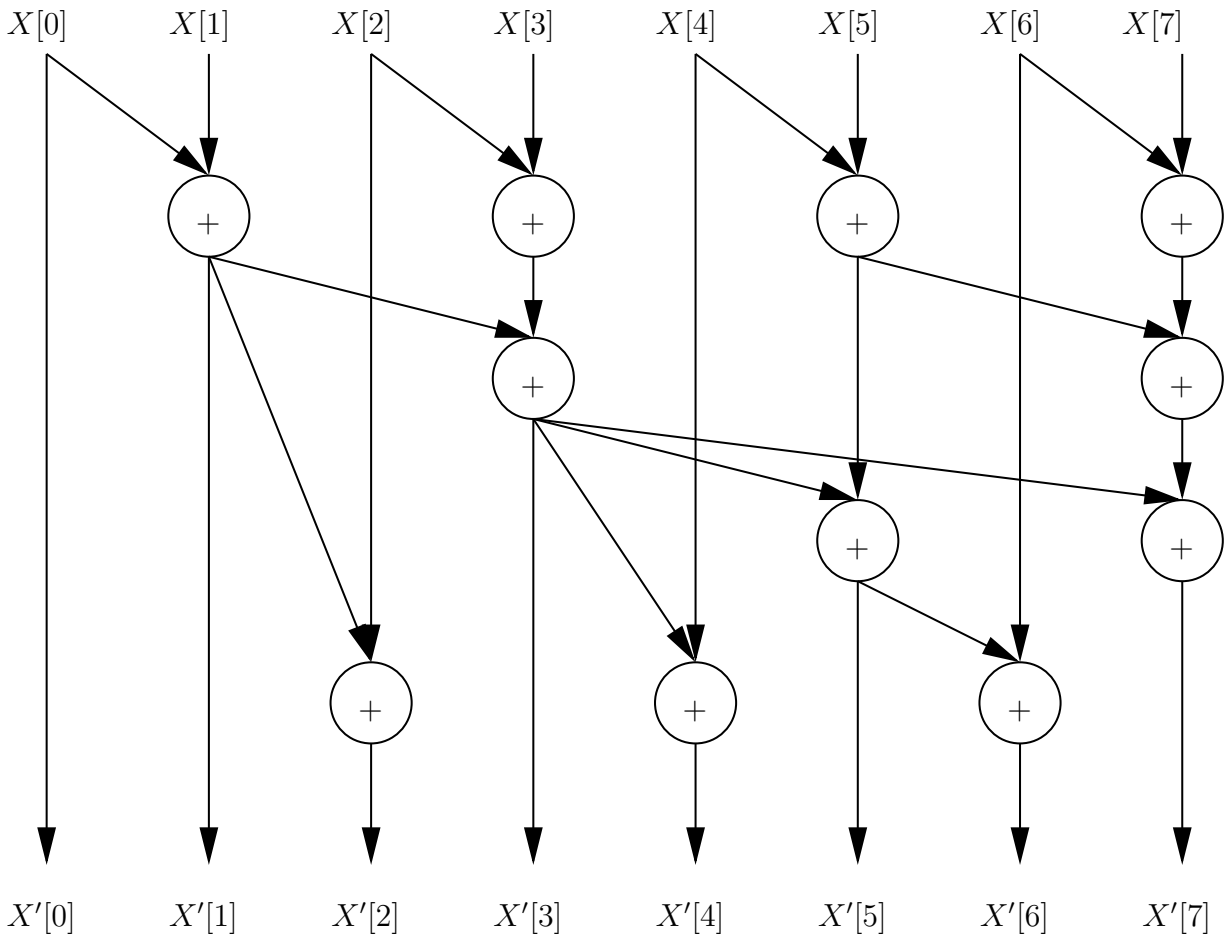
```
1: procedure OddEven^◇_EREW(X, n)
2:     if n ≤ 4 then
3:         S[0] ← X[0]
4:         for i ← 1 to n do
5:             S[i] = X[i] + S[i − 1]
6:         return S
7:     for i ← 0 to n − 1 do in parallel
8:         processor i | i mod 2 = 0 does
9:             S[i] ← X[i]
```



10: **processor** $i \mid i \bmod 2 = 1$ **does**
11: $P\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \leftarrow X[i-1] + X[i]$
12: $\frac{n}{2}$ **processor array** $i \mid i \bmod 2 = 1, 0 < i \leq n-1$ **does**
13: $A \leftarrow \textsc{OddEven}^{\diamondsuit}_{\text{EREW}}(P, \frac{n}{2})$
14: **for** $i \leftarrow 2$ **to** $n-1$ **do in parallel**
15: **processor** $i \mid i \bmod 2 = 0$ **does**
16: $S[i] \leftarrow S[i] + A\left[\frac{i}{2} - 1\right]$
17: **return** S

- Clearly the construction of $P^{oe}(n)$ adds 2 levels of operations to the depth for each time that the number of inputs are halved.
- When $n = 4$, the construction of $P^{oe}(4)$ finishes with 2 levels of operations.
- Considering only when $n = 2^t$, the depth of $P^{oe}\left(2^t\right)$ is

$$\sum_{i=3}^{t} 2 + 2 = 2t - 2 = 2\log_2 n - 2, t \geq 2.$$


$X[0]$
$X[1]$
$X[2]$
$X[3]$
$X[4]$
$X[5]$
$X[6]$
$X[7]$
+
$X'[0]$
$X'[1]$
$X'[2]$
$X'[3]$
$X'[4]$
$X'[5]$
$X'[6]$
$X'[7]$

- Similarly for size, the construction of $P^{oe}(2^t)$ is

$$\sum_{i=1}^{t}(2^i - 1) = 2^{t+1} - t - 2$$
$$= 2n - \log_2 n - 2, t \geq 0.$$

  This is significantly better than size of $P_{ul}(n)$, but still the work on $p = \Theta(n)$ processors is $\Theta(n \log n)$ which is not optimal.

Optimal parallel prefix sum algorithm

To be done in class.

Ladner and Fischer's parallel prefix algorithm

- A more sophisticated parallel prefix algorithm that allows finer tradeoffs between depth and size was proposed by Ladner and Fischer in 1980.
- Their work actually defines a class of algorithms called $P_j(n)$ for $j \geq 0$.
- Here we examine the fundamental construction that uses $P_0(n)$ and $P_1(n)$ only. Both the upper/lower and even/odd constructions are used.
- Basically, the odd/even construction is used to construct $P_1(n)$ from $P_0(n/2)$, and the upper/lower construction is used to define $P_0(n)$ in terms of both $P_1(n/2)$ and $P_0(n/2)$.

The example construction for 8 elements is equivalent to the upper/lower construction for 8 elements because odd/even on 4 elements is equivalent to upper/lower on 4 elements. A large construction is needed to see the difference.

The construction for $P_0(n)$, $n = 2^t$, maintains a depth of $\lceil \log_2 n \rceil$. This is due to the odd/even construction of $P_1(n)$ having one less operation than its depth in the right most element. The construction for $P_0(n)$ is using the upper/lower construction which maintains a minimum depth.

The size for $P_0(n)$ is approximately $4n - 4.96n^{0.69} + 1$. The proof is based on the Fibonacci sequence :-)

The size is by no means as easy to show. Could you guess what it is?

Consider the basic size relationships:

$$S_0(n) = S_1(\lceil n/2 \rceil) + S_0(\lfloor n/2 \rfloor) + \lceil n/2 \rceil$$
$$S_j(n) = S_{j-1}(\lceil n/2 \rceil) + n - 1, j \geq 1, \text{even } n \geq 2$$
$$S_j(n) = S_{j-1}(\lceil n/2 \rceil) + n - 2, j \geq 1, \text{odd } n \geq 3$$

Theorem 0.2 *If $n = 2^t$ then*

$$S_0(n) = 4n - F(2+t) - 2F(3+t) + 1$$
$$S_1(n) = 3n - F(1+t) - 2F(2+t)$$

where $F(m)$ is the m-th Fibonacci number.

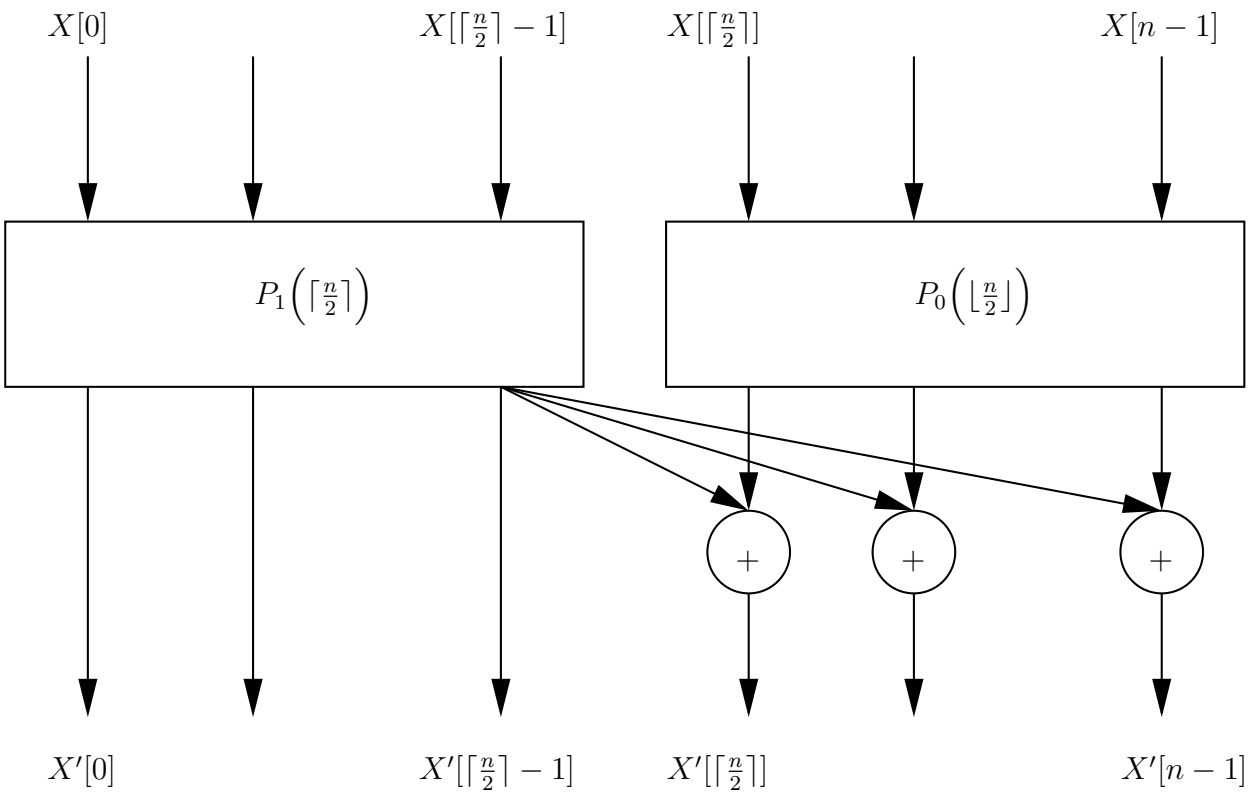


Figure 32: Definition of Ladner and Fischer's $P_0(n)$.

Note: $F(0) = 0$, $F(1) = 1$ and $F(m) = F(m-1) + F(m-2)$ for $m \geq 2$.

Proof For the base cases, $S_0(2^1) = 1 = 4{\times}2{-}F(3){-}2F(4){+}1$ and $S_1(2^2) = 4 = 3{\times}4{-}F(3){-}2F(4)$. Assume that for $t = k$

$$\begin{aligned} S_0(n) &= 4n - F(2+k) - 2F(3+k) + 1 \\ S_1(n) &= 3n - F(1+k) - 2F(2+k). \end{aligned}$$

Now show the result for $k+1$ using the size relationships deduced earlier:

$$\begin{aligned} S_0(2^{k+1}) &= 3 \times 2^k - F(1+k) - 2F(2+k) + \\ &\quad 4 \times 2^k - F(2+k) - 2F(3+k) + 1 + 2^k \\ &= 4 \times 2^{k+1} - F(3+k) - 2F(4+k) + 1, \end{aligned}$$

$$\begin{aligned} S_1(2^{k+1}) &= 4 \times 2^k - F(2+k) - 2F(3+k) + 1 + 2^{k+1} - 1 \\ &= 3 \times 2^{k+1} - F(2+k) - 2F(3+k). \end{aligned}$$

Substituting $k' = k+1$ yields the same form as our assumptions and the proof is complete. ■

A well known asymptotic formula for the Fibonacci numbers is:

$$F(m) = \frac{\phi^m - \hat{\phi}^m}{\sqrt{5}}$$

where

$$\phi = \frac{1+\sqrt{5}}{2}$$

and

$$\hat{\phi} = \frac{1-\sqrt{5}}{2}.$$

For large m, $F(m) \approx \frac{\phi^m}{\sqrt{5}}$.

Consequently:

$$\begin{aligned} S_0(n) &\approx 4n - \frac{\phi^{k+2}}{\sqrt{5}} - 2\frac{\phi^{k+3}}{\sqrt{5}} + 1 \\ &= 4n - 4.9597n^{0.69424} + 1 \end{aligned}$$

Bounds on dyadic operations

It is easy to show that no matter how many processors we have, the depth of an arithmetic expression with n atoms is at least $\lceil \log_2 n \rceil$ (consider a tree with the root as the final answer and each node in the tree combining at most 2 atoms), and bounds the computation time from below.

Harder to prove, is that any expression with n atoms can be rearranged using associativity, commutativity, and distributivity so that the time required to evaluate the expression is no more than $\lceil 4 \log_2 n \rceil$ with the use of no more than $3n$ processors.

Such a rearrangement though takes $O(n \log_2 n)$ steps, so the computational overhead to rearrange the expression may be larger than the benefit gained.

Pointer jumping

- Sometimes we have data structures that are represented by pointer based data structures, where e.g. P is a list of nodes and $P[i] \in P$ is a pointer to a node in P.
- *Pointer jumping* is a well known technique to process such lists in parallel.
- Consider the following problems:
  - List Ranking: List P is a list of nodes where $P[i]$ points next node in the list for node i. Return the array R where $R[i]$ is the distance from node i to the head of the list.
  - Forest P is an array of nodes where $P[i]$ points to the parent of node i. Return the array S where $S[i]$ points to the root of node i.
  - Prefix Sum: List P is a list of nodes where $P[i] = (n, v)$ is a $(Next, Value)$ tuple, $Next$ pointing to the next node in the list for node i and $Value$ is a number being held by node i. Return the array S where $S[i]$ is the prefix sum from the head of the list.

Description: List P is a list of nodes where $P[i]$ points next node in the list for node i. Returns array R where $R[i]$ is the distance from node i to the head of the list.
Analysis: $\mathcal{O}(\log n)$ steps.
Processors: n

```
1: procedure ListRank^◇_EREW(P, n)
2:     for i ← 0 to n − 1 do in parallel
3:         processor i does
4:             if P[i] = ∅ then
5:                 R[i] ← 0
6:             else
7:                 R[i] ← 1
8:             S[i] ← P[i]
9:             while S[i] ≠ ∅ do
10:                R[i] ← R[i] + R[S[i]]
11:                S[i] ← S[S[i]]
12:    return R
```

Description: Forest P is an array of nodes where $P[i]$ points to the parent of node i. Returns array S where $S[i]$ points to the root of node i.
Analysis: $\mathcal{O}(\log n)$ steps.
Processors: n

```
1: procedure FindRoots^◇_CREW(P, n)
2:     for i ← 0 to n − 1 do in parallel
3:         processor i does
4:             S[i] ← P[i]
5:             while S[i] ≠ S[S[i]] do
6:                 S[i] ← S[S[i]]
7:     return S
```

Description: List P is a list of nodes where $P[i] = (n, v)$ is a $(Next, Value)$ tuple, $Next$ pointing to the next node in the list for node i and $Value$ is a number being held by node i. Returns array S where $S[i]$ is the prefix sum from the head of the list.
Analysis: $\mathcal{O}(\log n)$ steps.
Processors: n

```
1: procedure ListPrefixSum^◇_EREW(P, n)
2:     for i ← 0 to n − 1 do in parallel
3:         processor i does
4:             R[i] ← P[i]_Value
5:             S[i] ← P[i]_Next
6:             while S[i] ≠ ∅ do
7:                 R[i] ← R[i] + R[S[i]]
8:                 S[i] ← S[S[i]]
9:     return R
```

Description: List P is a list of nodes where $P[i] = (n, v)$ is a $(Next, Value)$ tuple, $Next$ pointing to the next node in the list for node i and $Value$ is a number being held by node i. Returns array S where $S[i]$ is the prefix sum from the head of the list.
Analysis: $\mathcal{O}\left(\frac{n}{p} + \log p\right)$ steps.

```
1: procedure ListPrefixSum^★_EREW(P, n, p)
2:     for i ← 0 to p − 1 do in parallel
3:         processor i does
4:             S[i] ← P[i]_Value
5:             c ← i                                  ▷ The current node for processor i
6:             R[i] ← ∅                               ▷ Prepare array of pointers in reverse direction
7:             for j ← 0 to log n − 1 do
8:                 p ← c
9:                 c ← P[c]_Next
10:                if c ≠ ∅ then
11:                    R[c] ← p                       ▷ Build pointers in the reverse direction
12:            e ← c                                  ▷ Remember the start of the next list
13:            c ← p
14:            for j ← 0 to log n − 2 do
15:                c ← R[c]
16:                S[c] ← S[c] + S[P[c]_Next]
17:            T[p − i − 1] ← (e, S[i])               ▷ Prepare the shorter list of length n/log n
18:    all processors do
19:        X ← ListPrefixSum^◇_EREW(T, p)
20:    for i ← 0 to p − 1 do in parallel
21:        processor i does
22:            c ← i
23:            for j ← 0 to log n − 1 do
24:                S[c] ← S[c] + T[p − i − 1]
```

25: $c \leftarrow P[c]_{Next}$

26: **return** S

Sorting, Merging and other building blocks

- *Compaction* For array $X = [x_0 \dots x_{n-1}]$ such that k elements of X have nonempty values, move all nonempty values into the first k consecutive locations of X, e.g. $[\emptyset, 1, \emptyset, \emptyset, 4]$ becomes $[1, 4, \emptyset, \emptyset, \emptyset]$.
- *Unique Counts* Given a sorted array X of n elements, return an array C of length n that contains tuples of the form $(Value, Count)$ for each distinct element (value) in X, consecutively in the lower portion of C and $\emptyset$ for all remaining unused elements of C.
- *Distribution* Let X contain elements such that some are empty and some have values, e.g. $[6, 3, \emptyset, \emptyset, \emptyset, 5, \emptyset, \emptyset]$, then the returned array has the value of all nonempty elements copied to itself and all consecutive nonempty positions: $[6, 3, 3, 3, 3, 5, 5, 5]$.

Description: For array $X = [x_0 \dots x_{n-1}]$ such that k elements of X have nonempty values, moves all nonempty values into the first k consecutive locations of X, e.g. $[\emptyset, 1, \emptyset, \emptyset, 4]$ becomes $[1, 4, \emptyset, \emptyset, \emptyset]$.
Analysis: $\Theta\left(\frac{n}{p} + \log p\right)$

```
1:  procedure CompactionEREW★(X, n, p)
2:      for i ← 0 to p − 1 do in parallel
3:          processor i does
4:              for j ← i n/p to (i + 1) n/p − 1 do                ▷ Θ(n/p) steps
5:                  if X[j] ≠ ∅ then
6:                      R[j] ← 1
7:                  else
8:                      R[j] ← 0
9:                  A[j] ← ∅                          ▷ Setup a temporary array as well
10:     all processors do
11:         R ← PrefixSumEREW★(R, n, p)                          ▷ Θ(n/p + log p) steps
12:     for i ← 0 to p − 1 do in parallel
13:         processor i does
14:             for j ← i n/p to (i + 1) n/p − 1 do              ▷ Θ(n/p) steps
15:                 if X[j] ≠ ∅ then
16:                     A[R[j] − 1] ← X[j]
17:     for i ← 0 to p − 1 do in parallel
18:         processor i does
19:             for j ← i n/p to (i + 1) n/p − 1 do              ▷ Θ(n/p) steps
20:                 X[j] ← A[j]
21:     return X
```

Description: Given a sorted array X of n elements, return an array C of length n that contains tuples of the form $(Value, Count)$ for each distinct element (value) in X, consecutively in the lower portion of C and $\emptyset$ for all remaining unused elements of C.
Require: $n > 0$
Analysis: $\Theta\left(\frac{n}{p} + \log p\right)$

```
1: procedure UniqueCounts★EREW(X, n, p)
2:     for i ← 0 to p − 1 do in parallel
3:         processor i does
4:             for j ← i n/p to (i + 1) n/p − 1 do                ▷ Θ(n/p) steps
5:                 if j = 0 then
6:                     R[0] ← 1
7:                 else if X[j] ≠ X[j − 1] then
8:                     R[j] ← 1
9:     all processors do
10:        R ← PrefixSum★EREW(R, n, p)                            ▷ Θ(n/p + log p) steps
11:    for i ← 0 to p − 1 do in parallel
12:        processor i does
13:            for j ← i n/p to (i + 1) n/p − 1 do                ▷ Θ(n/p) steps
14:                C[j] = ∅
15:    for i ← 0 to p − 1 do in parallel
16:        processor i does
17:            for j ← i n/p to (i + 1) n/p − 1 do                ▷ Θ(n/p) steps
18:                if j = 0 then
19:                    C[0] ← (X[0], 0)                           ▷ (Value, Rank)
20:                else
21:                    if X[j] ≠ X[j − 1] then
22:                        C[R[j] − 1] ← (X[j], j)
23:    for i ← 0 to p − 1 do in parallel
24:        processor i does
25:            for j ← i n/p to (i + 1) n/p − 1 do                ▷ Θ(n/p) steps
26:                if j < n − 1 then
27:                    if C[j + 1] ≠ ∅ then
28:                        C[j] ← (C[j]_Value, C[j + 1]_Rank − C[j]_Rank)   ▷ (Value, Count)
29:                    else
30:                        C[j] ← (C[j]_Value, C[j]_Rank + n − j)
31:                else
32:                    C[n − 1] ← (C[n − 1]_Value, 1)   ▷ A single unique value at the very end of the array
33:    return C
```

Description: Let X contain elements such that some are empty and some have values, e.g. $[6, 3, \emptyset, \emptyset, \emptyset, 5, \emptyset, \emptyset]$, then the returned array has the value of all nonempty elements copied to itself and all consecutive nonempty positions: $[6, 3, 3, 3, 3, 5, 5, 5]$.

Ensure: $X[0] \neq \emptyset$

Analysis: $\Theta\left(\frac{n}{p} + \log p\right)$

```
1: procedure Distribute★EREW(X, n, p)
2:     for i ← 0 to p − 1 do in parallel
```

```
3:        processor i does
4:            for j ← i n/p to (i+1) n/p − 1 do                    ▷ Θ(n/p) steps
5:                if X[j] ≠ ∅ then
6:                    M[j] ← 1
7:                else
8:                    M[j] ← 0
9:    all processors do
10:       M ← PrefixSum*_EREW(M, n, p)                             ▷ Θ(n/p + log p) steps
11:       CM ← UniqueCounts*_EREW(M, n, p)                         ▷ Θ(n/p + log p) steps
12:   for i ← 0 to p − 1 do in parallel      ▷ At most p values remain to be (completely) distributed after this step, and no subarray has more than one unfinished value to distribute.
13:       processor i does
14:           x ← ∅
15:           for j ← i n/p to (i+1) n/p − 1 do                    ▷ Θ(n/p) steps
16:               if X[j] ≠ ∅ then
17:                   x ← X[j]
18:               else if x ≠ ∅ then
19:                   X[j] ← x
20:   for i ← 1 to p − 1 do in parallel      ▷ Processor 0 is never active since X[0] ≠ ∅
21:       processor i does
22:           if X[i n/p] = ∅ then
23:               if X[i n/p − 1] ≠ ∅ then
24:                   x ← CM[M[i n/p] − 1]_Value
25:               k = p/n CM[M[i n/p] − 1]_Count − 1 processor array i ... i + k − 1 does
26:                   Broadcast*_EREW(x, k)                        ▷ Θ(log k) steps, k ≤ p
27:               for j ← i n/p to (i+1) n/p − 1 do                ▷ Θ(n/p) steps
28:                   if X[j] = ∅ then
29:                       X[j] ← x
30:                   else
31:                       break
32:   return X
```

Sorting

- Sorting algorithms are fundamental to many applications. Given an array of n data elements, $\{a_0, a_1, \ldots, a_{n-1}\}$, sorting rearranges the order of the elements to produce a sorted array, $\{b_0, b_1, \ldots, b_{n-1}\}$ such that $b_i \leq b_j$ for every $0 \leq i \leq j \leq n-1$.
- The worst-case time complexity of mergesort is $O(n \log_2 n)$.
- The average-case time complexity of quicksort is $O(n \log_2 n)$.
- Using n processors, at best we could expect a time complexity of $O(\log_2 n)$.

Mergesort without parallel merging

Mergesort proceeds from a single processor or process that holds an array of $n = 2^t$ elements. The divide-and-conquer approach is used to divide the array into two halves and give one half to another process. The subdivision continues until at most each of n processes holds exactly one element. Then the processes use Mergesort to generate the sorted array.

Assuming there are $p = n = 2^t$ processors. The first division phase of the Mergesort algorithm is essentially scattering the elements over the processors. Each processor receives one element of the array.

The total number of parallel computation steps is t, at each step, $i = 0, 1, \ldots, t-1$, two lists of size 2^i are merged with a single processor.

It takes $2n - 1$ steps in the worst case to merge two sorted lists each of n numbers.

The number of computational steps is then

$$2\sum_{i=0}^{t-1}\left(2^i - \tfrac{1}{2}\right) = 2^t - t - 2$$

which is $O(n)$.

Quicksort

Quicksort also parallelizes over n processors to obtain $O(n)$ parallel computational steps.

Recall that Quicksort selects a *pivot* for the elements in the array. All elements in the array that are less than the pivot are put into a lower array and all elements greater than the pivot are put into a higher array. The Quicksort algorithm is then recursively applied on the higher and lower arrays.

Selection of the pivot is not too important for the sequential algorithm however it is important for the parallel algorithm in order to keep the tree of processes reasonably balanced.

Mergsort with parallel merge

- Merge two sorted lists, $A = [a_0 \ldots a_{n_1-1}]$ and $B = [b_0 \ldots b_{n_2-1}]$, of length n_1 and n_2 resp., where $a_i, b_j \in \{1, 2, \ldots, n\}$ for all $0 \le i < n_1$, $0 \le j < n_2$.
- Sort list $A = [a_0 \ldots a_{n-1}]$ of length n where $a_i \in \{1 \ldots n\}$ for all $0 \le i < n$.

Description: Merge two sorted lists, $A = [a_0 \ldots a_{n_1-1}]$ and $B = [b_0 \ldots b_{n_2-1}]$, of length n_1 and n_2 resp., where $a_i, b_j \in \{1, 2, \ldots, n\}$ for all $0 \le i < n_1$, $0 \le j < n_2$. Based on Bahig and Bahig, 2007.

Require: $n_1 > 0, n_2 > 0$, $n = \max\{n_1, n_2\}$

Analysis: $\Theta\left(\frac{n}{p} + \log p\right)$

1: **procedure** $\text{MERGE}^{\bigstar}_{\text{EREW}}(A, B, n_1, n_2, p)$
2: **for** $i \leftarrow 0$ **to** $p - 1$ **do in parallel**
3: **processor** i **does**
4: **for** $j \leftarrow i\frac{n}{p}$ **to** $(i+1)\frac{n}{p} - 1$ **do** ▷ $\Theta\left(\frac{n}{p}\right)$ steps
5: $X[j] \leftarrow \emptyset$

6: **all processors do**
7: $CA \leftarrow \textsc{UniqueCounts}^{\bigstar}_{\text{EREW}}(A, n_1, p)$ ▷ $\Theta\left(\frac{n_1}{p} + \log p\right)$ steps,$n_1 \leq n$
8: $CB \leftarrow \textsc{UniqueCounts}^{\bigstar}_{\text{EREW}}(B, n_2, p)$ ▷ $\Theta\left(\frac{n_2}{p} + \log p\right)$ steps,$n_2 \leq n$
9: **for** $i \leftarrow 0$ **to** n_1 **do in parallel**
10: **processor** i **does**
11: **for** $j \leftarrow i\frac{n}{p}$ **to** $(i+1)\frac{n}{p} - 1$ **do** ▷ $\Theta\left(\frac{n}{p}\right)$ steps
12: **if** $CA[j] \neq \emptyset$ **then**
13: $X[CA[j]_{Value}] \leftarrow CA[j]$
14: **for** $i \leftarrow 0$ **to** n_2 **do in parallel** ▷ Aggregate the counts
15: **processor** i **does**
16: **for** $j \leftarrow i\frac{n}{p}$ **to** $(i+1)\frac{n}{p} - 1$ **do** ▷ $\Theta\left(\frac{n}{p}\right)$ steps
17: **if** $CB[j] \neq \emptyset$ **then**
18: **if** $X[CB[j]_{Value}] = \emptyset$ **then**
19: $X[CB[j]_{Value}] \leftarrow CB[j]$
20: **else**
21: $X[CB[j]_{Value}] \leftarrow (CB[j]_{Value}, CB[j]_{Count} + X[CB[j]_{Value}]_{Count})$
22: **all processors do**
23: $X \leftarrow \textsc{Compaction}^{\bigstar}_{\text{EREW}}(X, n, p)$ ▷ $\Theta\left(\frac{n}{p} + \log p\right)$ steps
24: **for** $i \leftarrow 0$ **to** $n - 1$ **do in parallel**
25: **processor** i **does**
26: **for** $j \leftarrow i\frac{n}{p}$ **to** $(i+1)\frac{n}{p} - 1$ **do** ▷ $\Theta\left(\frac{n}{p}\right)$ steps
27: **if** $X[j] \neq \emptyset$ **then**
28: $PX[j] \leftarrow X[j]_{Count}$
29: **else**
30: $PX[j] \leftarrow 0$
31: **all processors do**
32: $PX \leftarrow \textsc{PrefixSum}^{\bigstar}_{\text{EREW}}(PX, n, p)$ ▷ $\Theta\left(\frac{n}{p} + \log p\right)$ steps
33: **for** $i \leftarrow 0$ **to** $n - 1$ **do in parallel**
34: **processor** i **does**
35: **for** $j \leftarrow i\frac{n}{p}$ **to** $(i+1)\frac{n}{p} - 1$ **do** ▷ $\Theta\left(\frac{n}{p}\right)$ steps
36: **if** $j = 0$ **then**
37: $M[0] \leftarrow X[0]_{Value}$
38: **else**
39: **if** $X[j] \neq \emptyset$ **then**
40: $M[PX[j]] \leftarrow X[j]_{Value}$
41: **all processors do**
42: $M \leftarrow \textsc{Distribute}^{\bigstar}_{\text{EREW}}(M, n, p)$ ▷ $\Theta\left(\frac{n}{p} + \log p\right)$ steps
43: **return** M

Description: Sort list $A = [a_0 \dots a_{n-1}]$ of length n where $a_i \in \{1 \dots n\}$ for all $0 \leq i < n$. Based on Bahig and Bahig, 2007.

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

```
1: procedure MergeSort^◇_EREW(A, n, p)
2:     if n = 1 then
3:         return A
4:     if p = 1 then
5:         return SequentialSort(A, n)
6:     all processors do                                   ▷ Θ((n/p) log(n/p) + ((n/p) + log p) log p) steps
7:         p/2 processor array 0 ... n/2 − 1 does
8:             L ← MergeSort^◇_EREW(A[0 ... n/2 − 1], n/2, p/2)
9:         p/2 processor array n/2 ... n − 1 does
10:            U ← MergeSort^◇_EREW(A[n/2 ... n − 1], n/2, p/2)
11:    k = min{p, n/log n} processor array 0 ... k − 1 does
12:        A ← Merge^★_EREW(L, U, n/2, n/2, k)              ▷ Θ((n/k) + log k) steps
13:    return A
```

Rank Sort

Rank sort algorithms count for each element, a_i, the number of of elements, c_i, that are smaller than a_i. Thus the sorted array elements $b_{c_i} = a_i$.

We only consider arrays of *unique* elements, however the algorithms can be modified to take into account arrays that contain non-unique elements.

When all elements are unique then c_i is also unique over all $0 \leq i < n$.

```
for(i=0;i<n;i++){

  x=0;

  for(j=0;j<n;j++)

     if(a[i]>a[j]) x++;

 b[x]=a[i];

}
```

- Comparing each number against $n-1$ other numbers requires $n-1$ computation steps. There are n elements so there are $n(n-1)$ computational steps in total.
- For n processors, each computing the index of an element in parallel, sorting can be accomplished in $O(n)$ computational steps.
- Each processor needs access to the entire array of numbers and so this is convenient for shared memory architectures. The efficiency is $\frac{\log_2 n}{n} \times 100\%$.

Consider the use of n^2 processors. Each processor $p_{i,j}$ compares a_i with a_j. (processors $p_{i,i}$ are not actually required.) Comparison requires $O(1)$ computational steps.

Using a reduction across i, processors $p_{i,j}$ can compute the index, b_i, of element i in $O(\log_2 n)$ computational steps. In a final $O(1)$ computational step, the element a_i is written to index b_i.

The sorting is accomplished in $O(\log_2 n)$ steps. However the efficiency is $\frac{1}{n} \times 100\%$.

Using a CRCW memory architecture with concurrent writes being handled as additions, the reduction operation can be accomplished in $O(1)$ steps. Thus the sorting is accomplished in $O(1)$ steps. The efficiency is now $\frac{\log_2 n}{n} \times 100\%$.

Description: Merge two lists of size n unique elements using n processors and return merged list S of size $2n$ elements.

Analysis: $\Theta(\log n)$

Processors: n

1: **procedure** RANKMERGE$^{\diamondsuit}_{\text{CREW}}(A, B, n)$
2: **for** $p \leftarrow 0$ **to** $n - 1$ **do in parallel**
3: **processor** p **does**
4: $RA[p] \leftarrow$ SEQUENTIALRANK$(A, n, A[p])$ + SEQUENTIALRANK$(B, n, A[p])$
5: $RB[p] \leftarrow$ SEQUENTIALRANK$(A, n, B[p])$ + SEQUENTIALRANK$(B, n, B[p])$
6: $S[RA[p]] \leftarrow A[p]$
7: $S[RB[p]] \leftarrow B[p]$
8: **return** S

Bitonic Mergesort

The basis of the bitonic mergesort is the *bitonic sequence*, a list having specific properties that will be utilized in the sorting algorithm.

A monotonic increasing sequence is a sequence of increasing numbers. A bitonic sequence has two sequences, one increasing and one decreasing.

Formally, a bitonic sequence is a sequence of numbers, $a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}$, which monotonically increases in values, reaches a maximum, and then monotonically decreases in value:

$$a_0 < a_1 < a_2 < \cdots < a_i > a_{i+1} > \cdots > a_{n-2} > a_{n-1}.$$

for some $0 \leq i < n$. A sequence is also bitonic if the preceding can be achieved by shifting the number cyclically (left or right).

The bitonic sequence has an interesting property that if we compare and exchange a_i with $a_{i+n/2}$ for all $0 \leq i < n/2$, we get two bitonic sequences, where the numbers in one sequence are all less than the numbers in the other sequence. For example before:

$$3, 7, 9, 8, 6, 5, 4, 1$$

and after

$$3, 5, 4, 1, 6, 7, 9, 8.$$

The second list is now two bitonic sequences, $3, 5, 4, 1$ and $6, 7, 9, 8$. Using this property, with $n = 2^t$ elements and n processors, after t parallel steps it is clear that a given bitonic list can be sorted. This is called a *bitonic sort operation*.

So sorting an unsorted list of numbers requires building bitonic lists and then sorting the bitonic lists.

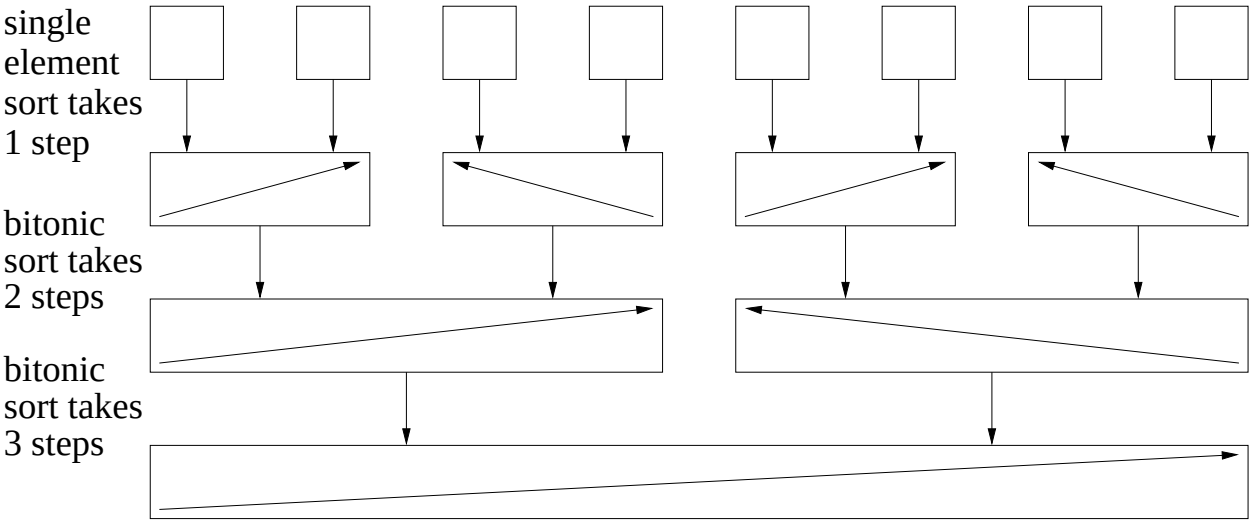


Figure 33: Bitonic Mergesort on 8 elements.

With $n = 2^t$ elements, there are t phases numbered $1, 2, \cdots, t$, each requiring a bitonic sorting operation (the first phase is simply sorting single elements) of t steps. Hence the total number of steps is given by

$$\sum_{i=1}^{t} i = \frac{t(t+1)}{2} = \frac{\log_2 n(\log_2 n + 1)}{2} = O(\log_2^2 n).$$

The speedup on n processors is thus $O\left(\frac{n}{\log_2 n}\right)$ and gives an efficiency of roughly $\frac{1}{\log_2 n} \times 100\%$.

Processor Optimal Parallel Merging

- From Huang and Kleinrock, 1990.
- Assume we wish to merge two sorted lists, L_1 and L_2, each of length N elements, or $2\,N$ elements in total. For this discussion we will assume that all the elements are distinct, but the approach will work in general.
- Assume we have $P = \sqrt{N}$ processors in total, or $N = P^2$.
- The optimal parallel merge algorithm will merge the lists in time $O(N/P) = O(\sqrt{N})$.

1. Divide L_1 into P sublists where the i-th sublist contains elements at locations $i, P + i, 2P + i, \ldots, N - P + i$. Also divide L_2 into P sublists similarly.
2. Have P_i merge the i-th sublist from L_1 and the i-th sublist from L_2 and put the result back to the locations originally occupied by these sublists $1 \leq i \leq P$. All P processors work simultaneously.

3. Group the resulting list after Step 2 into $2P$ groups with P consecutive elements in each group. Number these groups from 1 to $2P$. Have P_i merge groups $2i-1$ and $2i$ and put the result back to the locations originally by these two groups for $1 \leq i \leq P$. All P processors work simultaneously.

4. Group the resulting list after Step 3 into $2P$ groups with P elements in each group. Number these groups from 1 to $2P$. Have P_i merge groups $2i$ and $2i+1$ and put the result back to the locations originally occcupied by these two groups for $1 \leq i \leq P-1$. All $P-1$ processors work simultaneously.

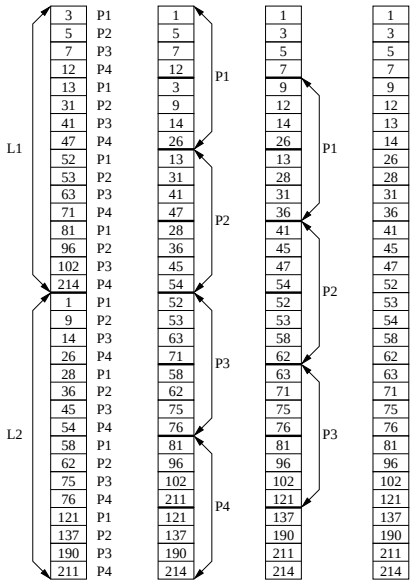


Figure 34: Example merging.

Intuitive understanding

- Lemma: After Step 2, every element is within $\pm P$ positions from its final position.
- Lemma: After Step 2, every group is sorted.
- Lemma: After Step 2, every element in group i is smaller than all elements in group j for $j \geq i+2$.
- Theorem: After Step 4, the entire list is sorted.
- Since each step 1 is constant time and steps 2 to 4 take $O(\sqrt{N})$ operations, the algorithm takes $O(\sqrt{N})$ steps in total, using $P = \sqrt{N}$ processors.

Multiway Parallel Sorting Algorithm

- Consider using $P = \sqrt{N}$ processors to sort $2\,N$ elements. Assume $P = \sqrt{N} = 2^k$.
- In Phase 1, assign $2\sqrt{N}$ elements to each processor and have each processor sort those elements independently, using the best known sequential algorithm.
- After Phase 1 we have P sorted lists.
- In Phase 2, recursively merge two sorted lists into one large sorted list until there is only one list which is totally sorted.
  - After Phase 1 there are $P = 2^k$ sorted lists; therefore we need to perform k merge runs to finish the mergesort.
  - At the beginning of the i-th step, $i = 1, 2, \ldots, k$, there are $2^k/2^{i-1}$ sorted lists each with size $N_i = 2^{k+i}$ elements.
  - The number of processors available to sort two lists at the i-th step is $P_i = 2^i$. Since $2^{k+i} \geq 2^{i+i} = (2^i)^2$, $N_i \geq P_i^2$.

Analysis of the runtime

- In the first phase each processor sorts two lists of length $\sqrt{N}$, which takes $O(\sqrt{N}\log N)$ or $O\big(\frac{N\log N}{P}\big)$.
- In the second phase if N_i is the length of the lists to merge at the i-th step and P_i is the number of processors then it takes $O(N_i/P_i) = O(2^{k+i}/2^i) = O(2^k) = O(N/P)$ operations at the i-th step.
- There are k steps in phase 2 and $k = \log P = \log\sqrt{N} = \frac{1}{2}\log N$. Therefore phase 2 takes $O(k\,N/P) = O\big(\frac{N\log N}{P}\big)$ steps in total.
- Since phase 1 and phase 2 have the same complexity, the total algorithm takes $O\big(\frac{N\log N}{P}\big)$ steps.

Huang and Kleinrock go on to show that for $P = N^{(2^k-1)/2^k}$ processors, the complexities of the merging algorithm and the sorting algorithm are $O(3^k N/P)$ and $O(3^k(N\log N)/P)$ respectively.

Vectorization

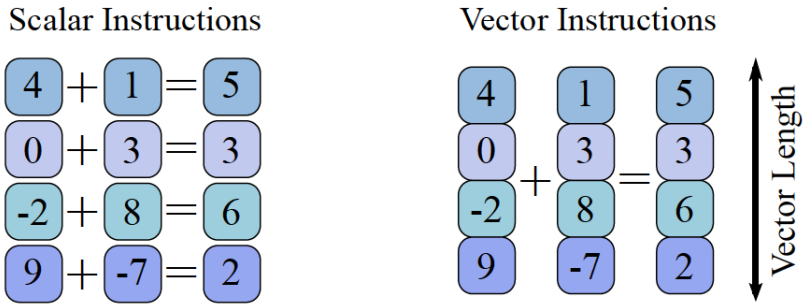
Agenda

- Vectorization
  - Automatic vectorization
  - SIMD construct
  - Data alignment

[fragile]

Vector Support

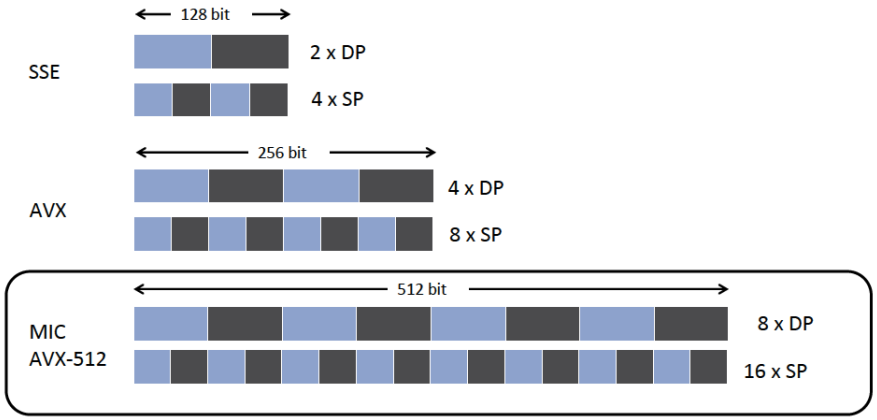
Vector instructions - one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.



[fragile]

SIMD on Intel Architecture

Width of SIMD registers has been growing:



[fragile]

Automatic Vectorization of Loops

```
#include <cstdio>

int main(){
  const int n=1024;
  int A[n] __attribute__((aligned(64)));
  int B[n] __attribute__((aligned(64)));

  for (int i = 0; i < n; i++)
  A[i] = B[i] = i;

  // This loop will be auto-vectorized
  for (int i = 0; i < n; i++)
    A[i] = A[i] + B[i];

  for (int i = 0; i < n; i++)
    printf("%2d␣%2d␣%2d\n",i,A[i],B[i]);
}
```

[fragile]

Automatic Vectorization of Loops

```
icpc autovec.cpp -qopt-report
cat autovec.optrpt
```

...
LOOP BEGIN at autovec.cpp(12,3)
remark#15300: LOOP WAS VECTORIZED
LOOP END
... [fragile]

Automatic Vectorization of Loops with conditions

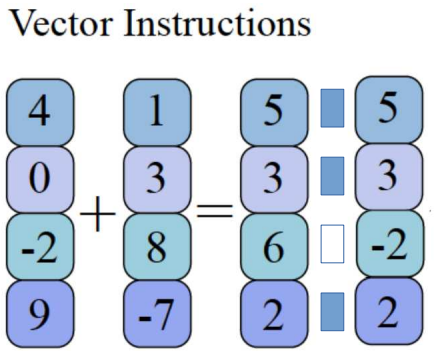
```
#include <cstdio>
int main(){
  const int n=1024;
  int A[n] __attribute__((aligned(64)));
  int B[n] __attribute__((aligned(64)));

  for (int i = 0; i < n; i++)
  A[i] = B[i] = i;

  // This loop will be auto-vectorized
  for (int i = 0; i < n; i++)
    if (B[i] % 2 == 1)
        A[i] = A[i] + B[i];
```

```
  for (int i = 0; i < n; i++)
    printf("%2d␣%2d␣%2d\n",i,A[i],B[i]);
}
```

[fragile]

Masked instructions

- Each SIMD PE computes the boolean condition → Mask
- Only perform operation on components for which mask is set
- Not all operations can use the mask
- Minimum requirement: Store
  - everyone computes result, only the chosen record it



[fragile]

Vectorizing with Unit-Stride Memory Access

Before:

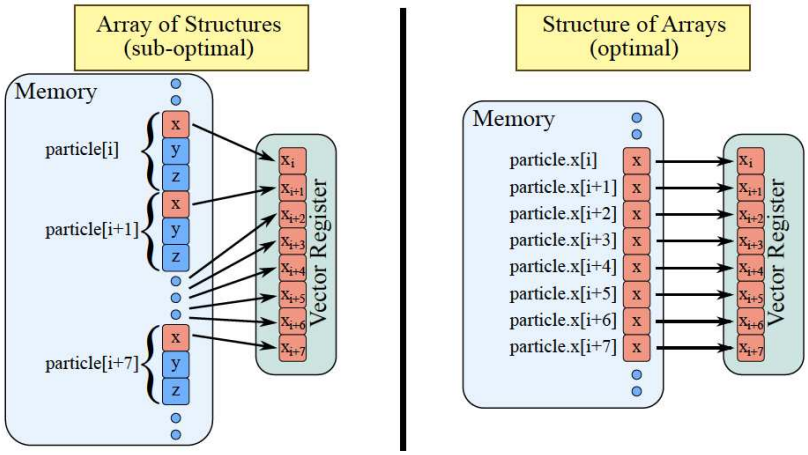
```
struct ParticleType {
 float x, y, z, vx, vy, vz;
 }; // ...
 const float dx = particle[j].x - particle[i].x;
 const float dy = particle[j].y - particle[i].y;
 const float dz = particle[j].z - particle[i].z;
```

After:

```
struct ParticleSet {
 float *x, *y, *z, *vx, *vy, *vz;
 }; // ...
 const float dx = particle.x[j] - particle.x[i];
 const float dy = particle.y[j] - particle.y[i];
 const float dz = particle.z[j] - particle.z[i];
```

[fragile]

Why AoS to SoA Conversion Helps: Unit Stride



[fragile]

Assumed Vector Dependence

True vector dependence vectorization impossible:

```
float *a, *b;
for (int i = 1; i < n; i++)
 a[i] += b[i]*a[i-1];
// dependence on the previous element
```

Assumed vector dependence compiler suspects dependence

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

[fragile]

Resolving Assumed Dependency

Restrict: Keyword indicating that there is no pointer aliasing (C++11)

```
void mycopy(int n, float* restrict a,
            float* restrict b) {
 for (int i=0; i<n; i++)
 a[i] = b[i];
}
```

`#pragma` ivdep : ignores assumed dependency for a loop (Intel Compiler)

```
void mycopy(int n, float* a, float* b) {
 #pragma ivdep
```

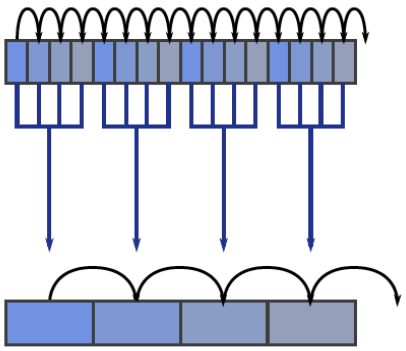
```
 for (int i=0; i<n; i++)
 a[i] = b[i];
}
```

[fragile]

Limitations on Automatic Vectorization

- Only for-looops can be auto-vectorized. Number of iterations must be known.
- Memory access in the loop must have a regular pattern, ideally with unit stride.
- Non-standard loops that cannot be automatically vectorized:
  - calculations with vector dependence
  - while-loops, for-loops with undetermined number of iterations
  - outer loops (unless `#pragma` simd overrides this restriction)
  - loops with complex branches (i.e., if-conditions)

*`#pragmaompsimd` to override



[fragile]

SIMD Loop Construct

Vectorize a loop nest

- Cut loop into chunks that fit a SIMD vector register
- No parallelization of the loop body

Syntax:

```
#pragma omp simd [clause[[,] clause], ...]
 for-loops
```

[fragile]

Simultaneous Threading and Vectorization

```
 #pragma omp parallel for
 for (int i = 0; i < n; i++)
 // Thread parallelism in outer loop
 #pragma simd
  for (int j = 0; j < m; j++)
  // Vectorization in inner loop
   DoSomeWork(A[i][j]);

#pragma omp parallel for simd
 for (int i = 0; i < n; i++)
 // If the problem is all data-parallel
  DoSomeWork(A[i]);
```

[fragile]

SIMD Function Vectorization

Declare one or more functions to be compiled for calls from a SIMD-parallel loop.

```
 #pragma omp declare simd [clause[[,] clause], ...]
```

[fragile]

SIMD Function Vectorization

Declare one or more functions to be compiled for calls from a SIMD-parallel loop.

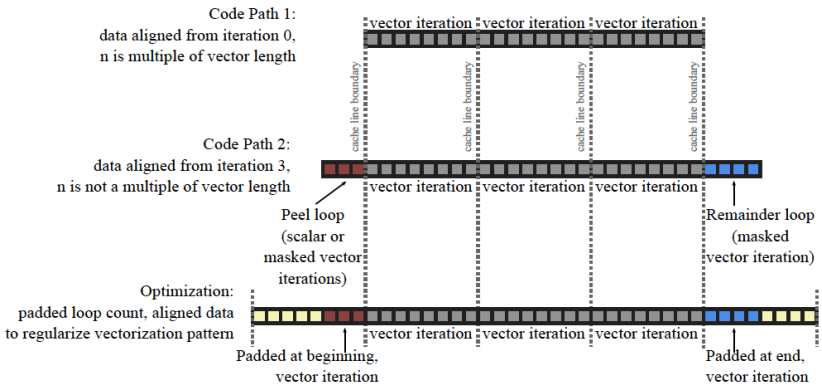
```
#pragma omp declare simd
 float min(float a, float b) {
   return a<b ? a: b;
}

#pragma omp declare simd
float distsq(float x, float y) {
   return (x-y)*(x-y);
}

void example(){
#pragma omp parallel for simd
 for (i=0;i<N; i++){
   d[i] = min(distsq(a[i],b[i]),c[i]);
 }
}
```

[fragile]

Data Alignment

Compiler may implement peel and remainder loops:

```
for (i = 0; i < n; i++) A[i] = ...
```



[fragile]

Creating Aligned Data Containers

Data alignment on the stack

```
float A[n] __attribute__((aligned(64)));
// 64-byte alignment applied
```

Data alignment on the heap

```
float *A = (float*) _mm_malloc(sizeof(float)*n, 64);
```

A[0] is aligned on a 64-byte boundary. [fragile]

Padding Multi-Dimensional Containers for Alignment

To use aligned instructions, you may need to pad inner dimension of multi-dimensional arrays to a multiple of 16 (in SP) or 8 (DP) elements.

No padding:

```
// A - matrix of size (n x n)
// n is not a multiple of 16
float* A =
 _mm_malloc(sizeof(float)*n*n, 64);

for (int i = 0; i < n; i++)
 // A[i*n + 0] may be unaligned
 for (int j = 0; j < n; j++)
  A[i*n + j] = ...
```

[fragile]

Padding Multi-Dimensional Containers for Alignment

Padding:

```
// ... Padding inner dimension
int lda=n + (16-n%16); // lda%16==0
float* A =
 _mm_malloc(sizeof(float)*n*lda, 64);

for (int i = 0; i < n; i++)
 // A[i*lda + 0] aligned for any i
 for (int j = 0; j < n; j++)
  A[i*lda + j] = ...
```

Reference

http://www.colfax-intl.com/nd/resources/slides.aspx http://www.prace-ri.eu/best-practice-gui

Agenda

- Intel MIC Architecture
- Models for Xeon Phi programming
  - Native
  - Offload

MIC: Why and what?

Key aspects of acceleration:

- An increasing number of transistors are being built into a processing chip; Moore's law is holding
- Clock rates have stalled at <4 GHz. Higher rates cause excess power consumption. (Denard's scaling no longer holds.)
- Only way to increase FLOPS per watt is through greater on-die parallelism
- This gives birth to the Many Integrated Core (MIC) architecure. In the high performance computing (HPC) market, Intel MIC is a direct competitor to the NVIDIA GPUs.

Intel Xeon Phi Coprocessors and MIC Architecture

- PCIe end-point device
- High power efficiency
- ∼1 TFLOP/s in DP
- Heterogeneous clustering

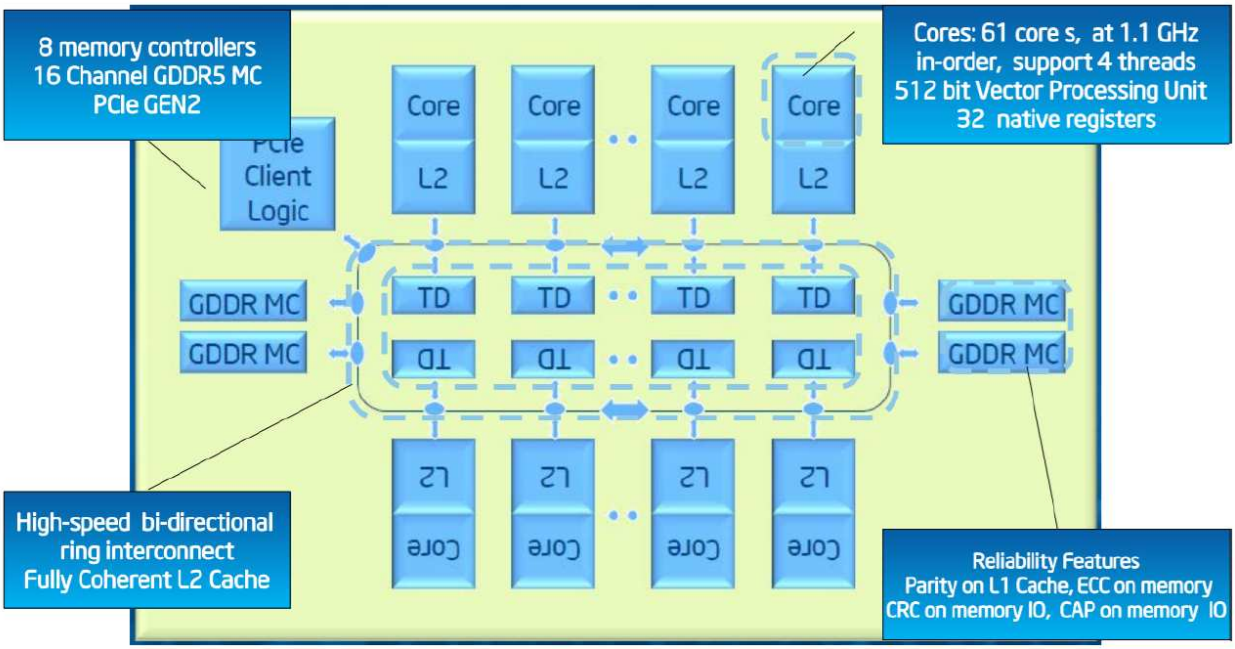
Intel MIC

- Xeon Phi = first (currently only) product of Intel's Many Integrated Core (MIC) architecture
- Usually run as a co-processor
  - PCI Express card
  - Stripped down Linux operating system, separate from host's
- Dense, simplified processor
  - Many power-hungry operations removed
  - Wider vector unit
  - Wider hardware thread count

Xeon Phi — MIC

- Based on x86 arcitecture (CPU with many cores)
  - x86 cores that are simpler, but allow for more compute throughput
- Can reuse existing x86 programming models
- Dedicate much of the silicon to floating point operations, increasing floating-point throughput
- Cache coherent
- Strip expensive features
  - No out-of-order execution (compiler interleaves computations)
  - No branch prediction (compiler unrolls loops)
- Widen SIMD registers for more throughput
- Fast (GDDR5) memory on card

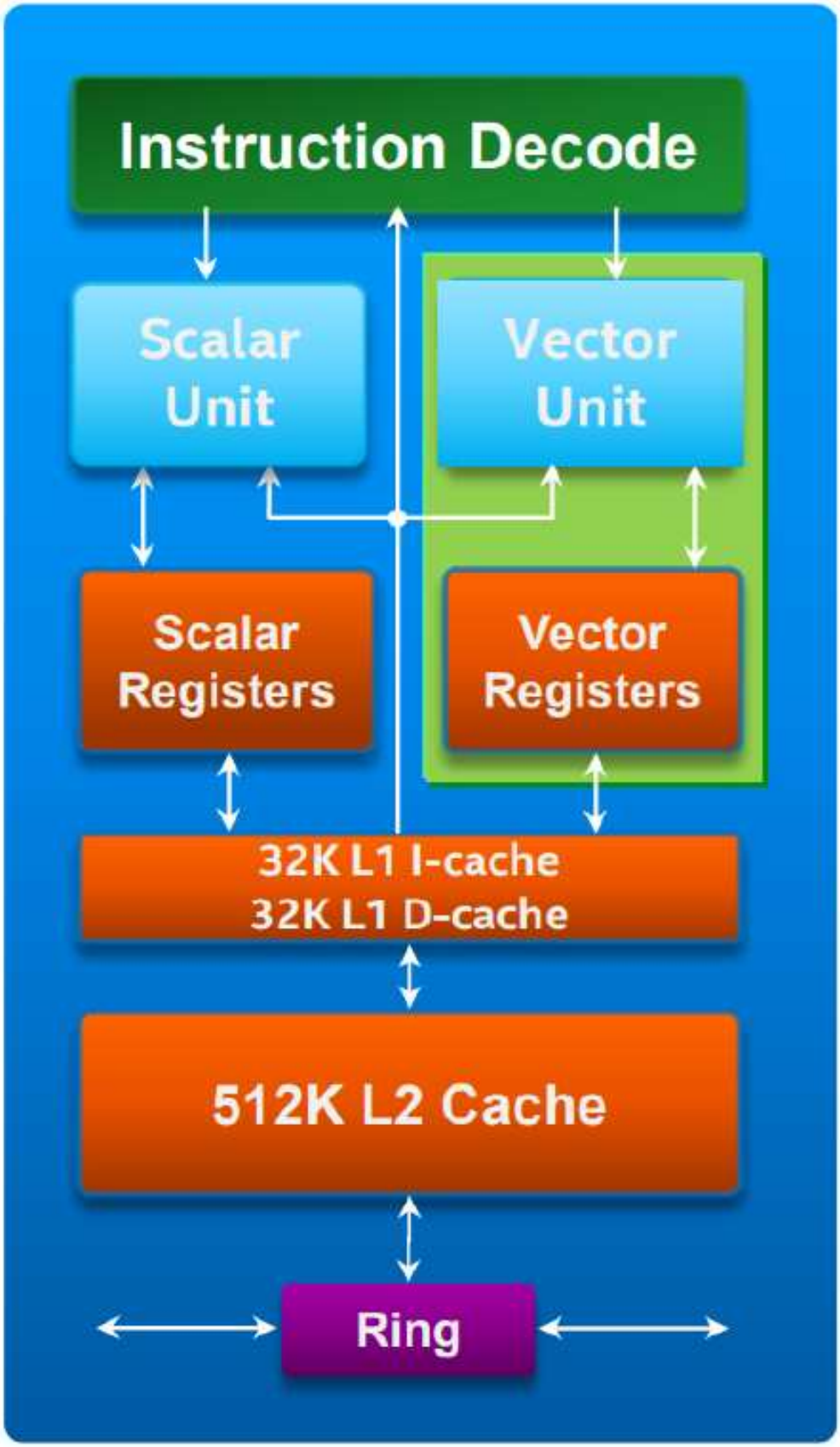
MIC Xeon Phi Ring



https://newsroom.intel.com/wp-content/uploads/sites/11/2016/01/ Intel_Xeon_Phi_Hotchips_architecture_presentation.pdf

Core architecture

- Fetches and decodes instructions from four hardware thread execution contexts
  - The execution has a 4 clock latency, hidden by round-robin scheduling of threads
- Executes the x86 ISA, and Knights Corner vector instructions
- The core can execute 2 instructions per clock cycle, one per pipe
- SIMD vector processing engine 32 kB, 8-way set associative L1 I-cache and D-cache
- Coherent 512 kB L2 cache per core


Instruction Decode
Scalar Unit
Vector Unit
Scalar Registers
Vector Registers
32K L1 I-cache
32K L1 D-cache
512K L2 Cache
Ring

https://software.intel.com/sites/default/files/ Intel\%C2\%AE_Xeon_Phi\%E2\%84\%A2_Coprocessor_Architecture_Overview.pdf

Comparison with a multicore CPU

Multi-Core Architecture

- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- $\leq$ 18 cores $\approx$ 3 GHz
- 2-way hyper-threading
- 256-bit AVX vectors

Intel Many Integrated Core (MIC) Architecture

- C/C++/Fortran; OpenMP/MPI
- Special Linux uOS distribution
- 6-16GB cached GDDR5 RAM
- 57-61 cores at $\approx$ 1GHz
- 4 hardware threads per core
- 512-bit IMCI vectors

Comparison with a CPU

- CPUs are designed for all workloads, high single-thread performance
- MIC is optimized for number crunching
  - Focus on high aggregate throughput via lots of weaker threads regularly achieves $> 2\times$ performance of dual E5 CPUs

	MIC (SE10P)	CPU (E5)	MIC is...
Number of cores	61	8	much higher
Clock speed (GHz)	1	2.7	lower
SIMD width (bits)	512	256	higher
DP GLFLOPS/core	16+	21+	lower
HO threads/core	4	1	higher

Comparison with a GPU with CUDA

- Both Xeon Phi and GPUs can be used as accelerators:
  - Accelerate applications in *offload mode*, where portions of the application are accelerated by a remote device
  - Run many threads in parallel to achieve massive parallelism
- Xeon Phi is not just an accelerator:
  - In *coprocessor native execution mode*, the Phi appears as another machine connected to the host, like another node in a cluster
  - In *symmetric execution mode*, application processes run on both the host and the coprocessor, communication through message passing
  - MIMD vs SIMD:
    * CUDA threads are grouped in warps (work groups in OpenCL) in a SIMD (single instruction, multiple data) model
    * Phi coprocessors run generic MIMD threads individually in a MIMD (multiple instruction, multiple data) model

Summary of MIC advantages

- Programming MIC is similar to programming for CPUs
  - Code compatible (C, C++, Fortran) with re-compilation
  - Any code can run on MIC, not just kernels
- Offers a new, flexible offload programming paradigm
  - C/Fortran markup to denote code to execute on Phi at runtme
  - Link to maths kernel library (MKL) implementation, which can offload automatically
- Goal: Optimizing for MIC is similar to optimizing for CPUs
  - Optimize once, run anywhere

Drawback

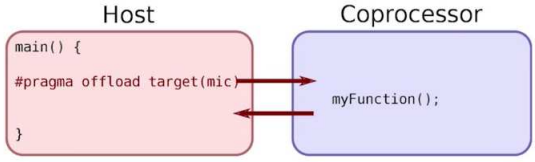
- In practice, just as much recoding as for GPUs
- The future of Xeon Phi at Intel is unclear
  - (It doesn't have the economy-of-scale from the gamer market.)

center

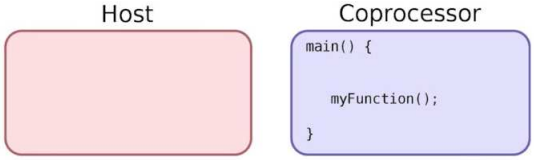
title # Models for Xeon Phi programming

Offload and Native modes

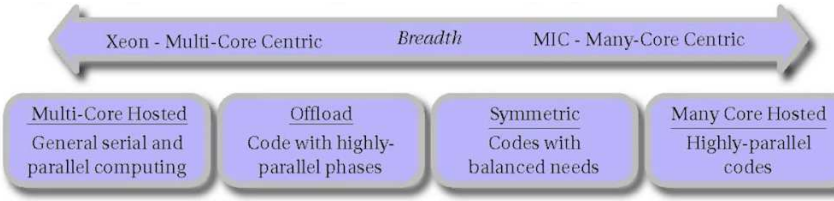
- Offload mode (explicit)



- Native mode (standalone application/ MPI process):



Coupling modes



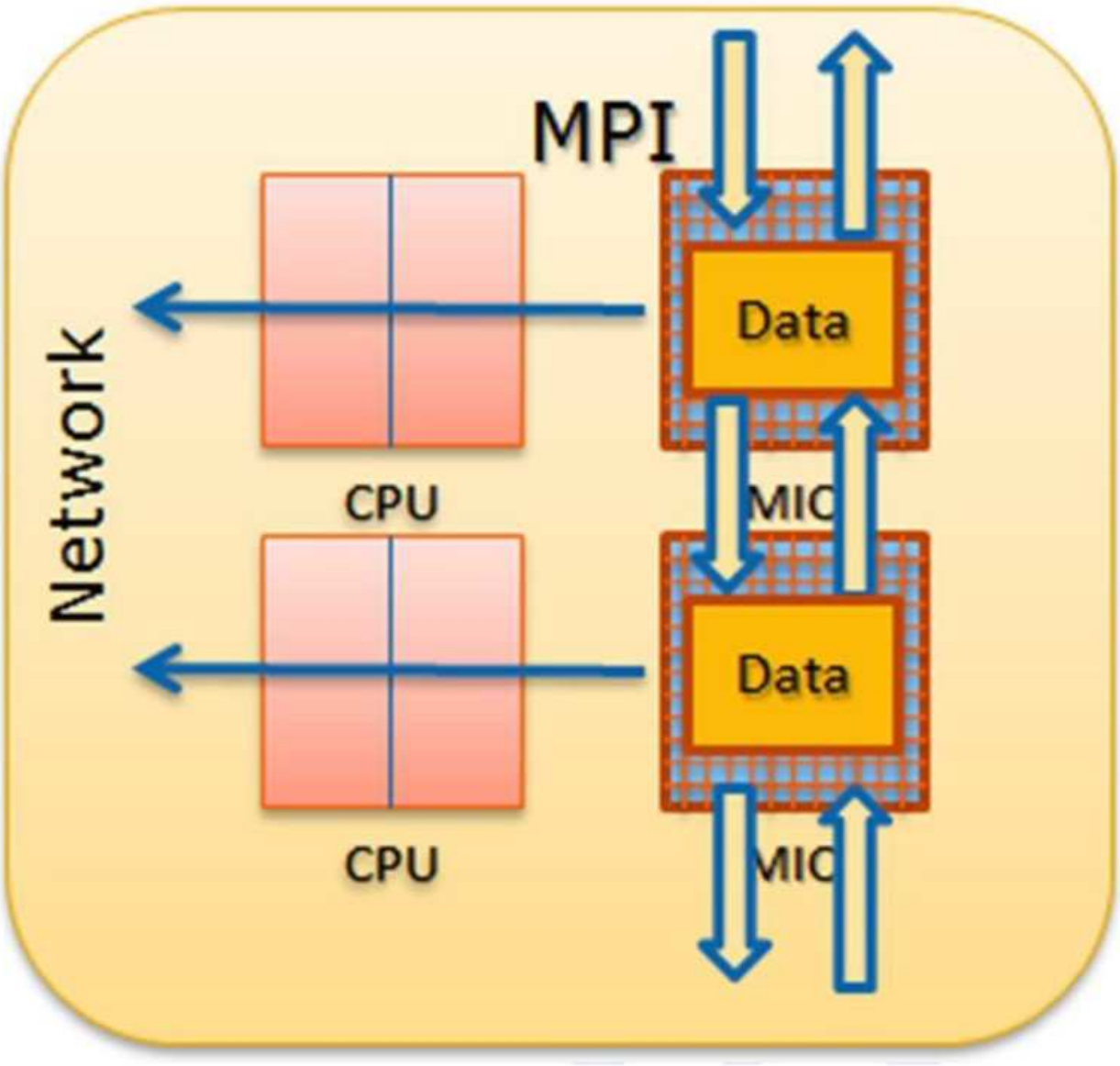
Programming models allow a range of CPU+MIC coupling modes

center

title Native Execution

Native execution

- The MPI/openMP processes reside on the MIC coprocessor only
- MPI/openMP libraries, the application, and other needed libraries are uploaded to the coprocessors



[fragile]

Native Execution

"Hello World" application

```
#include <stdio.h>
#include <unistd.h>

int main() {
  printf("Hello world! I have %ld logical cores.\n",
  sysconf(_SC_NPROCESSORS_ONLN));

}
```

```
icc hello.c
./a.out
```

Hello world! I have 16 logical cores. [fragile]

Native Execution

Compile and run the same code on the coprocessor in the native mode:

The tool `micnativeloadex` transfers code and dependent libraries and runs the application.

```
export SINK_LD_LIBRARY_PATH=/usr/local/intel/
        composer_xe_2015/lib/mic/
icc hello.c -mmic
micnativeloadex a.out
```

Hello world! I have 240 logical cores.

- Use -mmic to produce executable for MIC architecture
- Set `SINK_LD_LIBRARY_PATH` to help the tool find libraries
- Runs under micuser on coprocessor

[fragile]

Matrix-Matrix Multiplication — Native Mode

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

int main (int argc, char **argv)
{
   int n;
   int i, j, k;
   double time1, time2;
   # pragma omp parallel
   {
      int nthreads, threadid;
      nthreads = omp_get_num_threads ();
      threadid = omp_get_thread_num ();
      if (threadid == 0) {
         printf ("Open-MP, threads = %d\n", nthreads);
      }
   }
```

[fragile]

Matrix-Matrix Multiplication — Native Mode

```
if (argc != 2) {
   fprintf(stderr,"Usage: %s matrix size\n",argv[0]);
   exit (EXIT_FAILURE);
}
n = atoi (argv [1]);
if (n > 0) {
   printf ("Matrix size is %d\n",n);
}
else {
   fprintf (stderr, "Error, matrix size is %d\n", n);
   exit (EXIT_FAILURE);
}
double (*a)[n] = malloc (sizeof (double [n][n]));
double (*b)[n] = malloc (sizeof (double [n][n]));
double (*c)[n] = malloc (sizeof (double [n][n]));
```

[fragile]

Matrix-Matrix Multiplication — Native Mode

```
if (a == NULL || b == NULL || c == NULL) {
fprintf (stderr, "Not enough memory!\n");
   exit (EXIT_FAILURE);
}
for (i =0; i < n; i ++)
   for (j =0; j < n; j ++) {
      a [i][j] = ((double)rand())/((double)RAND_MAX);
      b [i][j] = ((double)rand())/((double)RAND_MAX);
      c [i][j] = 0.0;
   }
time1 = omp_get_wtime ();
# pragma omp parallel for private (k, j)
for (i =0; i < n; i ++)
   for (k =0; k < n; k ++)
      for (j =0; j < n; j ++) {
          c [i][j] += a [i][k]* b [k][j];
      }
time2 = omp_get_wtime () - time1;
```

[fragile]

Matrix-Matrix Multiplication — Native Mode

```
// check a random element
i = rand () % n;
```

```
    j = rand () % n;
    double d = 0.0;
    for (k =0; k < n; k ++)
       d += a [i][k]* b [k][j];

    printf ("Check␣randon␣element:␣%18.9lE\n",
            fabs (d - c [i][j]));
    printf ("Elapsed␣time␣(s)␣=␣%f\n", time2);
    return 0;
}
```

[fragile]

Running Native OpenMP Applications

Source code: `shared/xeonphi/hello_openmp.c`

To compile OpenMP program for native execution:

```
export SINK_LD_LIBRARY_PATH=/usr/local/intel/
        composer_xe_2015/lib/mic/
icc -openmp -mmic hello_openmp.c
```

[fragile]

Running Native OpenMP Applications

script:

```
#!/bin/bash
#SBATCH --time=00:01:00
#SBATCH --nodes=1
#SBATCH --gres=mic

echo 'Job␣number'
echo $SLURM_CPUS_ON_NODE

micnativeloadex a.out -a 100
```

To request a single Xeon Phi

center

title Explicit Offload

Offload Execution

- In this mode, MPI communication takes place only between host processors
- The co-processors are used only through the offload capabilities

Offload Execution

- Option 1: With compiler-assisted offload, you write code with offload annocations
  - No specific compiler flags needed, offload is implicit where markup is encountered
  - Offload code will automatically run on MIC at runtime if MIC is present, otherwise a host version is run.
- Option 2: With automatic offload, you link to a library that can perform offload operations (e.g., MKL, Math Kernel Library)
  - MKL is offload-capable; all you do is link to it (`-lmkl`)
  - Need to explicitly tell MKL to use offload at runtime via environment variable `MKL_MIC_ENABLE=1`

[fragile]

Offload: Pragma-based approach

"Hello World" in the explicit offload model:

```
#include <stdio.h>

int main(int argc, char * argv[]) {
  printf("Hello␣World␣from␣host!\n");

#pragma offload target(mic)
{
  printf("Hello␣World␣from␣coprocessors!\n");
}
  printf("Bye\n");
}
```

Application runs on the host, but some parts of the code and data are offloaded to the coprocessor.
[fragile]

Offload: Pragma-based approach

Important: set environment variables

```
source /usr/local/intel/composerxe/bin/
        compilervars.sh intel64
```

Request a single Xeon Phi coprocessor

```
# SBATCH --gres = mic
```

[fragile]

Offload: Pragma-based approach

To compile OpenMP program for offload execution

```
icc hello_offload.c -o hello_offload
./hello_offload
```

- No additional arguments if compiled with an Intel compiler
- Run application on host as a regular application
- Code inside of `#pragmaoffload` is offloaded automatically
- Console output on Intel Xeon Phi coprocessor is buffered and mirrored to the host console
- If coprocessor is not installed, code inside `#pragramoffload` runs on the host system

[fragile]

Offload Functions

```
__attribute__((target(mic))) void MyFunction() {
  // .. implement function as usual
}

int main(int argc, char *argv[]) {

#pragma offload target(mic)
{
  MyFunction();
}
}
```

- Functions used on coprocessor must be marked with the specifier `__attribute__((target(mic)))`
- Compiler produces a host version and a coprocessor version of such functions(to enable fall-back to host)

[fragile]

Offload Multiple Functions

- To mark a long block of code with the offload attribute, use `#pragma offload_attribute(push/pop)`

```
#pragma offload_attribute(push, target(mic))
void MyFunctionOne() {
// ... implement function as usual
}

void MyFunctionTwo() {
// ... implement function as usual
}
#pragma offload_attribute(pop)
```

[fragile]

Matrix-Matrix Multiplication — Offload Mode

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

void MatMatMult (int size,
   float (* restrict A)[size],
   float (* restrict B)[size],
   float (* restrict C)[size])
{
   # pragma offload target (mic) \
   in  (A : length (size * size)) \
   in  (B : length (size * size)) \
   out (C : length (size * size))
   {
```

[fragile]

Matrix-Matrix Multiplication — Offload Mode

```
      # pragma omp parallel for \
              default (none) shared (C, size)
      for (int i = 0; i < size; i ++)
         for (int j = 0; j < size; j ++)
            C [i][j] = 0.0 f;

      # pragma omp parallel for default (none) \
                                shared (A, B, C, size)
      for (int i = 0; i < size; i ++)
        for (int j = 0; j < size; j ++)
           for (int k = 0; k < size; k ++)
              C [i][j] += A [i][k] * B [k][j];
   }
```

```
}
```

[fragile]

Matrix-Matrix Multiplication — Offload Mode

```
int main (int argc, char * argv [])
{
   if (argc != 4) {
      fprintf (stderr, "Use␣:␣%␣s␣size␣nThreads␣nIter\n",
               argv [0]);
      return -1;
   }
   int i, j, k;
   int size = atoi (argv [1]);
   int nThreads = atoi (argv [2]);
   int nIter = atoi (argv [3]);
   omp_set_num_threads (nThreads);
   float (*restrict A)[size] = malloc(sizeof (float)
                                      * size*size);
   float (*restrict B)[size] = malloc(sizeof (float)
                                      * size*size);
   float (*restrict C)[size] = malloc(sizeof (float)
                                      * size*size);
```

[fragile]

Matrix-Matrix Multiplication — Offload Mode

```
   # pragma omp parallel for default (none) \
            shared (A,B, size) private (i,j, k)
   for (i = 0; i < size; i ++)
      for (j = 0; j < size; j ++)
      {
         A [i][j] = (float) i + j;
         B [i][j] = (float) i - j;
      }
   double avgMultTime = 0.0;
   MatMatMult (size, A, B, C); // warm up
   double startTime = dsecnd ();
   for (int i  =0; i < nIter; i ++)
       MatMatMult (size, A, B, C);
   double endTime = dsecnd ();

   avgMultTime = (endTime - startTime)/ nIter;
```

[fragile]

Matrix-Matrix Multiplication — Offload Mode

```
   # pragma omp parallel
   # pragma omp master
```

```
    printf("%s␣nThrds␣%d␣mx␣%d␣%d␣time␣%g␣GFlop/s␣%g\n",
       argv [0], omp_get_num_threads (), size, size,
       avgMultTime, 2e-9*size*size*size  avgMultTime);

    # pragma omp barrier
    free (A); free (B); free (C);
    return 0;
}
```

Reference

http://www.colfax-intl.com/nd/resources/slides.aspx
   http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/

GPU – A short background

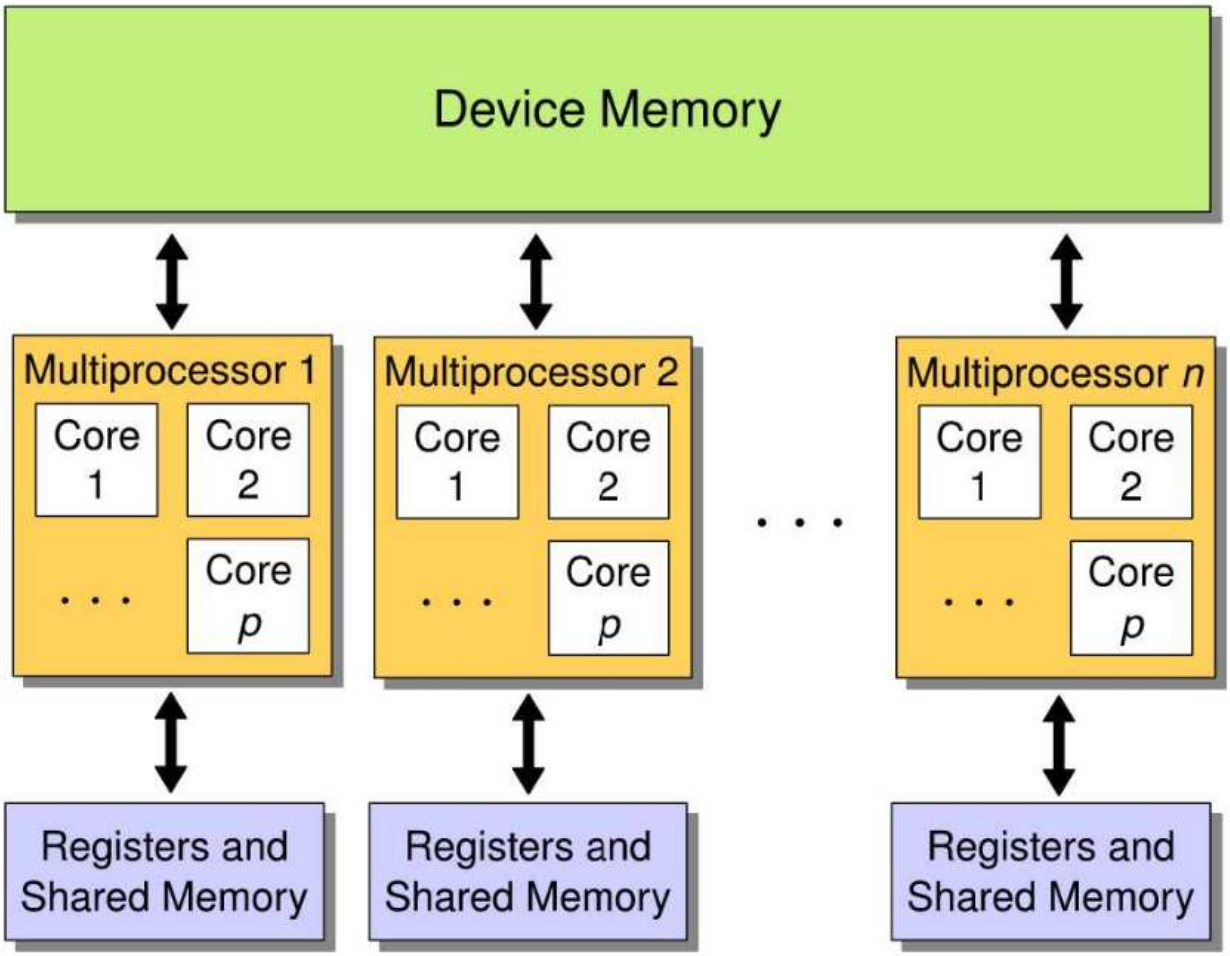
What is a GPU?

A Graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.

- GPUs can be foun in various computing devices:
  - Embedded systems, mobile phones, PCs, workstations, game consoles, and even supercomputers
  - A GPU can be present on a video card, or it can be embedded on the modeherboard or the CPU die

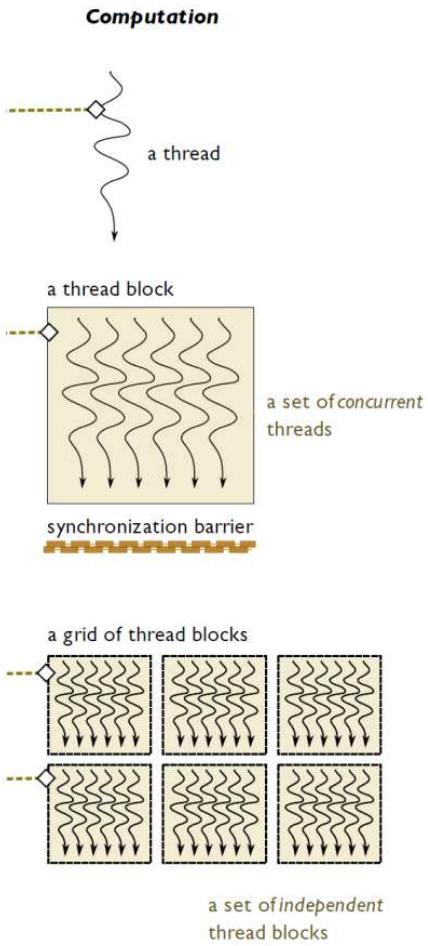
GPU components

- Hardware:
  - **SP**: streaming processor (AMD), aka CUDA core (Nvidia)
  - **SM**: streaming multiprocessor
- Software:
  - **Thread**: the smallest sequence of programmed instructions
  - **Block**: a bunch of threads that execute in a single SM and communicate through shared memory, aka *warp* (Nvidia)
  - **Grid**: a bunch of blocks that execute a kernel function



Execution hierarchy

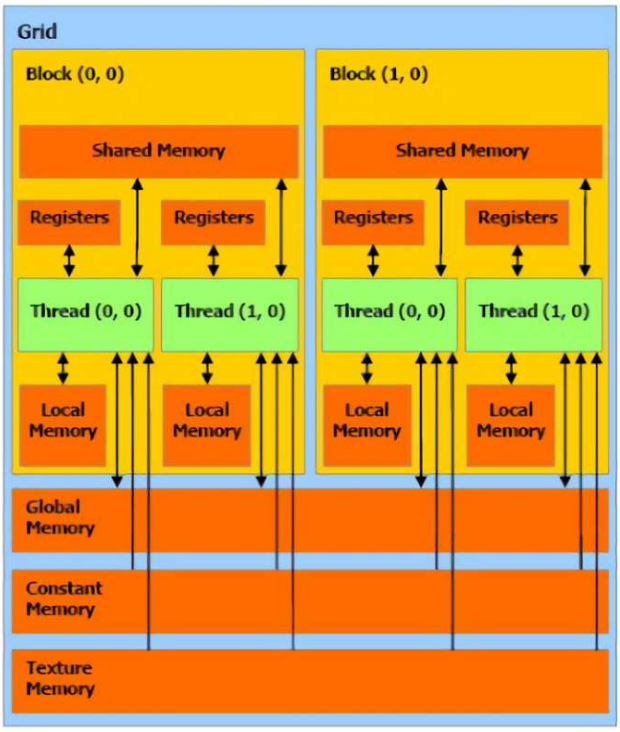
- Thread: smallest execution entity
  - Each thread has its own ID
  - Thousands of threads executing the same program logic
- Threads are grouped into blocks
  - Threads in a block can synchronize execution
- Blocks are grouped in a grid
  - Blocks are independent (must be able to be executed in any order)



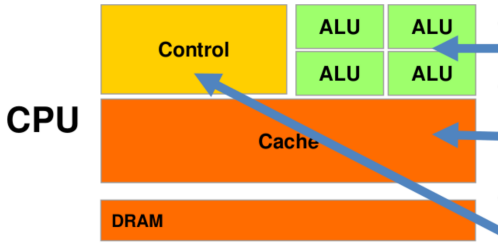
Memory hierarchy

- There are three types of memory
  - Global memory: 0.5 – 24 GB, with most now having ∼4 GB
  - Shared memory: ∼48 kB using hardware L1 cache
  - Registers and local memory: word width 32 bit
- Latency of memory access
  - Global memory: ∼300 ns
  - Shared memory: 5 ns
  - Registers and local memory: Fastest "memory", about 10× faster than shared memory
- Purposes
  - Global memory: I/O for grid
  - Shared memory: thread collaboration within a block

- Registers: store stack vars

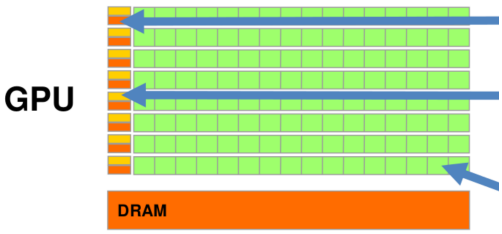


GPU vs CPU



CPU: Latency-oriented Design

- Powerful ALU: Reduced operation latency
- Large caches: Reduce memory latency
- Sophisticated control
  - branch prediction
  - data forwarding



GPU: Throughput-oriented design

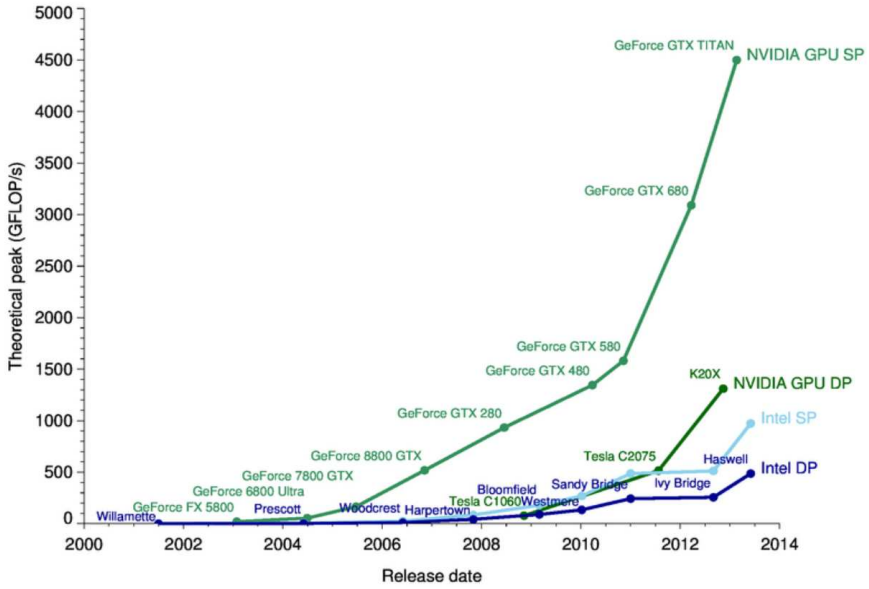
- Small caches boost memory throughput
- Simple control
- Energy efficient ALUs
  - Many
  - Long latency, but heavily pipelined
- Require massive number of threads to tolerate latencies

GPUs in parallel computing

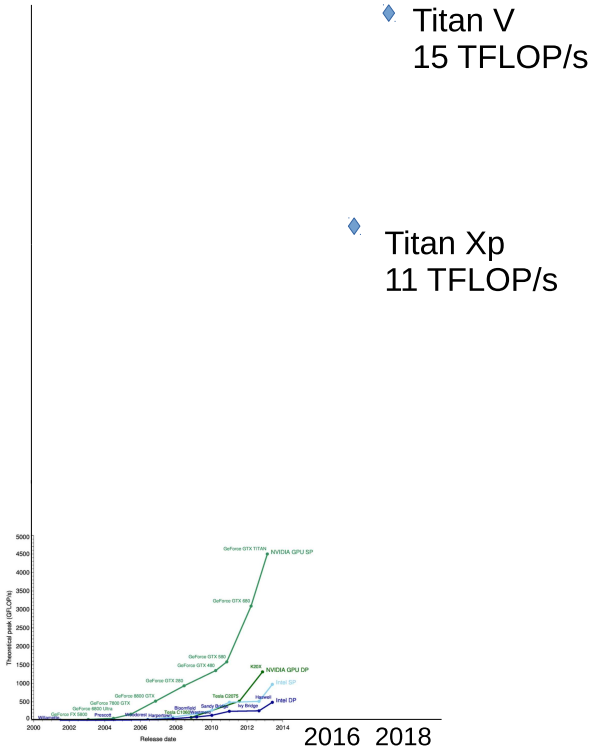
Advantages of using GPUs for parallel computation

The highly parallel structure maes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel.

- Massively parallel
- Highly scalable
- Rapidly advancing



GPUs in parallel computing



GPUs in parallel computing

- GPU acceleration can yield impressive speed-up for some algorithms
- The speed-up ratio depends on the non-parallel part: **Amdahl's Law**
- **Significance**: an accelerated program on a GPU can be as fast as its serial part.
- (Remember Gustavson's Law: If the serial part doesn't grow as the problem size grows, then it becomes insignificant.)

Figure

CUDA – A short background

What is CUDA?

CUA is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and softwre engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing.

- CUDA stands for **C**ompute **U**nified **D**evice **A**rchitecture

- That means it reveals a little bit about the GPU's architecture to allow parallel processing, but stays abstract enough that code is portable across Nvidia devices.
- CUDA is a compiler and toolkit for programming Nvidia GPUs
- The CUDA API extends C/C++, adds directives to translate them into instructions than run on the host CPU or GPU when needed
- CUDA allows for easy multi-threading – parallel executing on all streaming processors on the GPU

Two sides of CUDA

Advantages

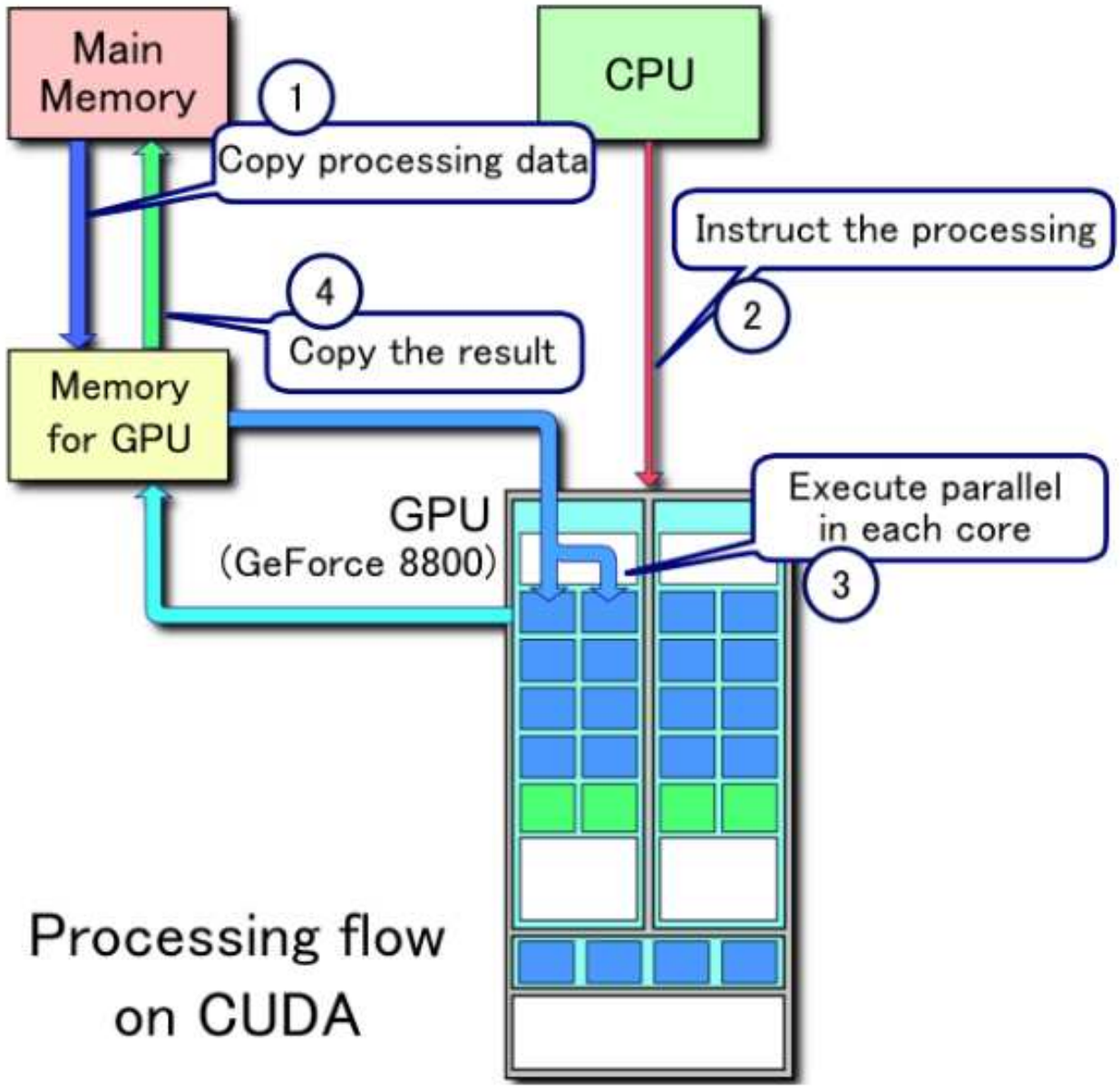
- Abstraction from the hardware
  - Programmers don't see every little aspect of the machine
  - Gives flexibility to the vendor to update hardware but keep legacy code forward compatible
- Automatic thread management
  - Multi-threading: hides latency and helps maximizing the GPU utilization
  - Transparent for the programmer
  - Limited synchronization between threads is provided
  - Avoid dead-lock (no message passing)

Disadvantages

- Vendor-lock to Nvidia (Alternative architecture-independent schemes – e.g., OpenCL – require more programming effort)
- Still difficult to program; impossible to accelerate chaotic code flow
- Hard to debug (not even printf in early versions)

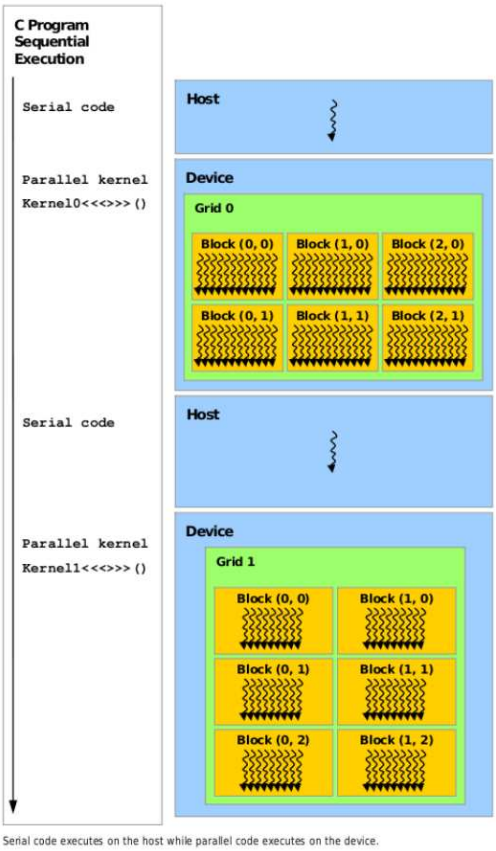
Example of CUDA processing flow

1. Copy data from main memory to GPU memory
2. CPU initiates the GPU compute kernel
3. GPU's CUDA cores execute the kernel in parallel
4. Copy the resulting data from GPU memory to main memory



Example of program execution flow

1. Host executes serial code
2. Device executes parallel kernel 0
3. Host executes serial code
4. Device executes parallel kernel 1



Serial code executes on the host while parallel code executes on the device.

Programming in CUDA

Basic C extensions

- Function modifiers: programmer can define where a function should run
  - __host__ : to be called and executed by the host CPU
  - __device__ : to run on the GPU, and the function can only be called by code running on the GPU
  - __global__ : to run on the GPU but called from the host. This is the access point to start multi-threaded code on the GPU
- Variable modifiers
  - _device_ : the variable resides in the GPU's global memory and is defined while the code runs
  - _shared_ : variable in shared memory, with the same lifespan as the block.
- _syncthreads(): sync of threads within a block

writing a __global__ function

- All calls to a global function must specify how many threaded copies to launch and in what configuration
- CUDA syntax: $<<<\cdots>>>$
  - Inside the $<<<>>>$, we need at least two arguments (can be more to overwrite default values)
  - Call example: *my_func* $<<<$*bg, tb*$>>>$(*arg1*,*arg2*)
  - *bg* specifies the dimensions of the block grid
  - *tb* specifies the dimensions of each thread block
  - *bg* and *tb* are both of type *dim3* (a new data type defined by CUDA: three unsigned ints where any unspecified component defaults to 1)
  - *dim3* has struct-like access: members are x, y and z
  - 1-D syntax allowed: *myfunc*$<<<$*5, 6*$>>>$() makes 5 blocks in a linear array, with 6 threads each, and runs *myfunc* on them all.

Allowing the CUDA kernel to get data

- Allocate CPU memory for n integers, e.g., malloc (...)
- Allocate GPU memory for n integers, e.g., cudaMalloc(...)
- Copy the CPU memory to GPU memory for n integers, e.g., cudaMemcpy(..., cudaMemcpyHostToDevice)
- Copy the GPU memory to CPU once computation is done, e.g., cudaMemcpy(..., cudaMemcpyDeviceToHost)
- Free the GPU and CPU memory, e.g., cudaFree(...)

[fragile]

Example: Vector adder

Simple example: add two arrays

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

[fragile]

Example: Vector adder

1. Memory allocation
2. Memory copy: Host → GPU
3. Kernel call
4. Memory copy: GPU → Host
5. Free GPU memory

```
// Host code
int main ()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcopyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcopyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
          (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the resut in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

[fragile]

Example: Caeser Cipher

```
__global__ void shift_cypher (
    unsigned int * input_array, unsigned int * output_array,
    unsigned int shift_amount, unsigned int alphabet_max,
    unsigned int array_length)
{
    unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int shifted = input_array [tid] + shift_amount ;
    if (shifted > alphabet_max)
        shifted = shifted % (alphabet_max + 1);
    output_array [tid] = shifted ;
}
```

[fragile]

Example: Caeser Cipher

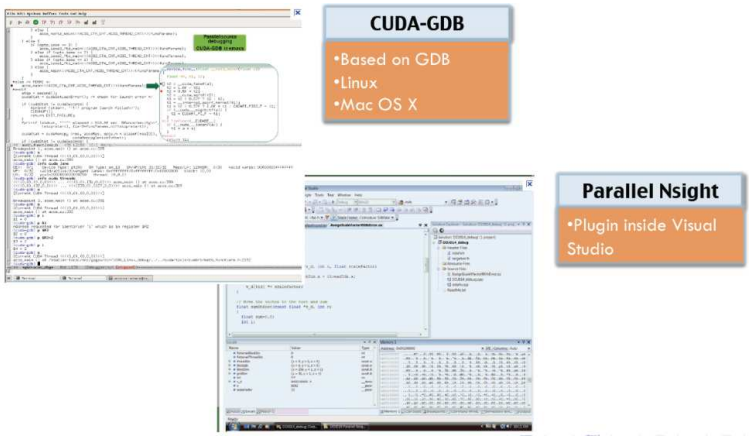
```
int main () {
    ...
    unsigned int num_bytes = sizeof (int) * (1 << 22);
    unsigned int *input_array = 0;
    unsigned int *output_array = 0;
    ...
    cudaMalloc ((void**)&input_array, num_bytes);
    cudaMalloc ((void**)&output_array, num_bytes);
    cudaMemcpy (input_array, host_input_array, num_bytes,
                cudaMemcpyHostToDevice);
    dim3 dimGrid (ceil (array_length)/ block_size);
    dim3 dimBlock (block_size);
    // gpu will compute the kernel and transfer the results
    // out of the gpu to host.
    shift_cypher<<<dimGrid, dimBlock>>>(input_array, output_array,
                      shift_amount, alphabet_max, array_length);
    cudaMemcpy (host_output_array, output_array, num_bytes,
                cudaMemcpyDeviceToHost);
    ...
    // free the memory
    cudaFree (input_array);
    cudaFree (output_array);
}
```

Compiling CUDA program

- CUDA code must be compiled using *nvcc*
- *nvcc* generates both instructions for CPU and GPU (PTX instruction set), as well as instructions to send data back and forwards between them

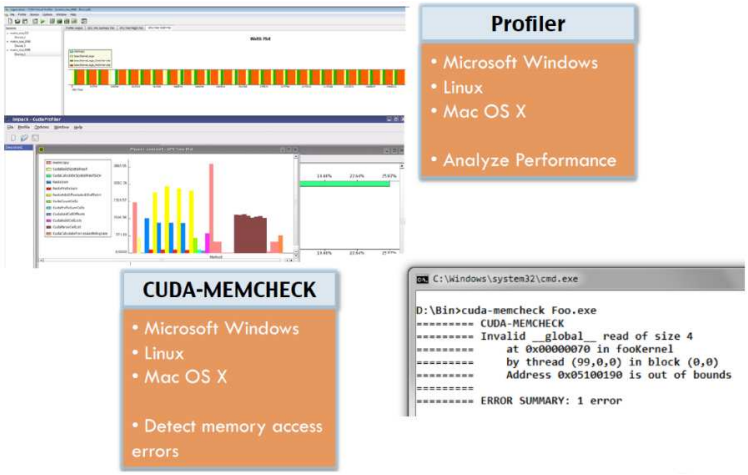
Debugging a CUDA program

- Various tools are available in the market for simultaneous CPU and GPU debugging
  - Set breakpoints and conditional breakpoints
  - Dump stack frames for thousands of CUDA threads
  - Inspect memory, registers, local/shared/global variables
- Runtime error detection
  - Supports multiple GPUs, multiple contexts, multiple kernels



Profiling a CUDA program

- The *Visual Profiler* is a graphical profiling tool that displays a timeline of your application's CPU and GPU activity
- *Profiler* also includes an automated analysis engine to identify optimization opportunities
- The *nvprof* profiling tool enables you to collect and view profiling data from the command line



An architecture-independent alternative — OpenCL

- CUDA and Intel's MIC interface were written by companies who want to encourage people to use their hardware
  - Ease of use is vital
  - Exploiting this specific architecture to the fullest is desired
  - Portability of code to other vendors is actually a disadvantage for the framework designers
- OpenCL is a framework for writing portable many-core code
  - Code should run on GPUs, Xeon Phi, CPU, DSPs, FPGAs
  - For maximum portability, kernel is compiled at runtime
    - Can part-compile to intermediate representation SPIR-V at compile time

[fragile]

A "Hello world" program in openCL

```
#define CL_USE_DEPRECATED_OPENCL_2_0_APIS

#include<CL/cl.hpp>
#include<iostream>
#include <fstream>

int main()
{
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);
```

```
    auto platform = platforms.front();
    std::vector<cl::Device> devices;
    platform.getDevices(CL_DEVICE_TYPE_CPU, &devices);

    auto device = devices.front();
```

[fragile]

A “Hello world” program in openCL

```
    std::ifstream helloWorldFile("hello.cl");
    std::string src(std::istreambuf_iterator<char>(helloWorldFile),
                    (std::istreambuf_iterator<char>()));

    cl::Program::Sources sources(1, std::make_pair(src.c_str(),
                                 src.length() + 1));

    cl::Context context(device);
    cl::Program program(context, sources);

    auto err = program.build("-cl-std=CL1.2");

    char buf[16];
    cl::Buffer memBuf(context,
                      CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
                      sizeof(buf));
    cl::Kernel kernel(program, "HelloWorld", &err);
    kernel.setArg(0, memBuf);
```

[fragile]

A “Hello world” program in openCL

```
    cl::CommandQueue queue(context, device);
    queue.enqueueTask(kernel);
    queue.enqueueReadBuffer(memBuf, GL_TRUE, 0, sizeof(buf), buf);

    std::cout << buf;
}
```

[fragile]

OpenCL “Hello world” program

```
__kernel void HelloWorld(__global char* data)
{
```

```
    data[0] = 'H';
    data[1] = 'E';
    data[2] = 'L';
    data[3] = 'L';
    data[4] = 'O';
    data[5] = ' ';
    data[6] = 'W';
    data[7] = 'O';
    data[8] = 'R';
    data[9] = 'L';
    data[10] = 'D';
    data[11] = '!';
    data[12] = '\n';
    data[12] = 0;
}
```

Reference

http://lorenabarba.com/gpuatbu/Program_files/ Cruz_gpuComputing09.pdf https://www.slideshare.net/piyushmittalin/a-beginners-guide-to-programming-gpus-with-cuda https://www.slideshare.net/RaymondTay1/introduction-to-cudas https://www.khronos.org/opencl

Interconnection networks

Types of networks

Both the shared memory and distributed memory architectures require an interconnection network to connect the processor and memory or the modules respectively.

Interconnection networks can be broadly categorized into three types:

1. buses,
2. static networks and
3. switching networks.

Static networks are sometimes called point-to-point networks.

The interesting aspects to consider are the complexity of communication between the processors and memory, or between modules, and the physical cost (wiring complexity) of the network.

Multi-bus systems

A full b-bus memory connection system with p processors and m memories uses $b\,(p+m)$ connections (not counting the buses themselves).

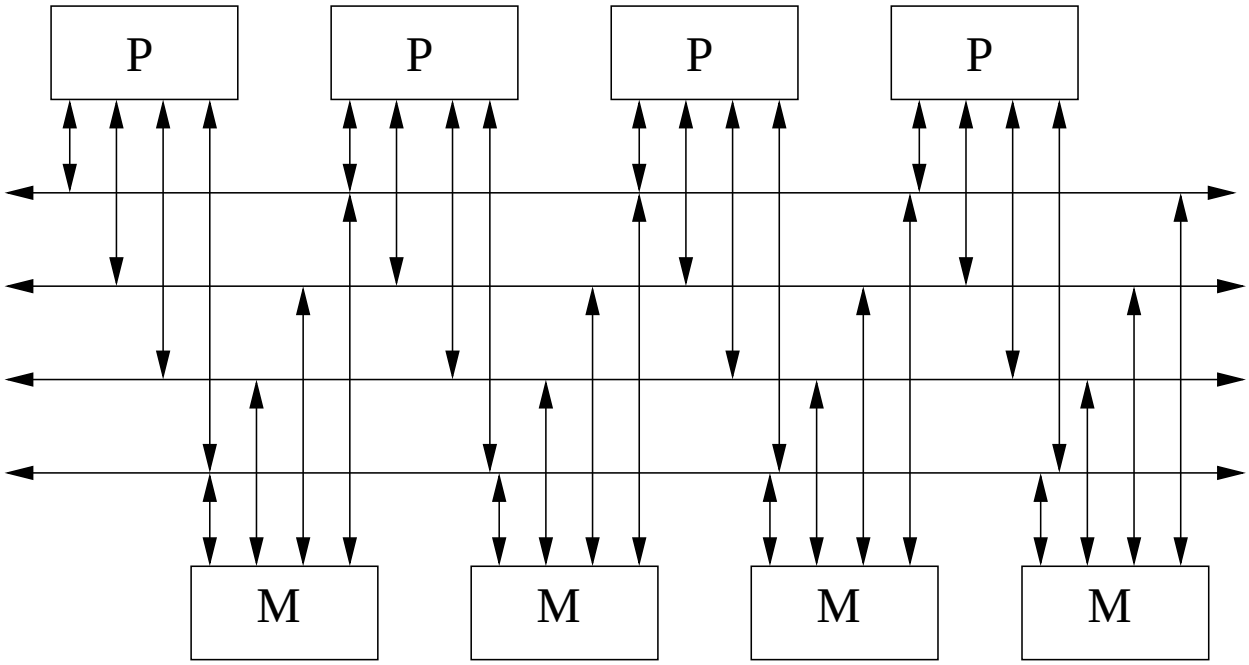


Figure 35: full memory connection

A partial b-bus memory connection system with p processors and m memories uses $b\,(p + \frac{m}{g})$ connections where $1 < g < b$.

A single b-bus memory connection system with p processors and m memories uses $b\,p + m$ connections.

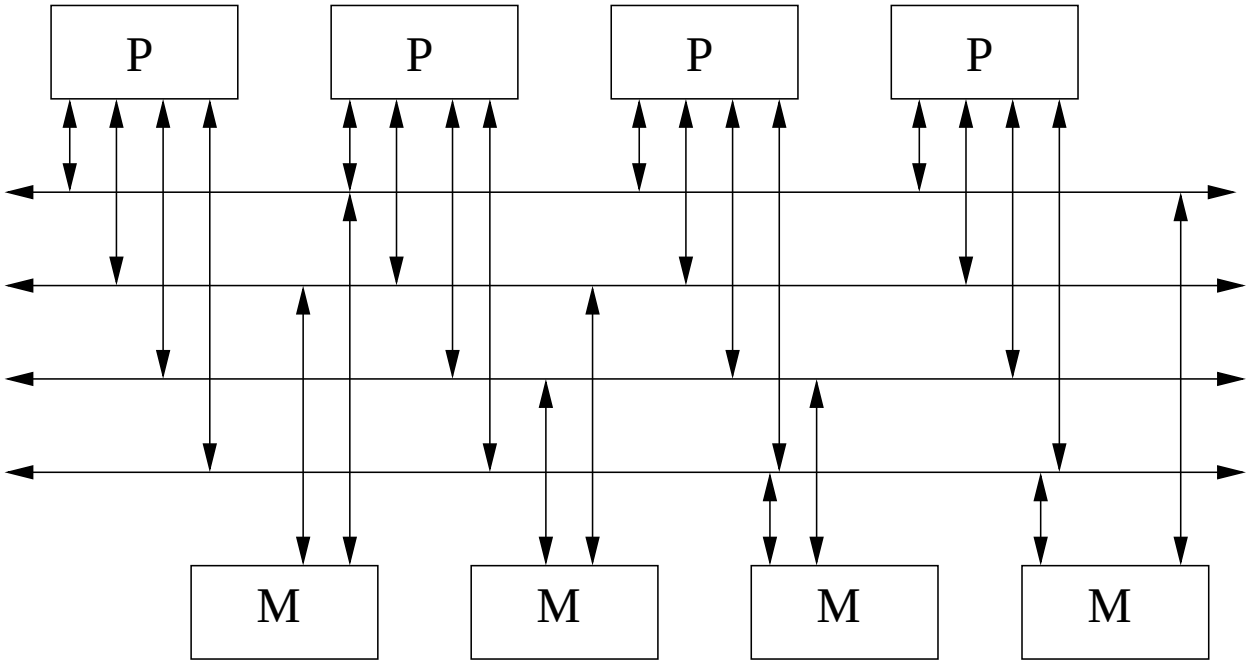


Figure 36: partial memory connection

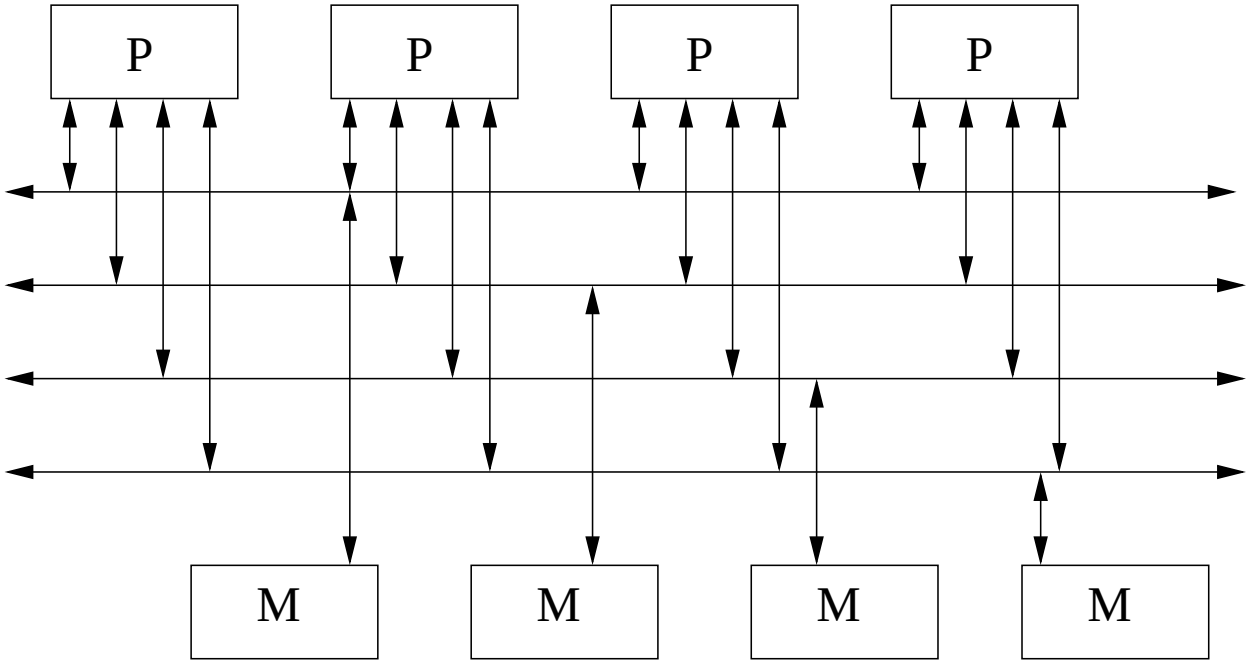


Figure 37: single memory connection

Bus characteristics

Access to a bus is arbitrated by a *bus master*.

Each node on a bus has a bus master which requests access to the bus, called a *bus request*, when then node requires to use the bus. This is a global request sent to all nodes on the bus.

The node that currently has access to the bus responds with either a *bus grant* or a *bus busy* signal, which is also globally known to all bus masters.

If all n nodes on a bus are continuously requesting access then a bus can supply $O(1/n)$ of its available bandwidth to each node.

The *minimum latency* on a bus, i.e. with no contention, is constant or $O(1)$ (which ignores signal propagation delay).

Static networks

A static network has fixed point-to-point connections between modules in the system. Each connection is a dedicated communication channel between the pair of modules.

Message passing is required between the modules for any module to access the memory of any other module (for a shared addresss space machine) or for processors to communicate (in a message passing machine).

The number of "hops" that a message requires is a dominant factor determining the latency for the memory access or communication.

The bandwidth of the connections and/or the routing technology/algorithms also determine the latency.

Communication module

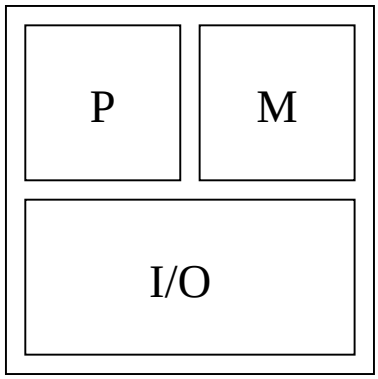


Figure 38: Processor, memory and communication module.

The communication module may be as straight forward as an Ethernet connection, or multiple Ethernet connections, or may be more dedicated hardware.

Topological properties of static networks

Topological properties are used to describe a given architecture.

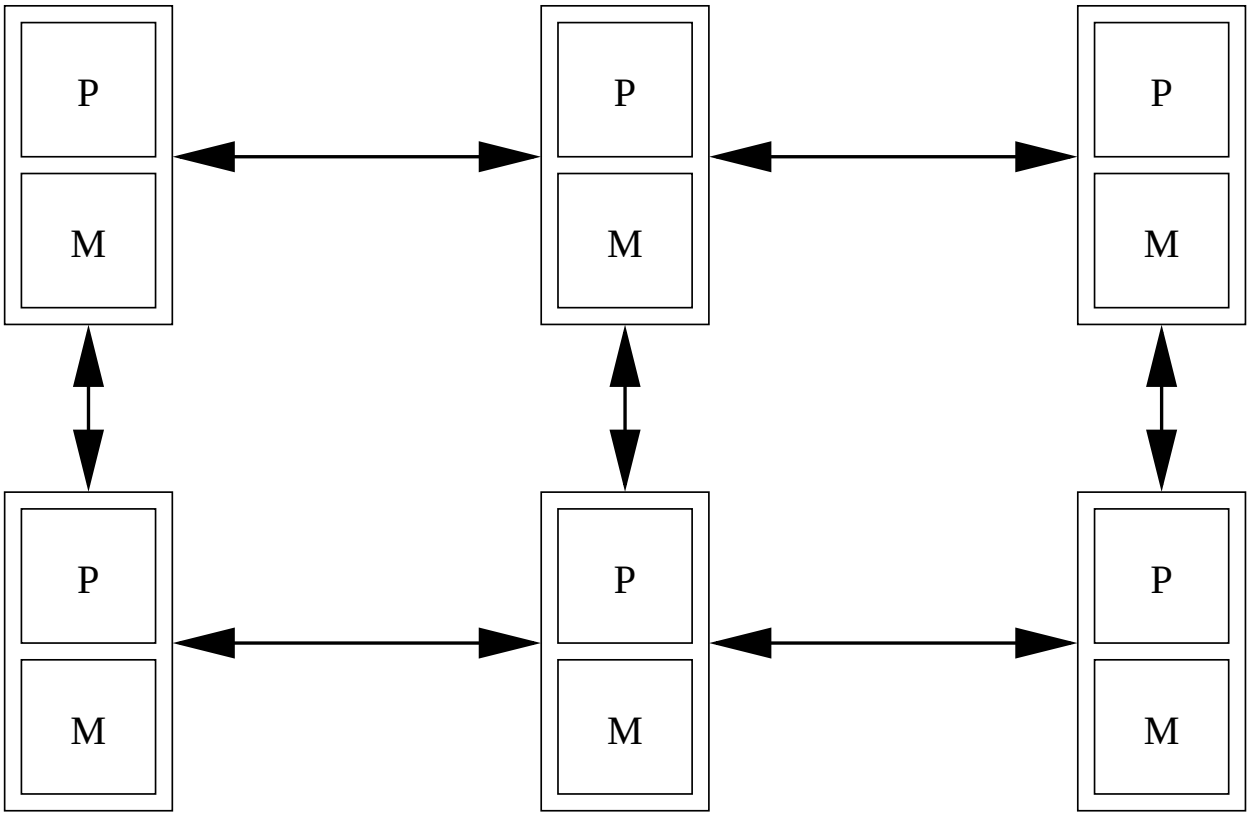


Figure 39: Distributed memory machine example

For static networks it is desirable to achieve a low "*cost*":
$c = d\,k$

degree (d) : the maximum number of edges connected to a single vertex in the graph.

diameter (k) : the minimum number of edges a message must traverse in order to be communicated between any two vertices in the graph.

Note that this "cost" is a compromise between the cost to build (d) and the communication delay (k), not just the cost to build.

Other properties of interest:

bisection width : The smallest number of edges that, when cut, would separate the network into two halves. Sometimes we call this edge bisection width and then also consider node bisection width.

planarity : Can the network be embedded in a plane without any edges crossing. This is particular useful for integrated circuits as it eliminates the need for multiple layers.

symmetry : Are all nodes topologically the same.

Graph theory is essential for the study of interconnection networks. A network is a graph, $G = (V, E)$, where $V = \{1, 2, 3, \cdots, n\}$ is the set of n vertices and $E \subseteq V \times V$ is the set of edges. Each vertex can for instance be a processing node or a process and each edge can represent communication or physical connections between nodes. The size of the graph is n and usually the graph is a function of n, $G(n)$, with *topological* properties of the graph being described as functions of n. Degree, diameter and cost are topological properties.

Maintaining a small degree is important because the degree is directly proportional to the complexity of the node. For example, if the network is fabricated as an integrated circuit then each node requires up to d communication connections. Clearly, a star topology that has $n - 1$ nodes connected to a central node is not feasible to construct for arbitrary large n.

Maintaining a small diameter is important because the diameter is directly proportional to the message passing latency, i.e. the time taken for a message from one node to reach another node.

Minimizing both degree and diameter (thus minimizing cost) is a fundamental problem for parallel computing. Other topological properties include *planarity* (whether the network can be drawn in a plane without any edges crossing) and *bi-section width* (the number of edges that must be cut in order to disconnect the network into two halves).

Degree 2 and diameter $O(n)$, has cost $O(n)$.

Generally the ring network with n vertices has $k = \lfloor \frac{n}{2} \rfloor$ and $d = 2$ so the cost is $2 \lfloor \frac{n}{2} \rfloor = O(n)$.

Topological parameters are often described simply by their asymptotic complexity. Asymptotic complexity must be used with care when comparing a property over a number of different networks. This is because it may be that the complexity only holds for values of n_0 that are excessively large, much larger than any practical system could be implemented to include.

The cost of a completely connected network is $O(n)$.

Node i is connected to node j, if $|j - i| = 2^r$ for some $r = 0, 1, 2, \ldots, k - 1$, where $n = 2^k$. The degree is $2k - 1$ and the diameter is $k/2$. Cost is $O(\log(n)^2)$

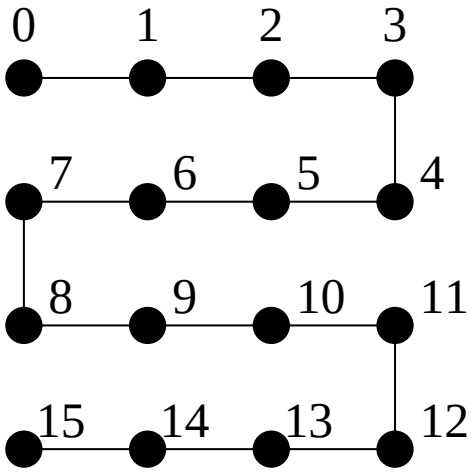


Figure 40: A linear array.

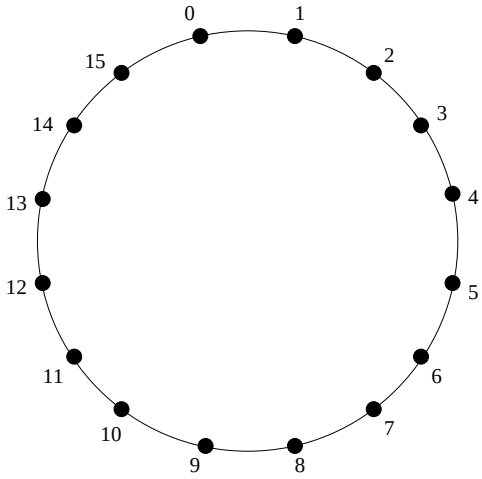


Figure 41: A ring network.

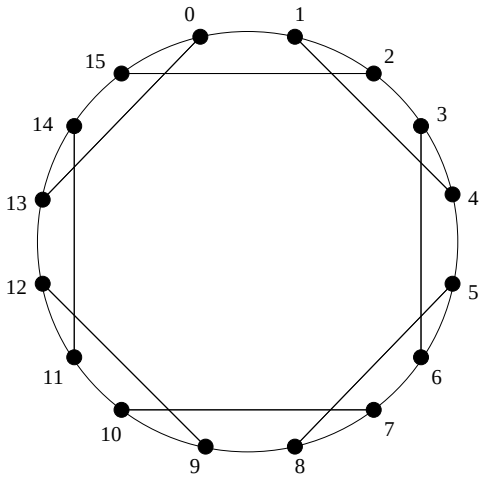


Figure 42: A chordal ring of degree 3.

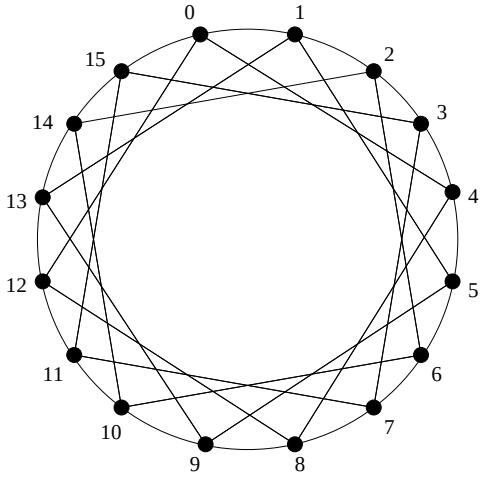


Figure 43: A chordal ring of degree 4.

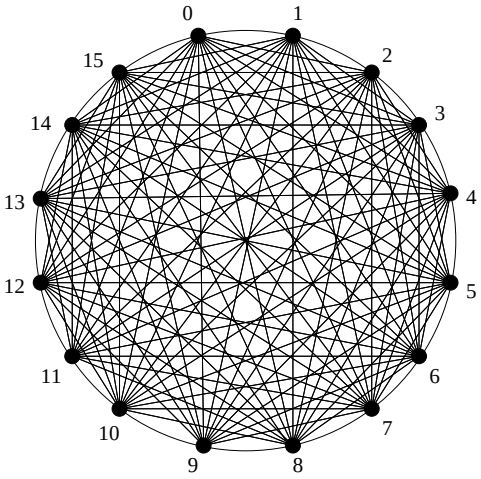


Figure 44: A completely connected network.

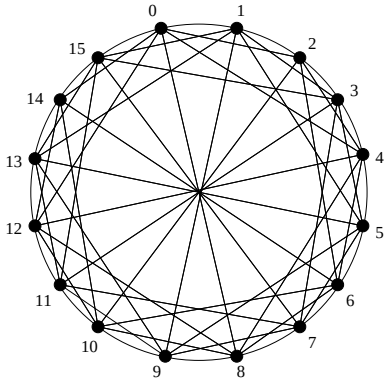
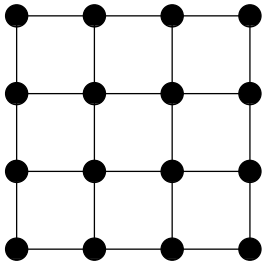


Figure 45: Barrel shifter.



$d = 4$

$k = 6$

$c = d\,k = 24$

Figure 46: A mesh network.

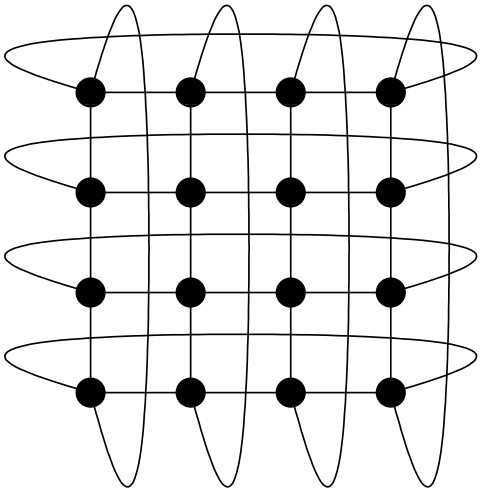
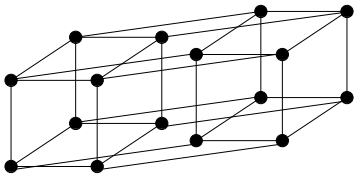


Figure 47: A torus network.



$d = 4$
$k = 4$
$c = d\,k = 16$

Figure 48: A hypercube network.

Generally the mesh network with $n = i \times j$ vertices has $k = i + j - 2$ and $d = 4$ so the cost is $4(i + j) - 8$ which is $8\sqrt{n} - 8 = O\big(\sqrt{n}\big)$ if $i = j$. Clearly the mesh has a smaller cost than a ring when n is large.

Generally the hypercube network with $n = 2^t$ vertices has $k = t$ and $d = t$ so the cost is t^2 or $\log_2^2 n = O\big(\log^2 n\big)$. The cost is far less than the previous two examples, for large n.

A network called Cube Connected Cycles (CCC), is constructed from a hypercube of size $n = 2^t$ nodes by replacing each node (of degree t) with a cycle of length t and connecting each incoming edge to one node of the cycle. Each node in the cycle now has degree 3.

Cube Connected Cycles

- The CCC network will now have $t\,2^t$ nodes.
- The degree is 3.
- The diameter is non-trivial to show; see the literature for a proof that it is $2\,t + \lfloor \frac{t}{2} \rfloor - 2$ for $t > 3$.
  - Check by inspection that diameter is 6 for $t = 3$.
- Prove for yourself that $t = \Theta(\frac{\log n}{\log \log n})$, which gives a cost of $O(\frac{\log n}{\log \log n})$.

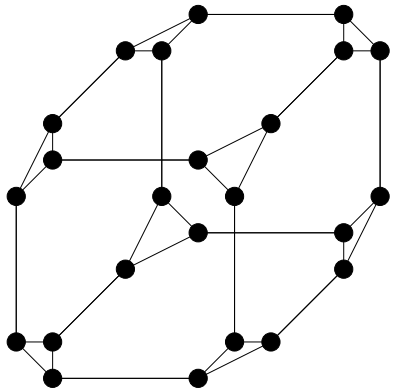
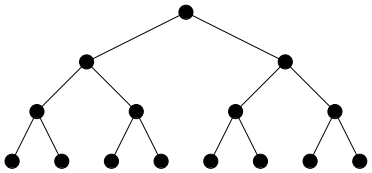


Figure 49: A CCC network: Cube Connected Cycles



$d = 3$

$k = 6$

$c = d\,k = 18$

Figure 50: A tree network.

For a tree with degree d and n vertices, $k \leq 2\lfloor \log_{d-1} n \rfloor = O(\log_d n)$ (twice the height of the tree) so that the cost is $O(d \log_d n)$. Setting $d = \log n$ gives a cost of $O\big(\frac{\log^2 n}{\log \log n}\big)$ which is better than the hypercube.

An interconnection network requires a routing algorithm that determines how messages from one node can reach another node. For the mesh network the routing algorithm may be intuitively described as "from the source, move up or down to the row containing the destination and then move left or right to the destination". As an exercise, devise a routing algorithm for the hypercube of size $n = 2^t$. Hint: you may label the nodes with binary strings.

Interconnection networks provide spatial parallelism at different levels of a machine architecture. They may be used at the integrated circuit level to provide a mesh of ALU's. They may be used at the package level to provide inter-processor and/or processor-memory communication. They may be used at the machine level to provide networks of machines and so on.

The fat tree is a practical extension of the tree that has progressively more wiring towards the root of the tree. This provides for greater data flow between the subtrees.

Data centre "fat tree"

- K-ary fat tree: three-layer topology (edge, aggregation and core)
  - each pod consists of (k/2) 2 servers + 2 layers of k/2 k-port switches
  - each edge switch connects to k/2 servers + k/2 aggr. switches
  - each aggr. switch connects to k/2 edge + k/2 core switches
  - (k/2) 2 core switches: each connects to k pods

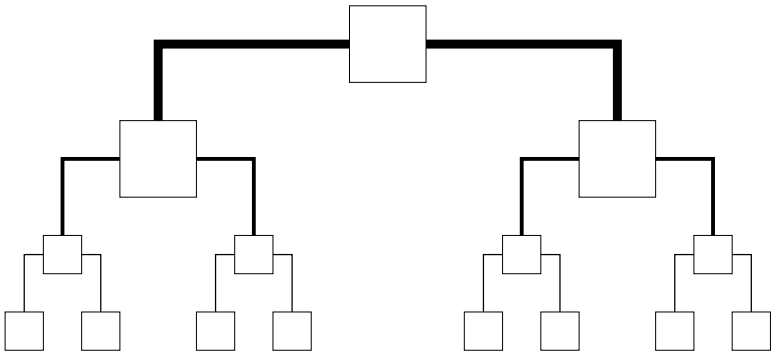


Figure 51: Fat Tree

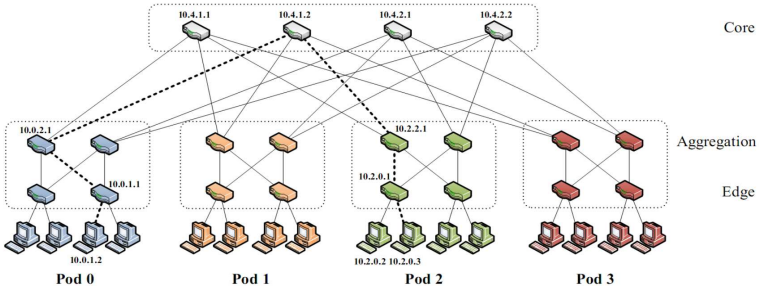


Figure 52: "Fat tree" with commodity switches

- Top two layers aren't processor/memory/communication modules.
- We now consider: What do these look like internally?

Switching networks

A basic switch has two inputs, say A and B and two outputs, say A' and B'. It is either in the direct setting where $A' = A$ and $B' = B$, or in the cross-over setting where $A' = B$ and $B' = A$.

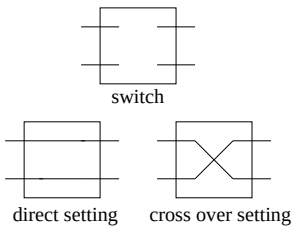


Figure 53: A switch.

With switching networks, the hardare cost is no longer proportional to the degree d. Instead, we measure the complexity as the number of switching elements.

Other settings are also possible.

A *crossbar* switching network has complexity $O(n^2)$ where n is the number of inputs and outputs. The $n \times n$ crossbar is comparable to an n-bus with single bus memory connections.

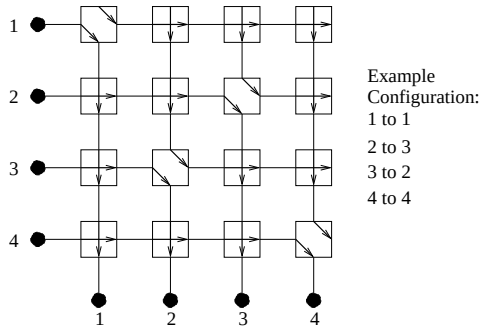


Figure 54: An example crossbar switching network.

The minimum latency for a crossbar is $O(1)$, i.e. any 1-to-1 mapping of inputs to outputs is possible.

A large crossbar (224×224) was actually built in a vector parallel processor (VPP500) by Fujitsu Inc. in 1992, which using a distributed memory architecture, i.e. the crossbar is used for interprocessor communication.

Clos Network

The Clos network is defined by integers n, m and r.

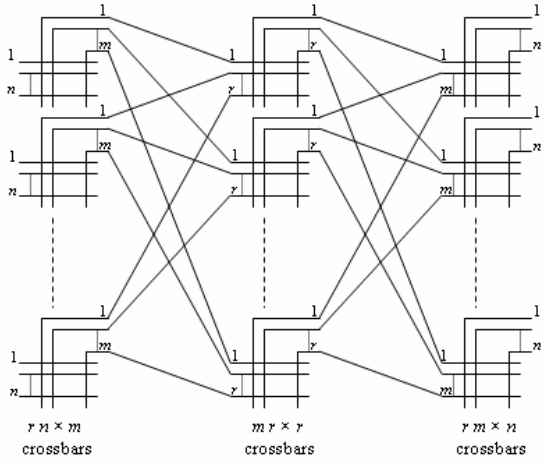


Figure 55: A Clos Network.

The Clos network connects $N = rn$ inputs to N outputs. A straight forward crossbar would require $N^2 = r^2n^2$ switches. However the Clos network requires $2rnm + mr^2$ switches.

If $m \geq 2n - 1$, the Clos network is said to be *strict-sense nonblocking*, meaning that an unused input on an ingress switch can always be connected to an unsed output on an egress switch, *without having to re-arrange any existing connections.*

If $m \geq n$, the Clos network is said to be rearrangeably nonblocking, meaning that an unused input on an ingress switch can always be connected to an unused output on an egress switch, but

for this to take place existing connections may need to be rearranged in the center stage.

If we make $m = n$ then we have $2rn^2 + nr^2$ switches or $2Nn + Nr$, which provides a lower switching complexity than N^2.

Multi-stage Clos Network

Substitute each of the $r \times r$ crossbars in the center stage with a Clos network, which increases the total stages from 3 to 5.

This reduces the switching complexity of the center stage.

Further substitution can be undertaken to created Clos networks with 7, 9, 11 stages, etc.

Beneš Switching Network

A rearrangeably nonblocking Clos network with $m = n = 2$ is generally called a Beneš Network. The number of inputs and outputs is $N = r \times n = 2r$, with $2\log_2 N - 1$ stages, each containing $\frac{N}{2}$ switches, totalling $N \log_2 N - \frac{N}{2}$ switches. An 8×8 Beneš Network is shown below.

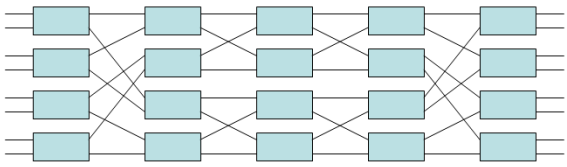


Figure 56: A 8×8 Beneš Switching Network.

- Here $r = 4$. Identify the two $r \times r$ "switches" in the middle, each consisting of a 3-stage Clos network.

The minimum latency for a Beneš network is again $O(1)$, i.e. any 1-to-1 mapping of inputs to outputs is still possible.

Omega Switching Network

An *Omega* switching network has complexity $O\big(n \log n\big)$ with n inputs and n outputs. In general, an n-input Omega network requires $\log_2 n$ stages of 2×2 switches. Each stage requires $n/2$ switch modules. In total, the network uses $(n/2) \log_2 n$ switches.

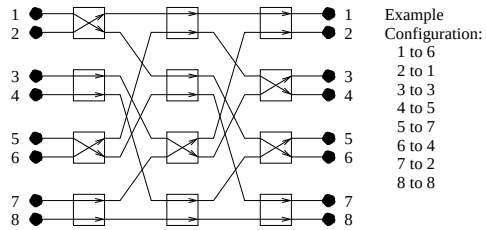


Figure 57: An 8×8 Omega switching network.

Uses roughly half as many 2×2 switches as a Benes network.

minimum latency increased to $O(\log n)$, i.e. some 1-to-1 I/O mappings require up to $\log n$ phases to be realized. Consider $1 \to 1$ and $5 \to 2$.

Baseline Switching Network

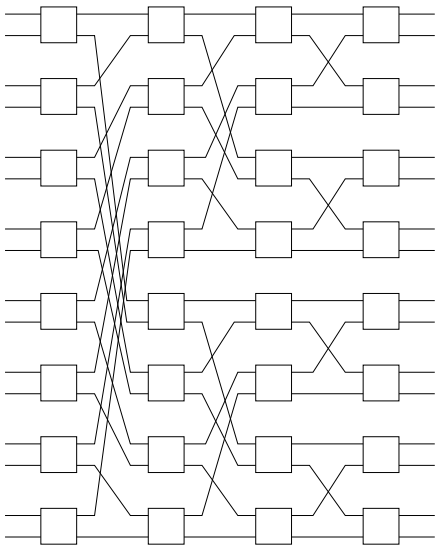


Figure 58: A 16×16 Baseline Switching Network.

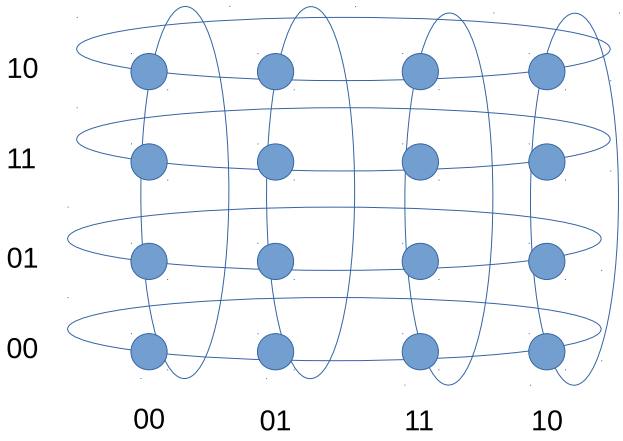
Embedding mesh topologies in a hypercube

Any mesh whose size is a power of two (flat, torus or with partial wrap-around) can be embedded in a hypercube. That is, it can be formed by "disabling" some links in the hypercube.

- The hypercube is $\{0,1\}^d$. Each node's address is the coordinates in Euclidean space, a d-bit binary number
- The size of each dimension of the mesh is a power of two.
- The coordinates of each node in a k-dimensional mesh are a k-tuple, each of $\log_2(d_i)$ bits, where d_i is the size in dimension i.
- Claim: If we partition the d bits of the hypercube into k groups, of sizes $d_1, \ldots, d_k$, then we can choose a numbering scheme for each coordinate i such that all of the links needed to make up the mesh are present in the hypercube.
  - That is, the mesh has been *embedded* into the hypercube

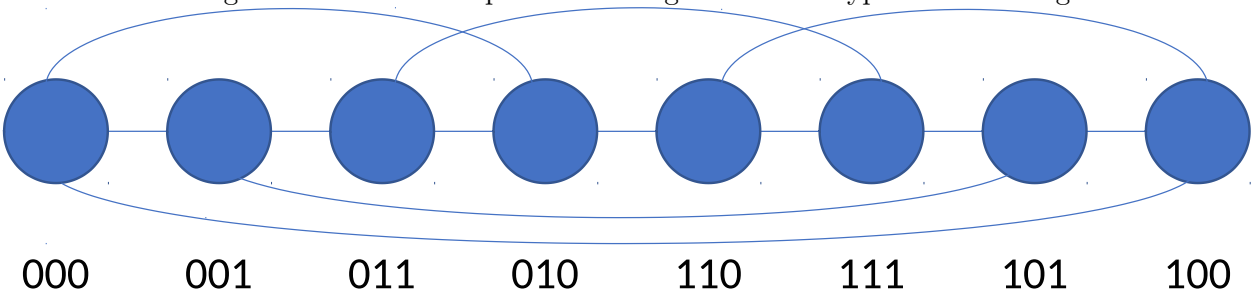
Trivial embeddings

- A 2×2 grid—or $2 \times 2 \times 2 \times ... \times 2$— is already a hypercube
- A $4 \times 4 \times \cdots \times 4$ grid is exactly a hypercube if the numbering is changed slightly



Embeddings by omitting edges

A mesh with side lengths of 8 or more requiers some edges from the hypercube to be ignored.



Similarly, linear topologies can be embedded in meshes. The 16-node linear network in an earlier slide can be seen to be part of a 4×4 mesh

These embeddings require fiddling with the order of bits, so that only one bit changes between neighbours. Is that always possible? [fragile]

Gray code

- Gray code is a reordering of d-bit integers, for any d, so that only one bit changes at a time,
- Which bit changes?
  - 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 3
  - To find the pattern, consider the most significant bit that would change with "normal" counting.

```
int gray = 0;
int i;

for (i = 0; i < d; i++) {
    printf  ("Gray code %d\n", gray);
    int normal_change = i ^ ((i+1) % d);
```

```
    int bit = normal_change ^ (normal_change >> 1);
    gray ^= bit;
}
```

- Only one bit still changes for wrapping from the last value back to 0.
- However, the `(i+1)` in the code must change to `((i+1) % d)`

There are *many* different switching networks with a variety of different properties. We don't have time to study them in detail in this subject.

Systolic Algorithms

Systolic Algorithms

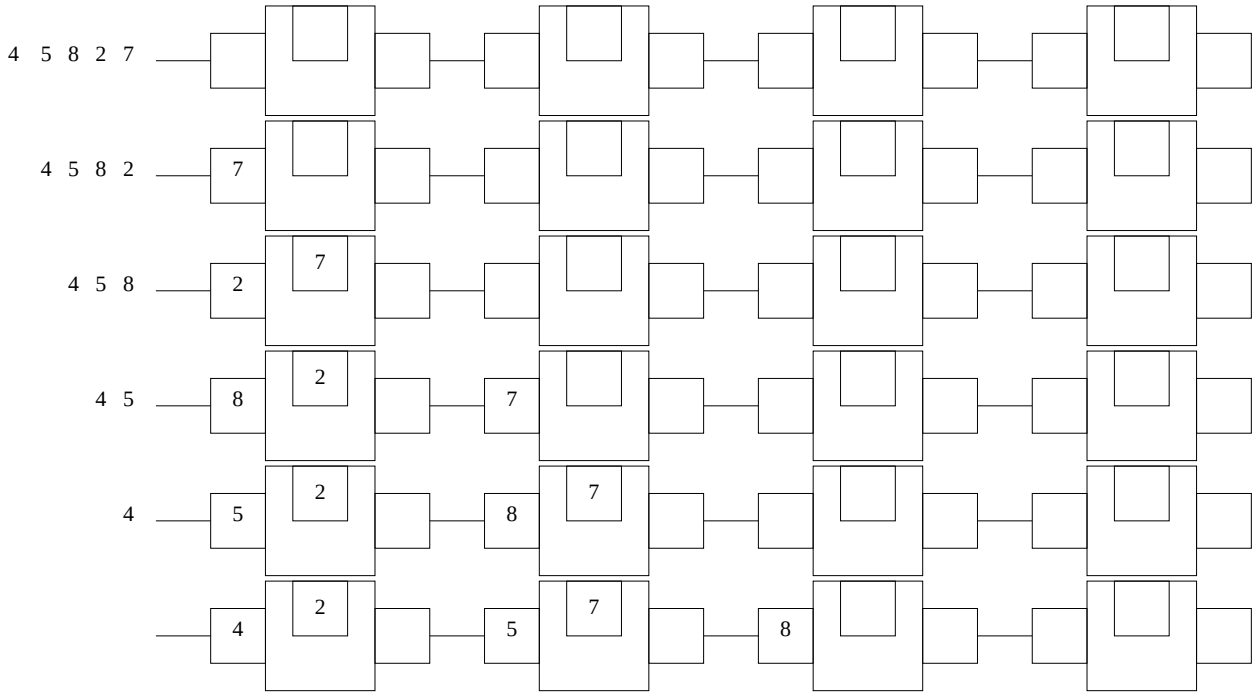
Systolic algorithms consider the flow of data through an array of processing elements. The processing elements themselves do not have access to any global memory, but rather just to values from adjacent processing elements.

From an implementation perspective, each processing element consists of input buffers, output buffers and local memory. Flow of data and computation takes place in two phases:

1. Processing elements read from their input buffers and local memory, undertake some computation and write results to their output buffers.
2. Output buffers are copied across to the input buffers of adjacent processing elements.

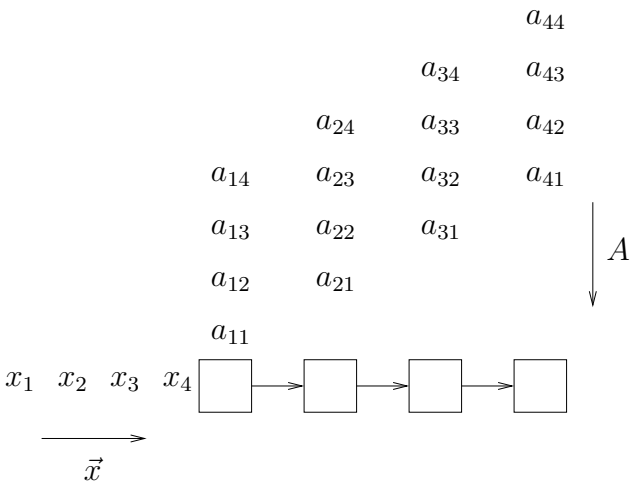
Sorting on a systolic linear array

Only the computation phase is shown below. A speedup of $\Theta(\log N)$ is achieved using an N-cell linear array.



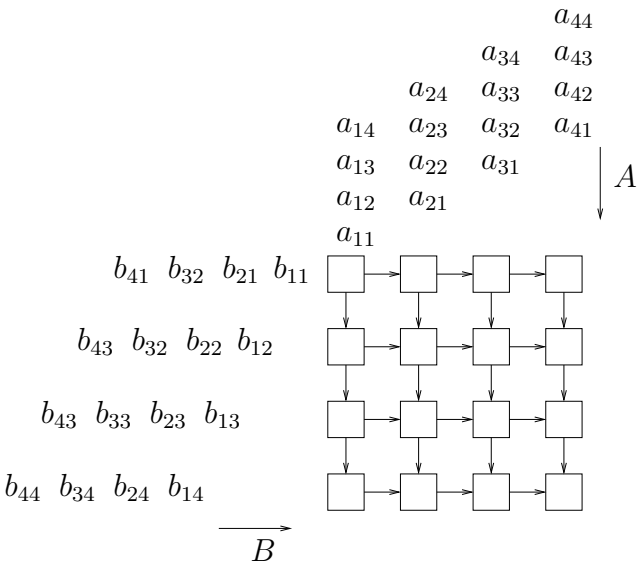
Matrix-vector multiplication

Computing a matrix-vector product $\vec{y} = A\,\vec{x}$ in seven steps on a 4-cell linear array. The values of $\vec{x}$ move rightward after each multiply/add step. A speedup of $\Theta(N)$ is achieved using an N-cell linear array.



Matrix-matrix multiplication

Computing a matrix-matrix product $C = A \times B$ in 10 steps on a 4×4 mesh. A speedup of $\Theta(N^2)$ is achieved using an $N \times N$ mesh.

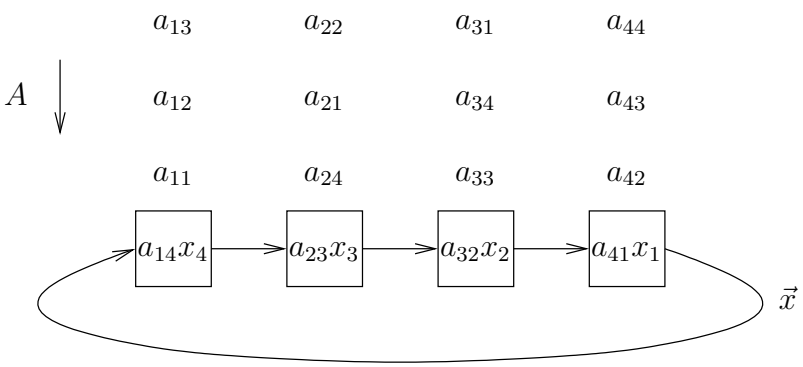


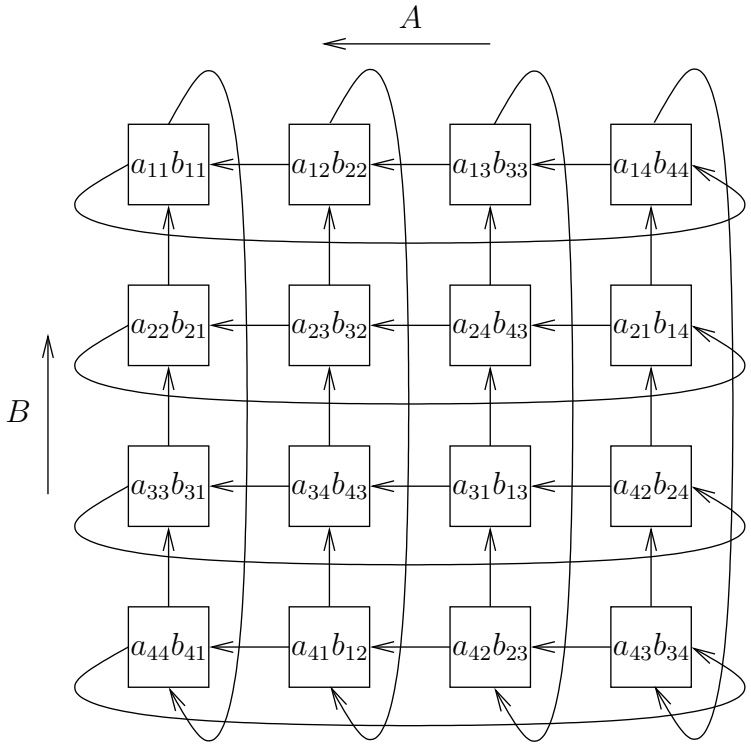
Matrix-vector multiplication

The first step in computing a matrix-vector product $\vec{y} = A\,\vec{x}$ in four steps on a 4-cell ring. The values of $\vec{x}$ move rightward after each multiply/add step.

Matrix-matrix multiplication

Computing a matrix-matrix product $C = A \times B$ in 4 steps on a 4×4 torus. A speedup of $\Theta(N^2)$ is achieved using an $N \times N$ mesh.

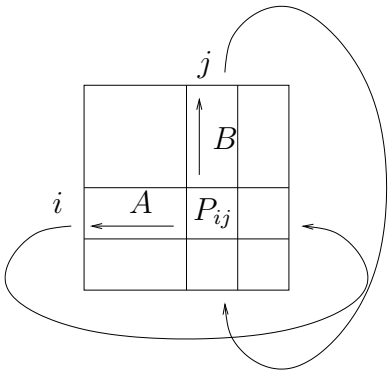

a_{13} a_{22} a_{31} a_{44}
A
a_{12} a_{21} a_{34} a_{43}
a_{11} a_{24} a_{33} a_{42}
$a_{14}x_4$ $a_{23}x_3$ $a_{32}x_2$ $a_{41}x_1$
$\vec{x}$


A
$a_{11}b_{11}$ $a_{12}b_{22}$ $a_{13}b_{33}$ $a_{14}b_{44}$
$a_{22}b_{21}$ $a_{23}b_{32}$ $a_{24}b_{43}$ $a_{21}b_{14}$
B
$a_{33}b_{31}$ $a_{34}b_{43}$ $a_{31}b_{13}$ $a_{42}b_{24}$
$a_{44}b_{41}$ $a_{41}b_{12}$ $a_{42}b_{23}$ $a_{43}b_{34}$

Cannon's algorithm

Cannon's algorithm (1969) describes the matrix-matrix multiplication on a torus.

Cannon's algorithm uses a $N \times N$ torus of processors. Processor (i, j) at location (i, j) initially begins with elements or submatrices a_{ij} and b_{ij}, for $i, j = 1, 2, \ldots, N$. As the algorithm progresses, the submatrices are passed left and upwards.



1. Initially P_{ij} begins with a_{ij} and b_{ij}.
2. Elements are moved from their initial positions to align them so that the correct submatrices are multiplied with one another. Note that submatrices on the diagonal don't actually require alignment. Alignment is done by shifting the i-th row of A $i - 1$ positions left and the j-th column of B $j - 1$ positions up.
3. Each processor, P_{ij} multiplies its current submatrices and adds to a cumulative sum.
4. The submatrices are shifted left and upwards.
5. The above two steps are repeated through the remaining submatrices.

Odd-even Transposition Sort

Recall the operation of a bubble sort algorithm. For n elements, a first phase iterates through the elements, comparing and exchanging neighbors, with $n - 1$ operations. After the first phase the largest element will have "bubbled" to the end of the array.

Since the largest element appears at the end of the array after the first phase, the second phase need only iterate $n - 2$ times. The bubble sort however takes $O(n^2)$ sequential steps.

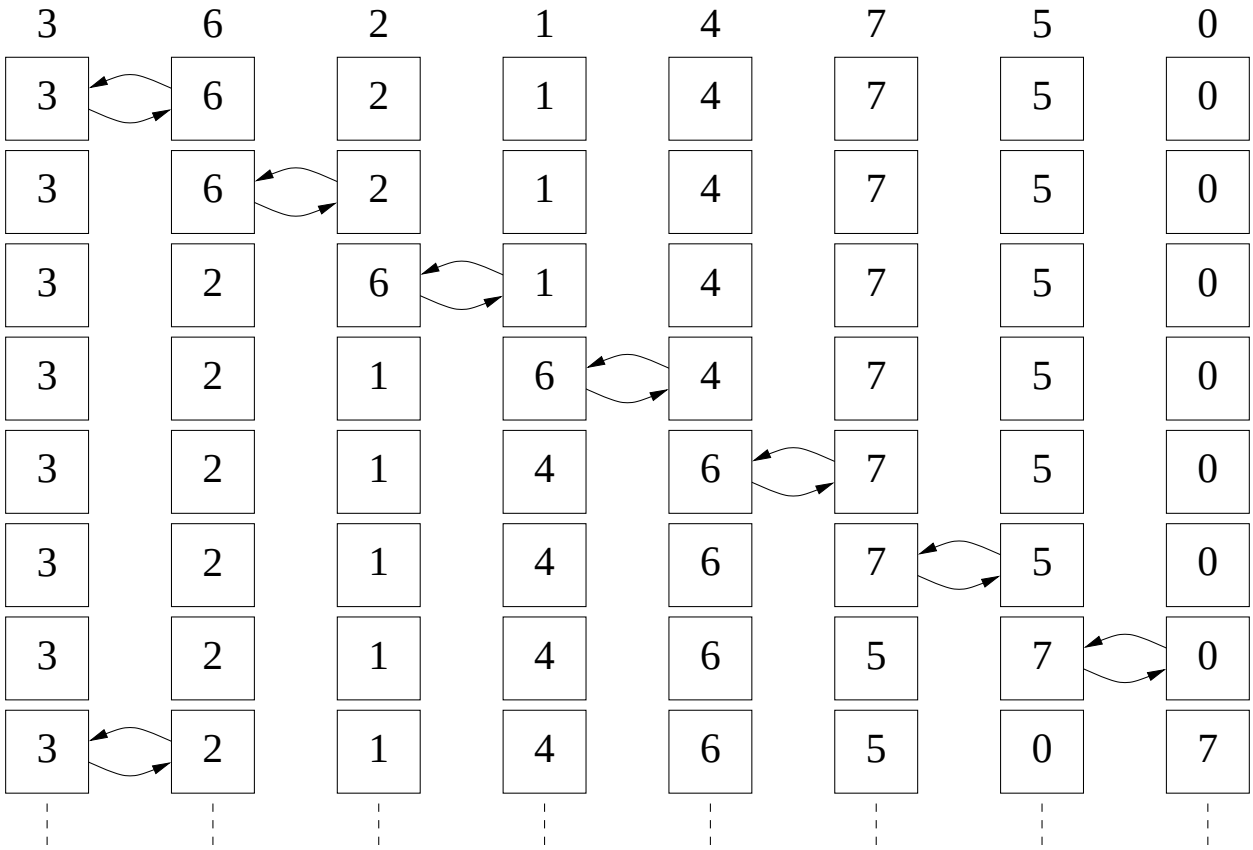
The odd-even transposition sort makes use of a pipelining technique to ultimately run many phases of the bubble sort in parallel.

The second phase of the bubble sort algorithm can begin after the second iteration of the first phase.

The third phase can begin after the fourth iteration, the fourth phase after the sixth iteration and so on.

In effect each parallel computational step can pair off either the odd or even neighboring pairs.

Assuming n is an even number.

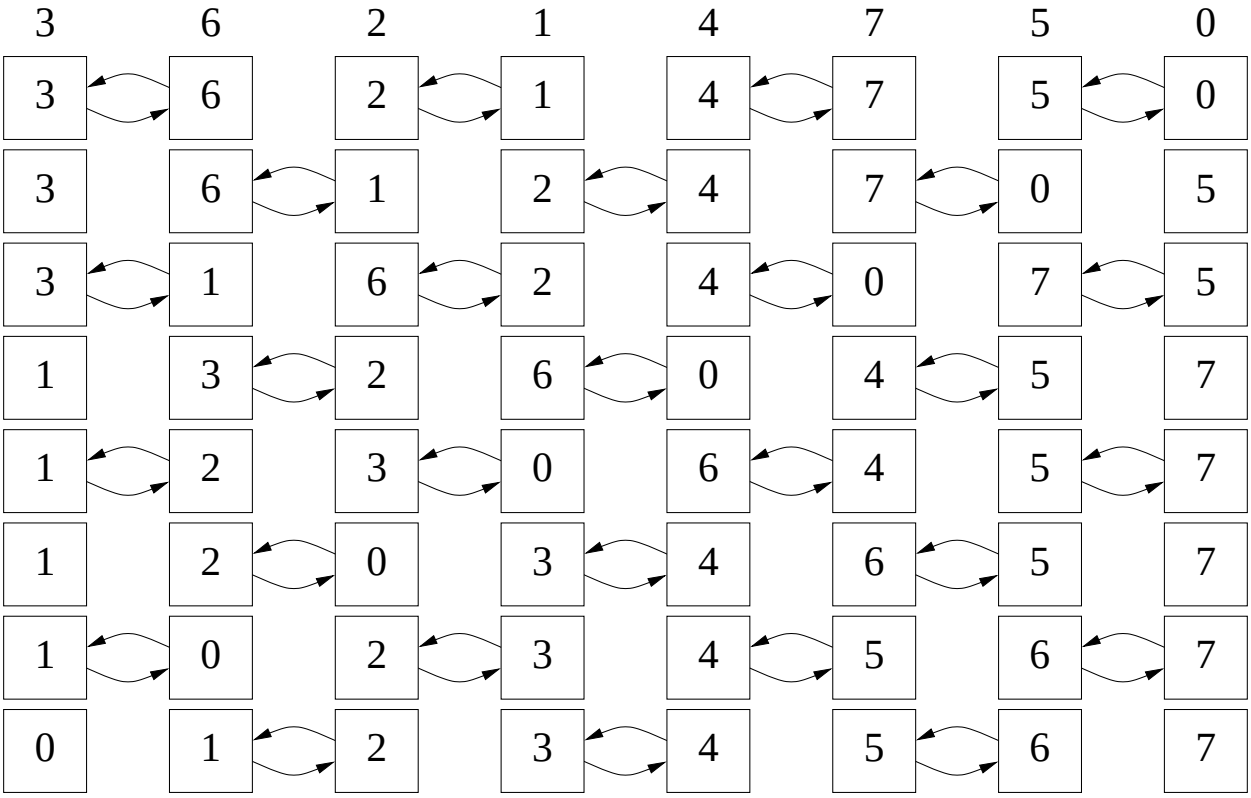


Processor P_i, i is odd:

```
send(A, Pi-1);
recv(B, Pi-1);
if(A<B) A=B;
if(i<=n-3){
  send(A, Pi+1);
  recv(B, Pi+1);
  if(A>B) A=B;}
```

Processor P_i, i is even:

```
recv(A, Pi+1);
send(B, Pi+1);
if(A<B) B=A;
if(i>=2){
  recv(A,Pi-1);
  send(B,Pi-1);
  if(A>B) B=A;}
```



Shear sort

The Shear sort is for mesh architectures that use message passing. The lower bound for a mesh architecture sort is $O(\sqrt{n})$ since it takes at most $2(\sqrt{n}-1)$ steps in the worst case to move a number from one node to another.

The Shear sort uses $\log_2 n + 1$ phases. In odd phases (1, 3, 5 ...) the following is done:

- even rows – the row is sorted with the smallest number placed at the right.
- odd rows – the row is sorted with the smallest number placed at the left.

In even phases (2, 4, 6, ...) the following is done:

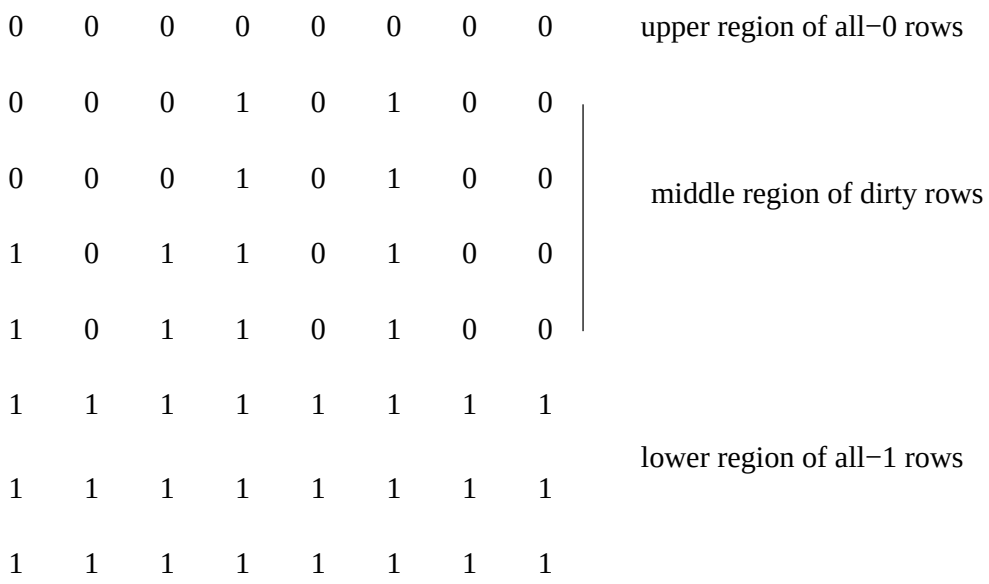
- each column of number is sorted independently with the smallest number placed at the top.

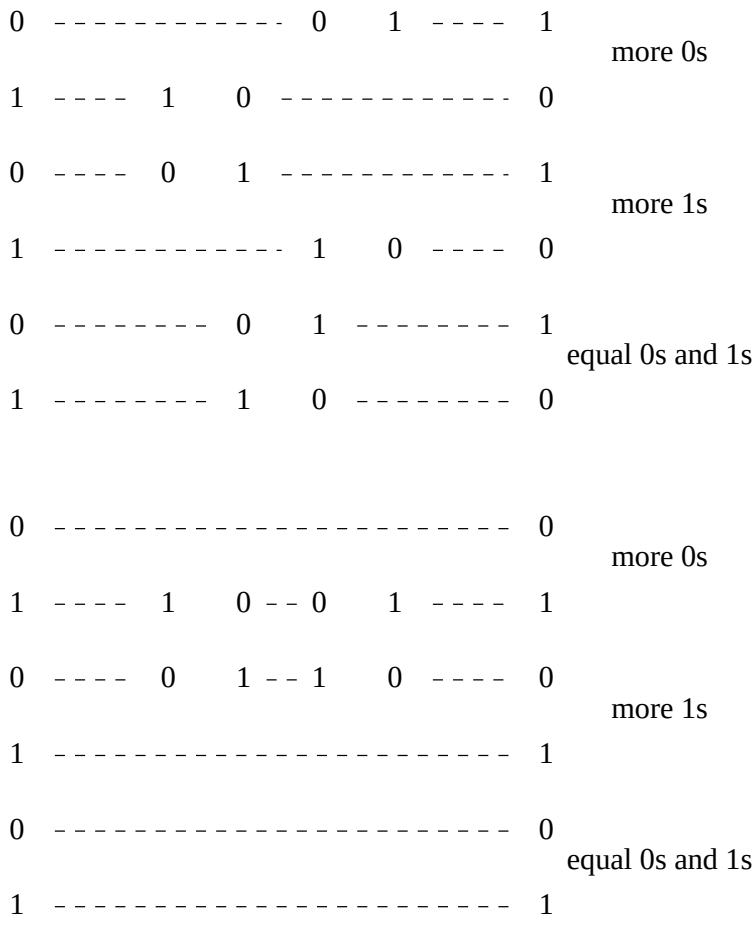
Sorting of rows and columns requires $\sqrt{n}$ time to complete using for example the odd-even transposition sort. The total time is then $\sqrt{n}(\log_2 n + 1)$.

There are other algorithms that reach the lower bound for meshes.

Shear Sort Intuitive Proof

Since at least one row in each pair becomes all-0 or all-1 and is moved out of the middle region after sorting the rows and columns, the middle region decreases in size by at least one-half for each pair of phases. Hence after $2 \log \sqrt{N} = \log N$ phases, the numbers are sorted, except for one row, and the algorithm is concluded by sorting this row in the last phase.


0 0 0 0 0 0 0 0
upper region of all−0 rows
0 0 0 1 0 1 0 0
0 0 0 1 0 1 0 0
middle region of dirty rows
1 0 1 1 0 1 0 0
1 0 1 1 0 1 0 0
1 1 1 1 1 1 1 1
lower region of all−1 rows
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1


0 0 1 1
more 0s
1 1 0 0
0 0 1 1
more 1s
1 1 0 0
0 0 1 1
equal 0s and 1s
1 1 0 0
0 0
more 0s
1 1 0 0 1 1
0 1 1 0 0
more 1s
1 1
0 0
equal 0s and 1s
1 1

Hypercube and Embeddings

Broadcasting in networks

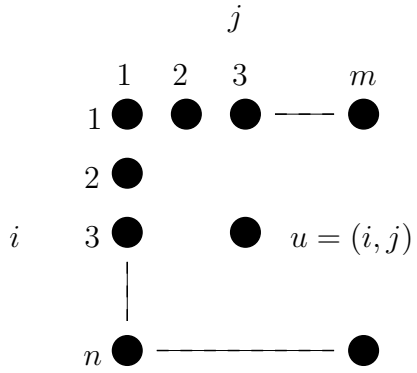
Consider a cycle with n nodes, numbered consecutively in the clockwise direction from 0 to $(n-1)$. A broadcast from node i takes place in $\lfloor \frac{n}{2} \rfloor$ rounds and is described by the following broadcast algorithm, where X is being broadcast from node i:

```
broadcast(i,X)
for j = 0 to ⌊n/2⌋−1
  do task 1 and task 2 in parallel
    task 1: node (i+j) mod n sends X to (i+j+1) mod n
    task 2: node (n+i−j) mod n sends X to (n+i−j−1) mod n
  done
done
```

Show a broadcast algorithm, broadcast(u,X) for the following networks:

1. mesh with n rows and m columns, where $u = (i, j)$, $1 \leq i < n$, $1 \leq j < m$, that completes in $O(n + m)$ rounds.
2. hypercube with $n = 2^t$ nodes, $t \geq 0$, where nodes are numbered using t bit binary strings and two nodes are connected iff their numbers differ in exactly 1 bit position, that completes in $O(t)$ rounds.

Mesh Broadcast

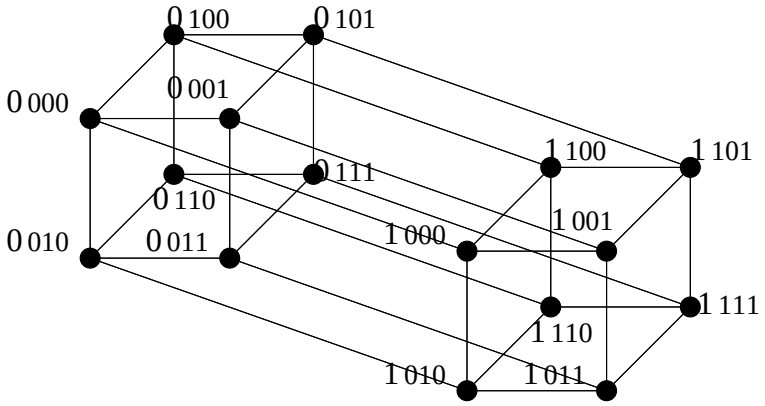



```
for k = 1 to m−1 do
  do task 1 and 2 in parallel
    task 1: if (j−k+1>1) then
      node (i,j−k+1) sends X to node (i,j−k)
    task 2: if (j+k−1<m) then
```

```
        node (i, j+k-1) sends X to node (i, j+k)
  done
done
for k = 1 to n-1 do
  for l = 1 to m do in parallel
    do task 1 and 2 in parallel
      task 1: if (i-k+1 > 1) then
        node (i-k+1, l) sends X to node (i-k, l)
      task 2: if (i+k-1 < n) then
        node (i+k-1, l) sends X to node (i+k, l)
    done
  done
done
```

The first part of the mesh algorithm takes $\Theta(m)$ and the second part takes $\Theta(n)$ so the total is $\Theta(m+n)$.

Hypercube Numbering



Hypercube Broadcast

$$u = a_{t-1}a_{t-2}\cdots a_2a_1a_0$$

$a_{t-1}a_{t-2}\cdots a_2\bar{a_1}a_0$ $\quad$ $a_{t-1}a_{t-2}\cdots a_2a_1a_0$

$\Leftarrow$ round 1 $\quad$ $\Downarrow$ round 0

$a_{t-1}a_{t-2}\cdots a_2\bar{a_1}\bar{a_0}$ $\quad$ $a_{t-1}a_{t-2}\cdots a_2a_1\bar{a_0}$

$\Downarrow$ round 2

$a_{t-1}a_{t-2}\cdots \bar{a_2}\bar{a_1}a_0$ $\quad$ $a_{t-1}a_{t-2}\cdots \bar{a_2}a_1a_0$

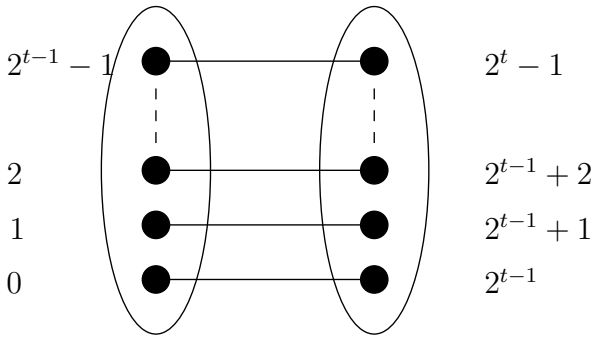
$a_{t-1}a_{t-2}\cdots \bar{a_2}\bar{a_1}\bar{a_0}$ $\quad$ $a_{t-1}a_{t-2}\cdots \bar{a_2}a_1\bar{a_0}$

```
broadcast(u,X)
for i = 0 to t-1
  for j = to 2^i - 1 do in parallel
    node u ⊕ j sends X to node u ⊕ j ⊕ 2^i
  done
done
```

Question: Consider a hypercube with $n = 2^t$ nodes and a number in the local memory of each node.

1. Write a parallel algorithm that computes the sum of all numbers in $O(t)$ rounds. The sum should be stored in node 0 by the end of the algorithm.
2. Write a parallel algorithm that computes the prefix sum in $O(t^2)$ rounds.

Answer: Consider the hypercube with nodes as numbered below.



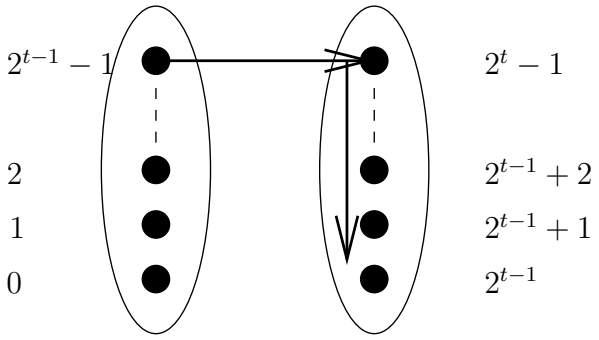
Let a_p be the number stored at processor p.

```
for i = t-1 down to 0
  for p = 0 to 2^i - 1 do in parallel
    processor p + 2^i sends a_{p+2^i} to processor p
    processor p does a_p ← a_p + a_{p+2^i}
  done
done
```

The loop takes $O(t)$ time.

For the prefix sum we consider the upper-lower prefix sum algorithm. We also assume that we can use a $Broadcast_t(u, X)$ algorithm that takes t steps to broadcast the value X from node u in a hypercube of dimension t.

```
PrefixSum_t(0,1,...,2^t - 1)
if (t > 1) then
  do task 1 and task 2 in parallel
    task 1: PrefixSum_{t-1}(0,1,...,2^{t-1} - 1)
    task 2: PrefixSum_{t-1}(2^{t-1}, 2^{t-1} + 1,...,2^t - 1)
  done
  node 2^{t-1} - 1 sends its value of prefix sum to node 2^t - 1
```




```
  Broadcast_{t-1}(2^t - 1, sum)
  all nodes in 2^{t-1}, ..., 2^t - 1 add sum to their current sum
else
  node 0 sends its value to node 1
  node 1 adds the value to its own value
done
```

The depth of recursion is $O(t)$. Each step of the recursion requires a broadcast that takes place in $O(t)$ time. Therefore the total runtime is $O(t^2)$.

An alternative solution is:

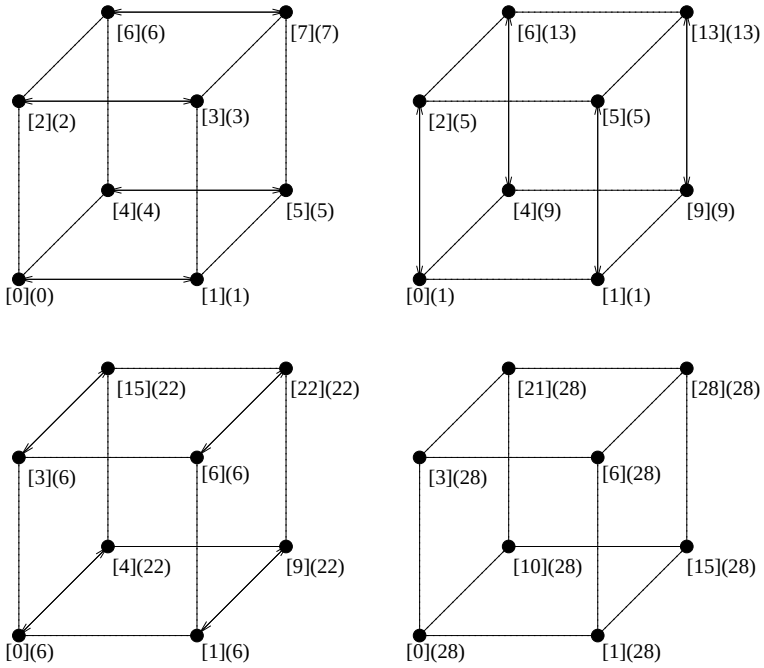
```
for i = 0 to t - 1
  for j = 0 to 2^{t-i-1} - 1 do in parallel
    processor p = (2^i - 1) + j 2^{i+1} broadcasts
      to processors p + 1, p + 2, ..., p + 2^i
    processors p + 1, p + 2, ..., p + 2^i add on the
      value to their prefix sum
  done
done
```

Optimal Hypercube Prefix Sum

An optimal hypercube prefix sum runs in $O(t)$ steps. Assume a_i holds the initial data element for processor i.

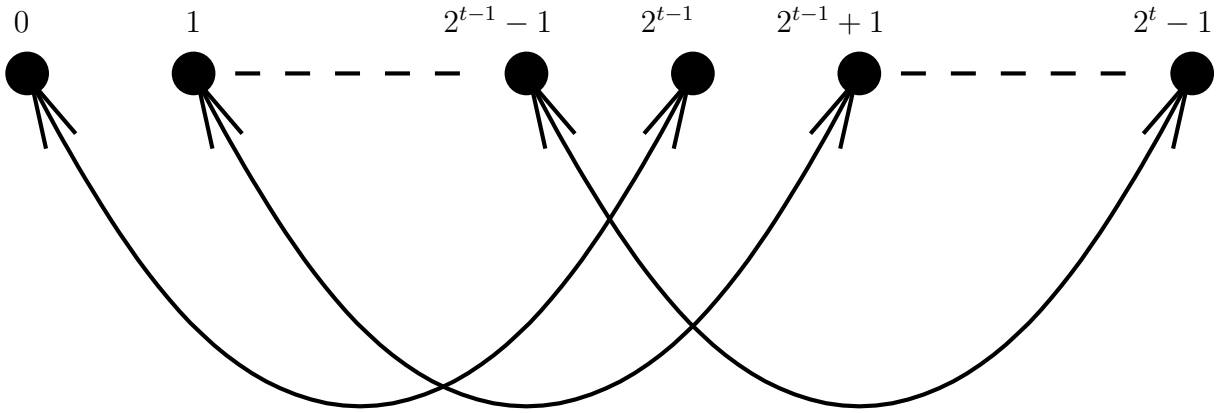
```
PrefixSum_t(0, 1, ..., 2^t - 1)
for i = 0 to 2^t - 1 do in parallel
    msg_i ← a_i
    res_i ← a_i
for k = 0, 1, ..., t - 1 do
    for i = 0 to 2^t - 1 do in parallel
        partner ← i ⊕ 2^k
        Send msg_i to partner and receive msg_partner from partner
        msg_i ← msg_i + a_i
        if partner < i then
            res_i ← res_i + a_i
```

```
        end
    end
end
```



Hypercube bitonic mergesort

The hypercube naturally supports the bitonic mergesort operation, where the arrows represent connections in the hypercube that are used for compare and exchange.



For the hypercube of 2^3 nodes there are 3 phases, where phase $i = 1, 2, 3$ has i steps. In the i-th step of each phase, there are $\frac{2^3}{2^i}$ bitonic merge operations in parallel. The direction of sorting for each of these merge operations alternates up and down.

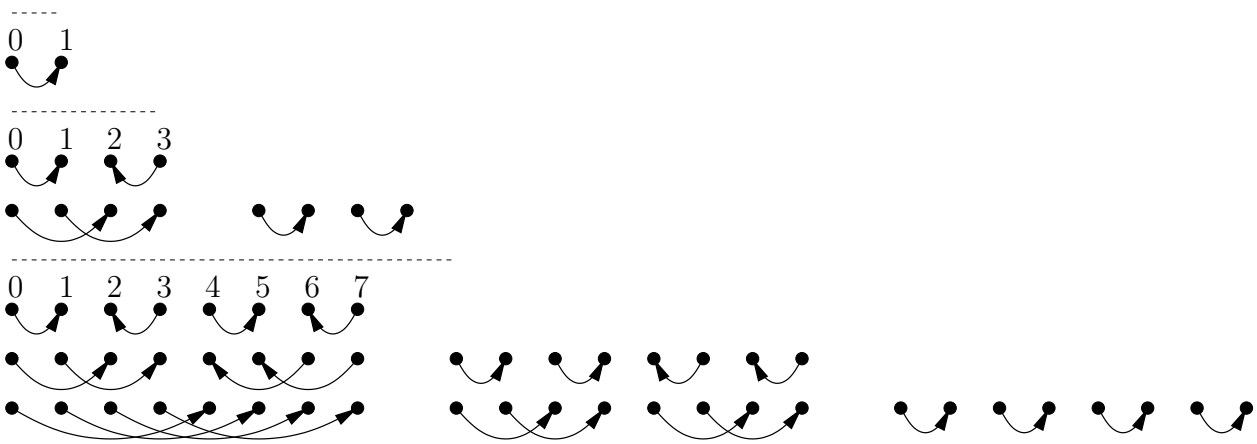


Figure 59: Examples for three hypercubes, of size 2, 4 and 8 nodes.

$A \to B$ means A compares value with B and the lower value is stored in A with the higher value stored in B.

```
Algorithm: bitonic(lower, upper, direction)
mid = ⌊(lower+upper)/2⌋
for i = lower to mid
do in parallel
    if direction = up
        P_i → P_{i+mid+1}
    else
        P_i ← P_{i+mid+1}
done

if upper − lower > 1
    do in parallel
        bitonic(lower, mid, direction)
        bitonic(mid + 1, upper, direction)
done
```

The hypercube mergesort algorithm on 2^t nodes has t phases where phase $i = 1, 2, \ldots, t$ calls 2^{t-i} bitonic merge sorts in parallel and each merge sort takes i steps. The direction of the bitonic merge sorts are alternating. There are $\Theta(t^2)$ operations in total.

```
Algorithm: mergesort(0, 1, 2, ..., 2^t − 1)
for i = 1 to t
    for j = 0 to 2^t − 1 step 2^i
    do in parallel
        bitonic(j, j + 2^i − 1, j mod 2^{i+1} == 0 ? up : down)
    done
done
```

Embeddings into the hypercube

A graph embedding shows how one graph can be contained within another graph. Graph embeddings allow algorithms designed for a given graph to be executed on another graph without changing the algorithm.

A hypercube on $N = 2^t$ nodes contains every N-node array as a subgraph.

That means any such array (in any dimension, i.e. 2 dimensional, 3 dimensional, etc) can be embedded into the hypercube.

This remains true if wraparound edges are included on the array (to create a torus, e.g. in 2 dimensions).

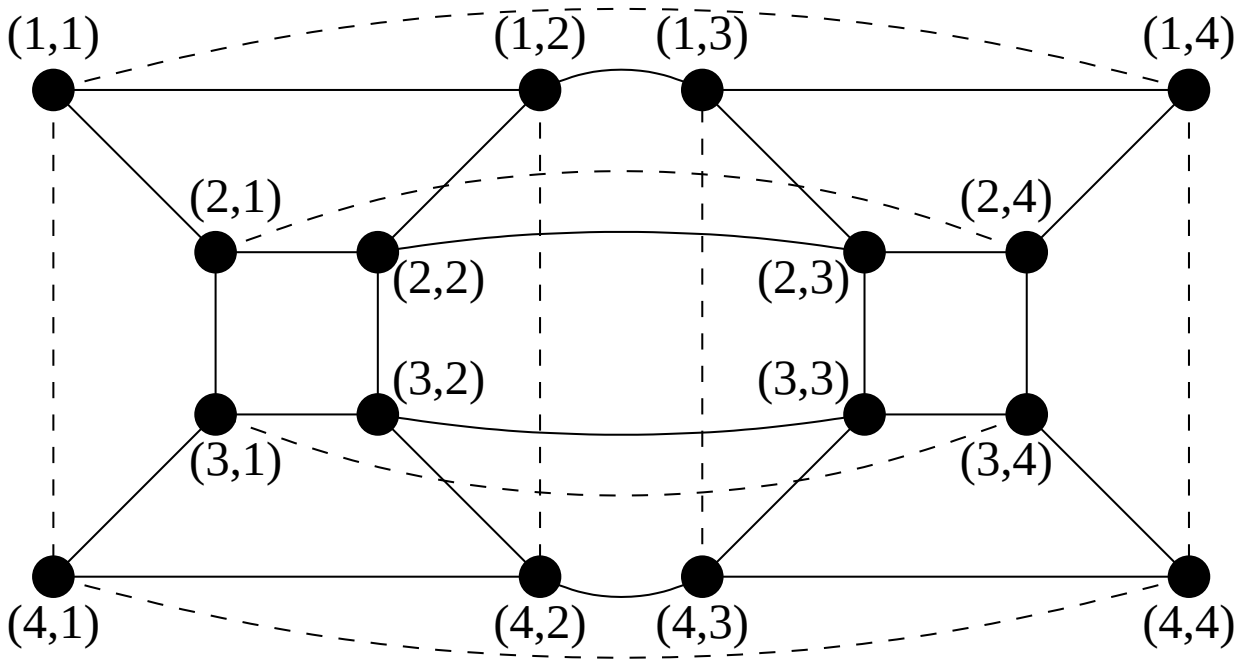


Figure 60: Embedding of a 4×4 mesh in a 16 node hypercube. In this example the dashed lines would form the wrap around edges of a torus.

First we consider a simpler example.

Lemma The N-node hypercube contains an N-node linear array (with wraparound) as a subgraph for $N \geq 4$.

Proof The proof is by induction. It is true for $N = 4$ by inspection. Consider two hypercubes of size $N/2$ and assume they each contain a cycle (linear array with wraparound) of length $N/2$. A cycle can be formed in the hypercube of N nodes by removing an edge from each cycle and connecting the cycles to form one cycle of length N.

Gray codes

A Hamiltonian cycle in the hypercube is defined by Gray codes, which are a sequence of bit string codes where each subsequent code differs by only one bit.

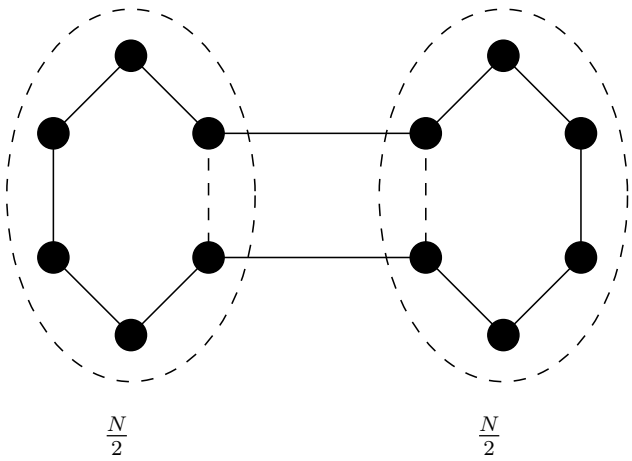


Figure 61: Forming a cycle of length N from two smaller cycles of length $N/2$.

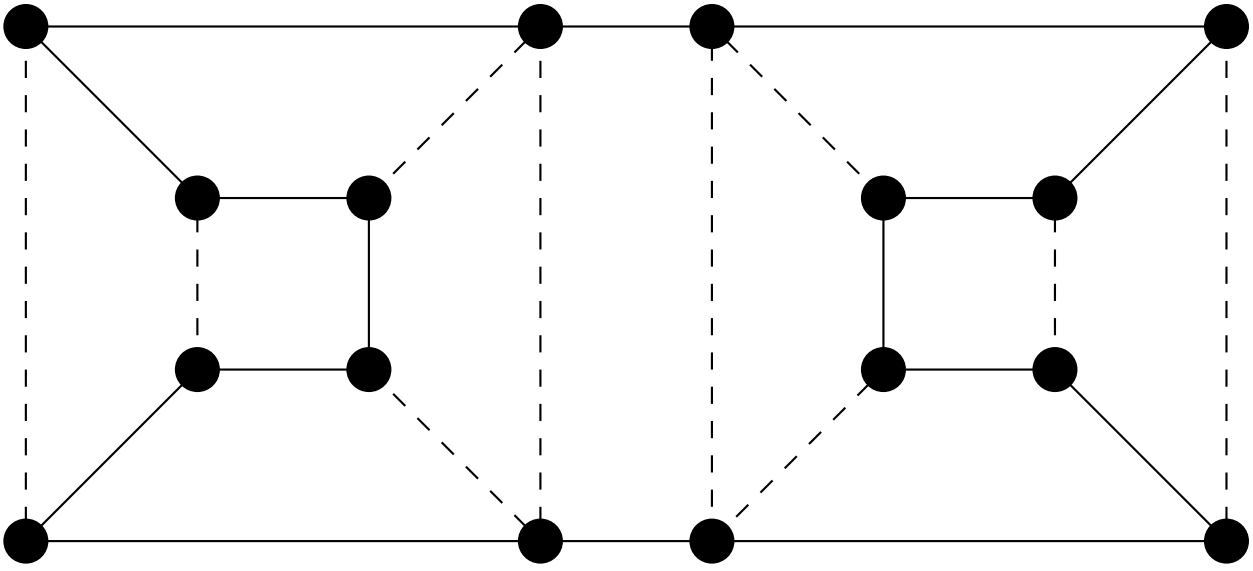


Figure 62: An example Hamiltonian cycle in a 16 node hypercube.

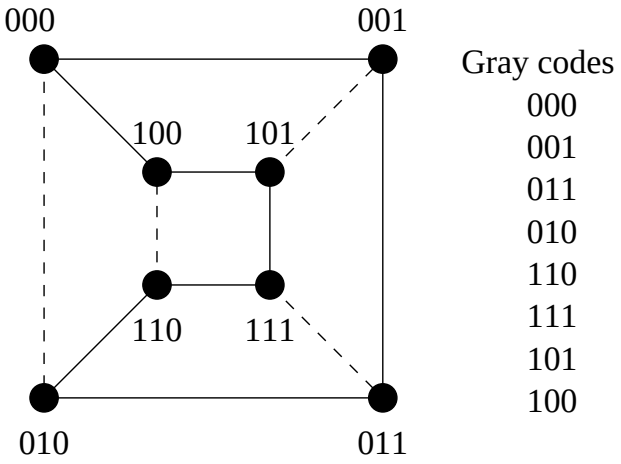


Figure 63: An example set of gray codes.

Cross product of graphs

To consider higher dimensional arrays embedding in a hypercube we first need to define the cross product of graphs.

We say that $G = (V, E)$ is the cross product of graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$,...,$G_k = (V_k, E_k)$ if:

$$V = \{(v_1, v_2, \ldots, v_k) \mid v_i \in V_i \text{ for } 1 \leq i \leq k\}$$

and

$$E = \Big\{ \{(u_1, u_2, \ldots, u_k), (v_1, v_2, \ldots, v_k)\} \mid \exists j \text{ such that } (u_j, v_j) \in E_j \text{ and } u_i = v_i \text{ for all } i \neq j \Big\}.$$

Notationally:

$$G = G_1 \otimes G_2 \otimes \cdots \otimes G_k.$$

Lemma For any $k \geq 1$ and $t = t_1 + t_2 + \cdots + t_k$, the t-dimensional hypercube on 2^t nodes, H_t, can be expressed as the cross product:

$$H_t = H_{t_1} \otimes H_{t_2} \otimes \cdots \otimes H_{t_k}$$

where H_{t_i} denotes the t_i-dimensional hypercube for $1 \leq i \leq k$.

Proof The t-dimensional hypercube H_t is a $\overbrace{2 \times 2 \times \cdots \times 2}^{t}$ array. Hence, H_t is the cross product

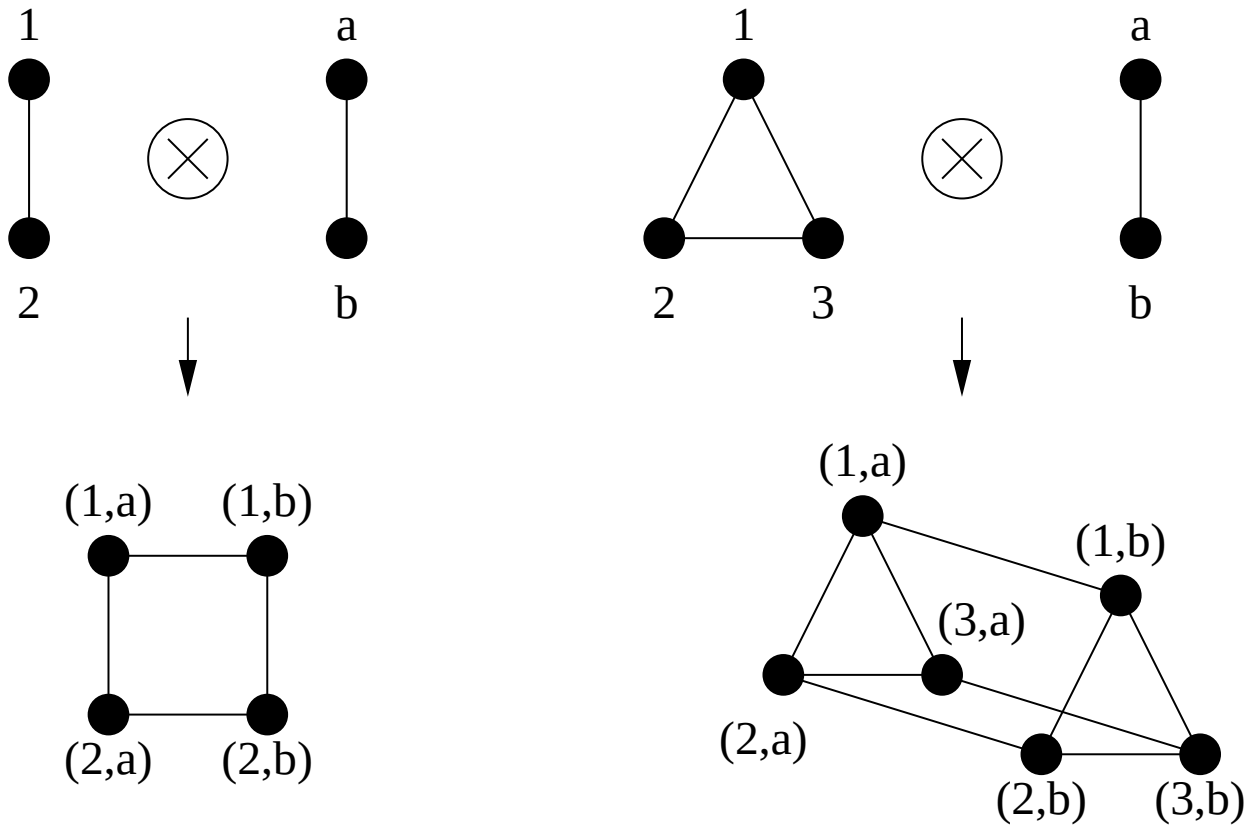


Figure 64: Simple examples of graph cross products.

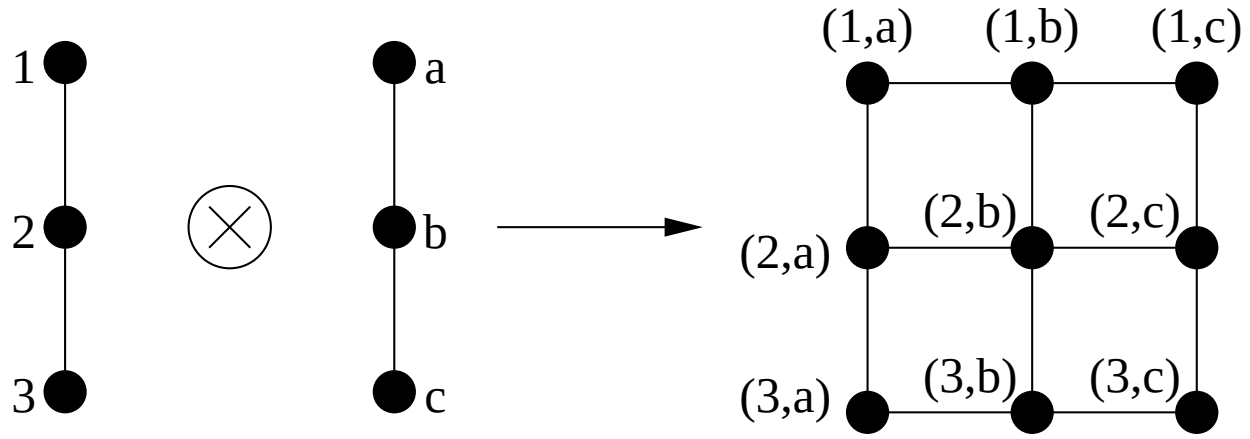


Figure 65: The cross product of linear arrays produces a multidimensional array.

of t 2-node linear arrays H_1. Since H_{t_i} is the cross product of t_i 2-node linear arrays,

$$\begin{aligned} H_t &= \overbrace{H_1 \otimes H_1 \otimes \cdots \otimes H_1 \otimes H_1}^{t} \\ &= \overbrace{H_1 \otimes H_1}^{t_1} \otimes \cdots \otimes \overbrace{H_1 \otimes H_1}^{t_k} \\ &= H_{t_1} \otimes \cdots \otimes H_{t_k}. \end{aligned}$$

We know:

- a multidimensional array is the cross product of linear arrays,
- a hypercube is the cross product of smaller hypercubes,
- a linear array is a subgraph of a hypercube.

We need to first prove one more general purpose lemma before we can conclude that a multidimensional array is a subgraph of a hypercube.

Lemma If $G = G_1 \otimes G_2 \otimes \cdots \otimes G_k$ and $G' = G'_1 \otimes G'_2 \otimes \cdots G'_k$ for some $k \geq 1$, and G_i is a subgraph of G'_i for $1 \leq i \leq k$, then G is a subgraph of G'.

Proof We construct a one-to-one mapping of nodes in G to nodes in G' that maps edges to edges. For each i $(1 \leq i \leq k)$, let σ_i be a map of the nodes of G_i to the nodes of G'_i that preserves edges. Given any node $v = (v_1, v_2, \ldots, v_k)$ of G, define $\sigma \colon G \to G'$ by

$$\sigma(v) = (\sigma_1(v_1), \sigma_2(v_2), \ldots, \sigma_k(v_k)).$$

To see that σ preserves edges, we need to show that if (u, v) is an edge of G, then $(\sigma(u), \sigma(v))$ is an edge of G'. We can, since if (u, v) is an edge in G, then there is a j such that (u_j, v_j) is an edge of G_j and such that $u_i = v_i$ for all $i \neq j$. Hence, for the same j, $(\sigma_j(u_j), \sigma_j(v_j))$ is an edge of G'_j and $\sigma_i(u_i) = \sigma_i(v_i)$ for all $i \neq j$. Thus $(\sigma(u), \sigma(v))$ is an edge of G'.

We can now conclude:

- a $2^{t_1} \times 2^{t_2} \times \cdots \times 2^{t_k}$ array is a subgraph of the 2^t node hypercube when $t = t_1 + t_2 + \cdots + t_k$.
- any 2^t node array of any dimension is a subgraph of the 2^t node hypercube.
- any $M_1 \times M_2 \times \cdots \times M_k$ array is contained in an N node hypercube where

$$N = 2^{\lceil \log M_1 \rceil + \lceil \log M_2 \rceil + \cdots + \lceil \log M_k \rceil}.$$

Containment of Complete Binary Trees

The hypercube on N nodes does not provide an embedding of an $(N-1)$ node complete binary tree.

Consider the *parity* of a hypercube node, where it is odd parity if it has an odd number of 1's in its label, or even parity if it has an even number of 1's in its label. Then the hypercube has $N/2$ odd nodes and $N/2$ even nodes.

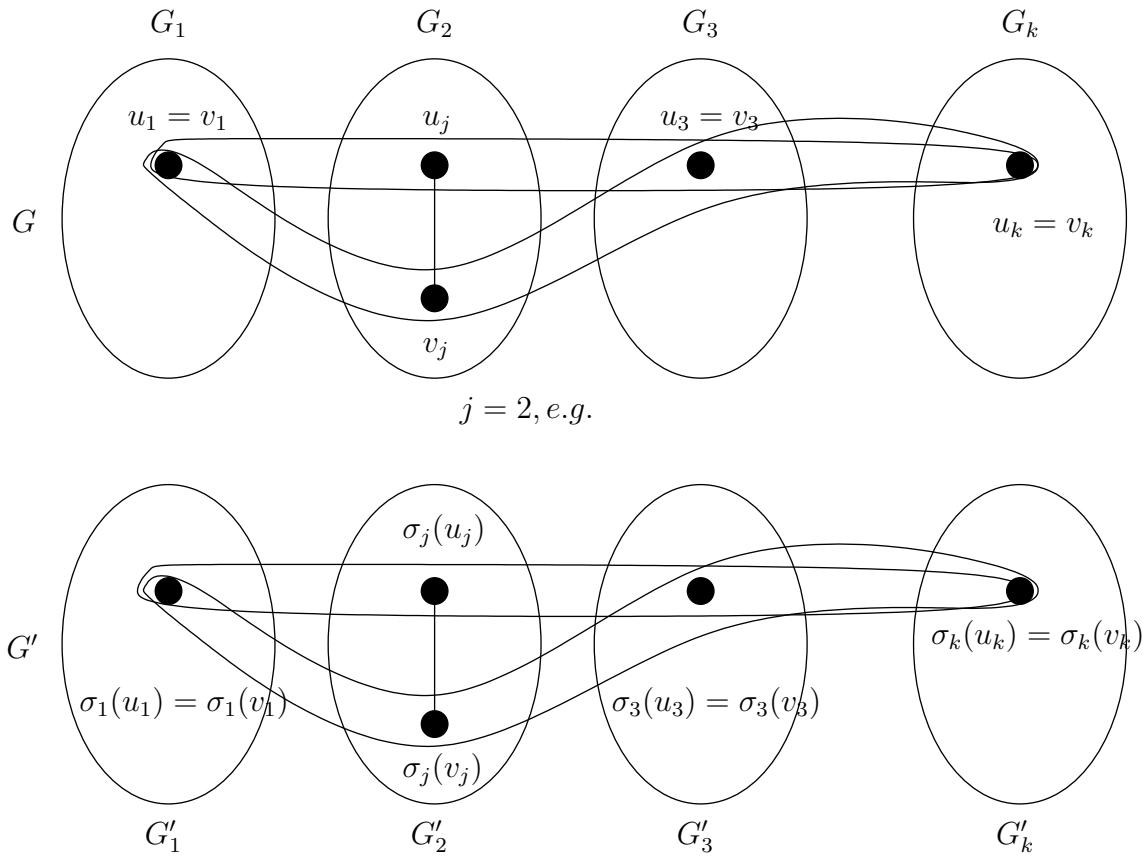


Figure 66: Figure for proof.

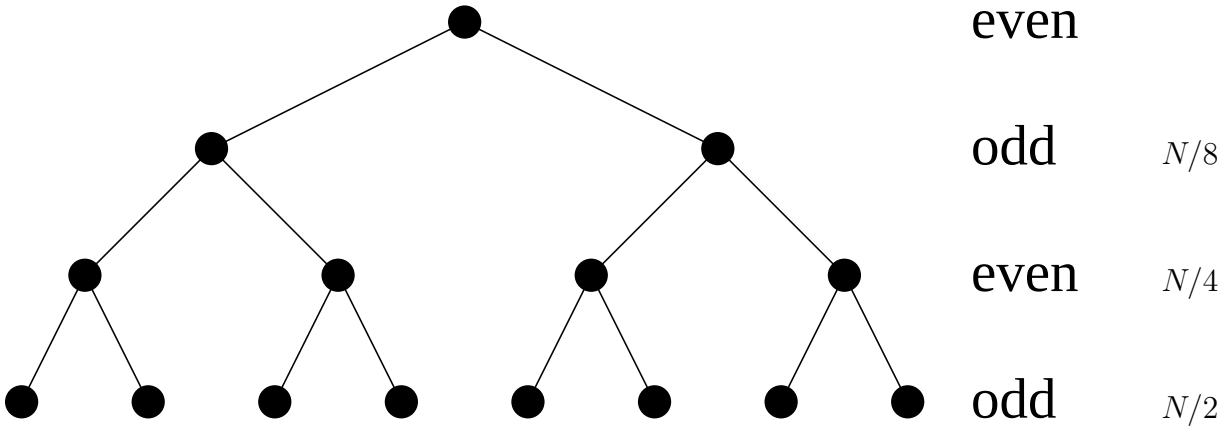


Figure 67: A complete binary tree of $N-1$ nodes cannot fit into a hypercube of N nodes.

Say the root node of the binary tree starts on an even node. Then the nodes in the next level must all be on odd nodes. Consider the $N/2$ leaves of the binary tree being odd. That would mean that there were $N/8$ internal nodes that are also odd, two levels above the leaves. This is at least $5\,N/8$ odd nodes, which is greater than $N/2$ odd nodes in the hypercube.

However a graph called a N node doubly rooted complete binary (DRCB) tree is contained in an N hypercube.

The DRCB tree is formed by replacing the root of a complete binary tree with 2 nodes that are connected. Left root is then connected to the left sub-tree and the right root is connected to the right sub-tree.

Consequently a DRCB tree on N nodes contains two $N/2-1$ node complete binary trees.

Also, the DRCB tree contains an $N-1$ node complete binary tree with *dilation* 2.

Therefore the N node hypercube contains a $N-1$ node complete binary tree with dilation 2, or two $N/2-1$ node complete binary trees with dilation 1.

A dilation of d for an embedding means that each edge of the contained graph is "stretched" over at most d edges of the containing graph.

Proof that a N DRCB tree is embedded into a N node hypercube is by induction. The base cases are clear for $N = 2, 4, 8$.

Alternative embedding of a tree

Consider the embedding where each leaf node of a $2\,N-1$ complete binary tree is mapped to the hypercube node of the same number, numbered left to right, and each internal node of the tree is mapped to the same hypercube node as its left-most descendant leaf.

- The embedding is not one-to-one since $\log N + 1$ tree nodes are mapped to node $00\cdots0$ in the hypercube.
- Each edge of the tree is either mapped to a single node or to a single edge of the hypercube.
- Every internal node is mapped to the same hypercube node as its left child, and level k edges linking an internal node to its right child are mapped to dimension k hypercube edges for $1 \leq k \leq \log N$.

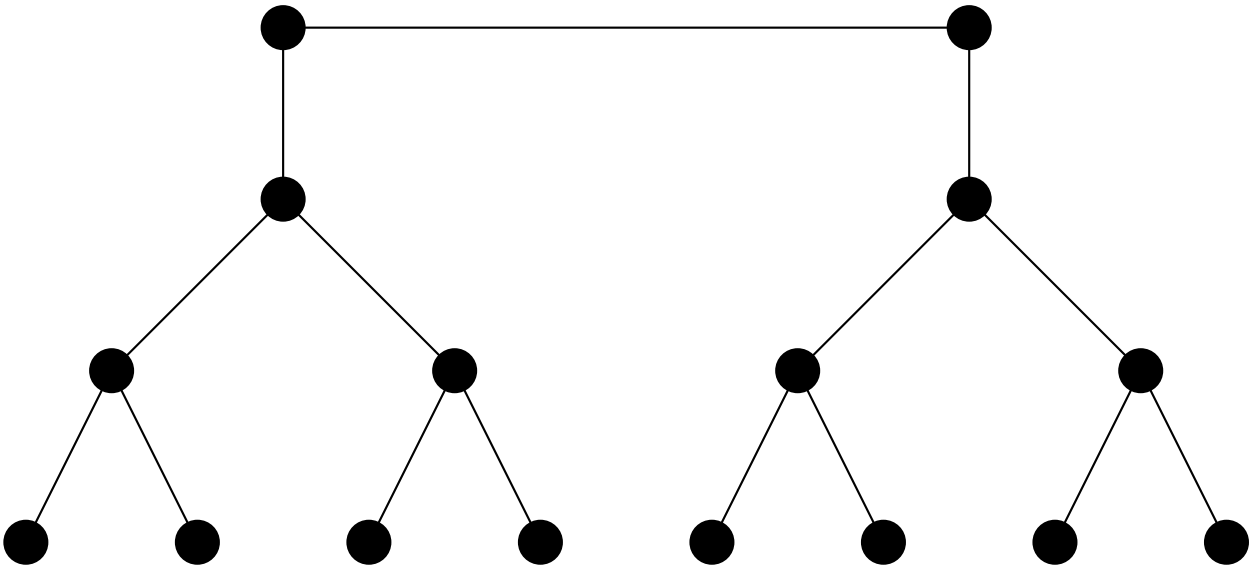


Figure 68: A 16 node DRCB tree.

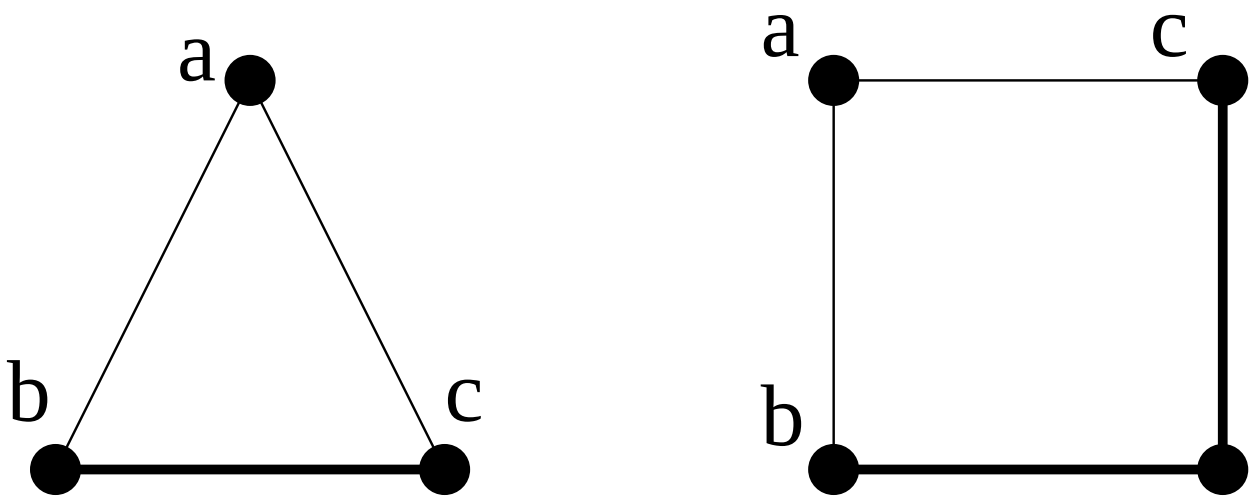


Figure 69: The triangle can be embedded into the square with a dilation of 2.

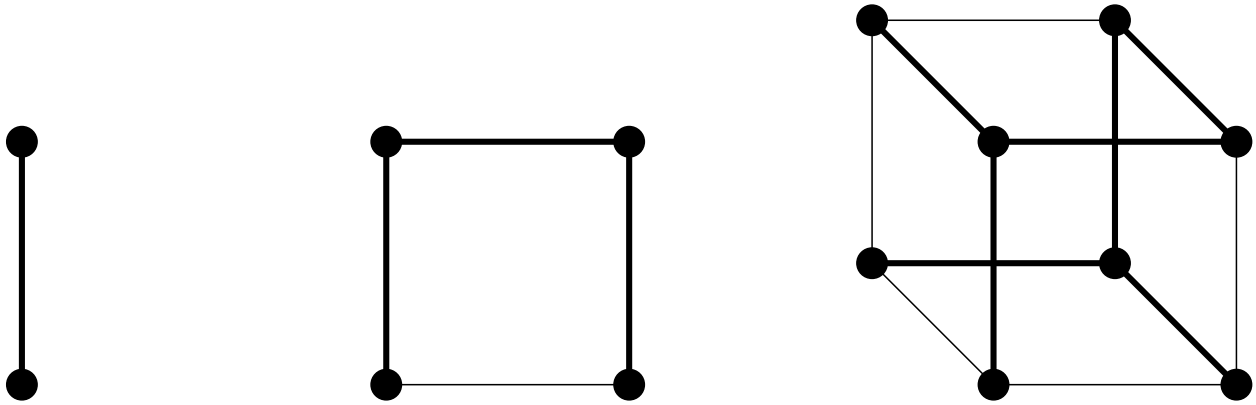


Figure 70: Bases cases for DRCB tree embedding.

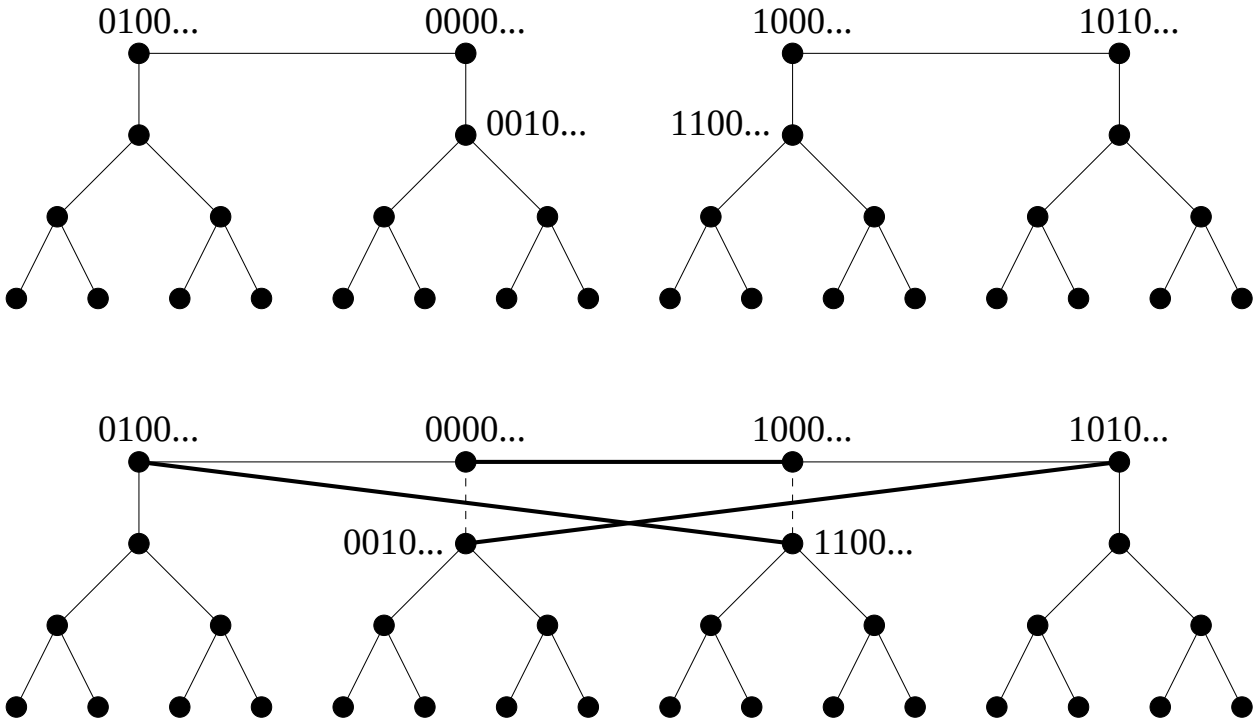


Figure 71: Joining two $N/2$ node DRCB trees to form a N node DRCB tree. Two orientation of the trees (i.e. the node labels) are important to get right.

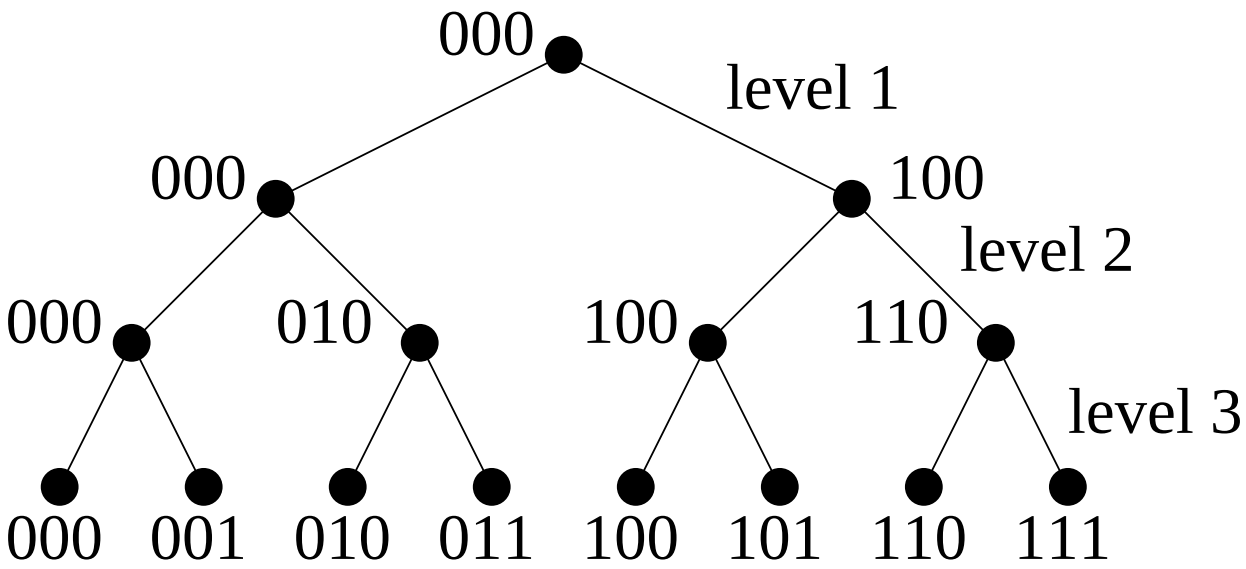


Figure 72: A $2N-1$ complete binary tree embedded in an N node hypercube.

- At most one node on any level of the tree is mapped to any hypercube node. Hence, any computation on the tree that uses only nodes on one level can be performed in one step on the hypercube, and any communication using only level k edges of the tree can be performed in one step on the hypercube using only dimension k edges.

General aspects of embeddings

Dilation of an embedding was described earlier. The *load* of an embedding is the maximum number of nodes of the contained graph that are embedded in any single node of the containing graph. For a load of L, an algorithm which runs on the contained graph in T time steps will take LT time steps after embedding.

The *expansion* of an embedding is the ratio of the number of nodes in the containing graph to

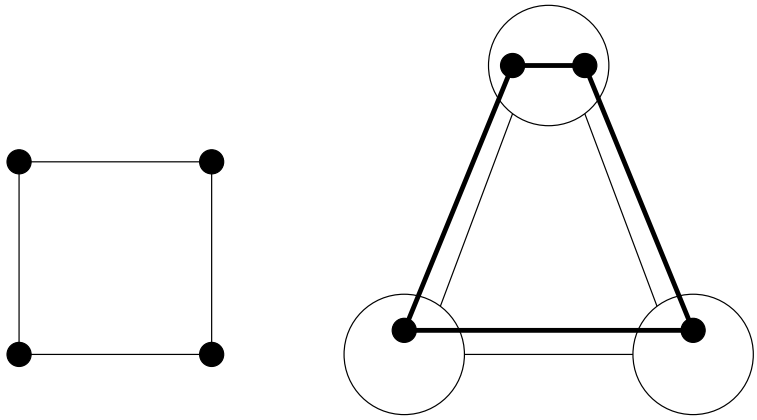


Figure 73: An embedding of a square into a triangle with load 2.

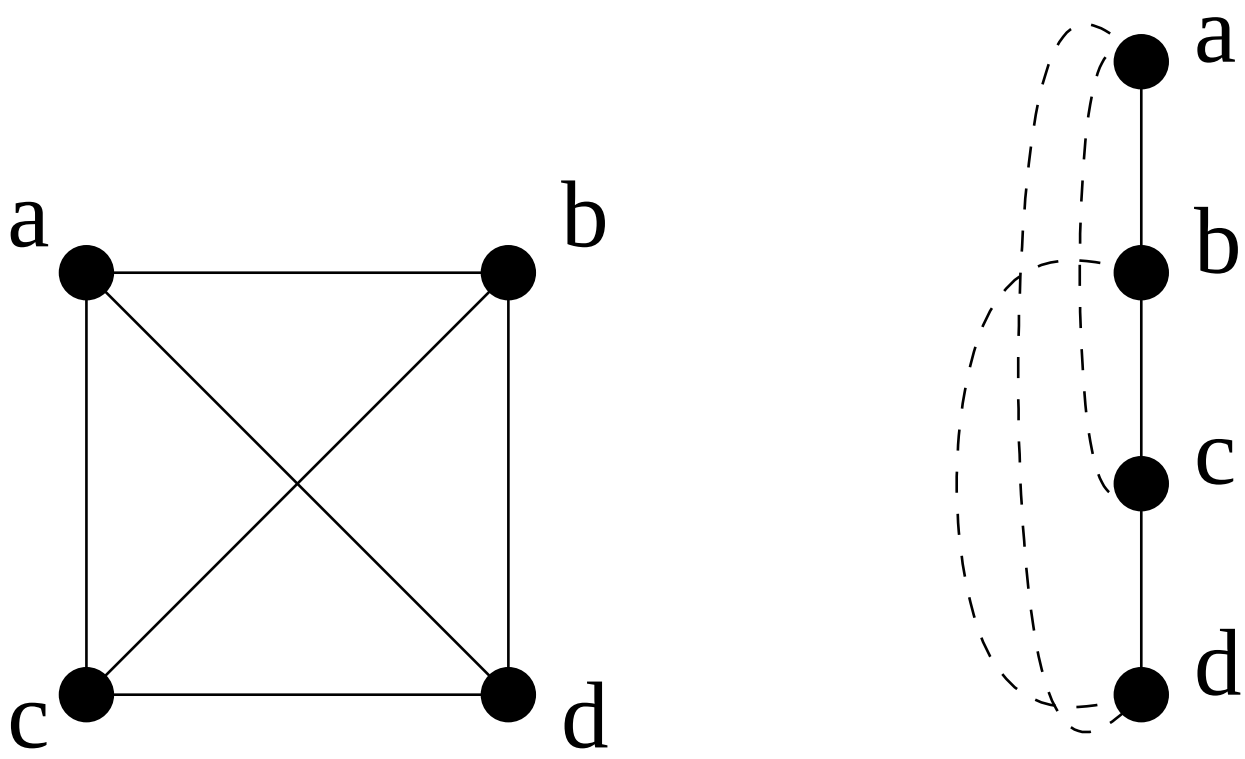


Figure 74: An embedding of a complete graph one a line with congestion 4 and dilation 3.

the number of nodes in the contained graph. E.g. a $N/2-1$ complete binary tree can be embedded in an N node hypercube with expansion $N/(N/2-1) \approx 2$, load 1, and dilation 1.

The *congestion* of an embedding is the maximum number of edges of the contained graph that are embedded using any single edge of the host graph.

Clearly, it is desirable to have embeddings with minimum dilation, load, expansion and congestion.