# Parallel Algorithms

Peter Harrison and William Knottenbelt

Email: pgh@doc.ic.ac.uk

Department of Computing, Imperial College London



## Course Structure

- 18 lectures
- 6 regular tutorials
- 3 lab-tutorials
- 1 revision lecture-tutorial (optional)



## Course Assessment

- Exam (answer 3 out of 4 questions)
- Either:
  - one laboratory exercise
  - one assessed coursework
- or:
  - two laboratory exercises
- 1 revision lecture-tutorial (optional)



## Recommended Books

- Kumar, Grama, Gupta, Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings. Second Edition, 2002.
  First Edition, 1994, is OK.

  **Main course text**
- Freeman and Phillips. *Parallel Numerical Algorithms*. Prentice-Hall, 1992.

  **Main text for stuff on differential equations**

# Other Books

- Cosnard, Trystram. *Parallel Algorithms and Architectures*. International Thomson Computer Press, 1995.
- Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1994.
- Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989.

  **An old classic**



# Course Outline

| Topic | No. of le |
|---|---|
| Architectures & communication networks | 4 |
| Message Passing Interface (MPI) | 2 |
| Parallel performance metrics | 2 |
| Dense matrix algorithms | 4 |
| Sparse matrix algorithms | 2 |
| Dynamic search algorithms | 4 |
| TOTAL | 18 |



# Computer Architectures

1. Sequential
   - John von Neumann model: CPU + Memory
   - Single Instruction stream, Single Data stream (SISD)
   - Predictable performance of (sequential) algorithms with respect to von Neumann machine



# Computer Architectures

2. Parallel
   - Multiple cooperating processors, classified by control mechanism, memory organisation, interconnection network (IN)
   - Performance of parallel algorithm depends on target architecture and how it is mapped

## Control Mechanisms

- Single Instruction stream, Multiple Data stream (SIMD): all processors execute the same instructions synchronously $\Rightarrow$ good for *data parallelism*
- Multiple Instruction stream, Multiple Data stream (MIMD): processors execute their own programs asynchronously $\Rightarrow$ more general
  - process networks (static)
  - divide-and-conquer algorithms (dynamic)



## Control Mechanisms – hybrid

- Single Program, Multiple Data stream (SPMD): all processors *run the same program asynchronously*
  - Hybrid SIMD / MIMD
  - also suitable for data-parallelism but needs explicit synchronisation



## Memory Organization

1. Message-passing architecture
   - Several processors with their own (local) memory interact *only* by message passing over the IN
   - *Distributed memory architecture*
   - MIMD message-passing architecture $\equiv$ multicomputer
2. Shared address space architecture
   - Single address space shared by all processors



## Memory Organization (2)

2. Shared address space architecture (cont.)
   - *Multiprocessor* architecture
   - Uniform memory access (UMA) $\Rightarrow$ (average) access time same for all memory blocks: e.g. single memory bank (or hierarchy)
   - Otherwise non-uniform memory access (NUMA): e.g. global address space is distributed across the processors' local memories (distributed shared memory multiprocessor)
   - Also *cache hierarchies* imply less uniformity

# Interconnection Network

1. Static (or direct) networks
   - Point to point communication amongst processors
   - Typical in message-passing architectures
   - Examples are ring, mesh, hypercube
   - Topology critically affects parallel algorithm performance (see coming lectures)



# Interconnection Network (2)

2. Dynamic (or indirect) networks
   - Connections between processors are constructed dynamically during execution using *switches*, e.g. crossbars or networks of these such as *multistage banyan (or delta, or omega, or butterfly) networks*.
   - Typically used to implement shared address space architectures
   - But also in some message-passing algorithms; e.g. the FFT on a butterfly (see textbooks)



# Parallel Random Access Machine

- The PRAM is an idealised model of computation on a shared-memory MIMD computer
- Fixed number $p$ of processors
- Unbounded UMA global memory
- All instructions last one cycle
- Synchronous operation ('common clock') but different instructions are allowed in different processors on the same cycle



# PRAM memory access modes

Four modes of 'simultaneous' memory access (2 types of access, 2 modes)

EREW: Exclusive read, exclusive write. Weakest PRAM model, minimum concurrency.

CREW: Concurrent read, exclusive write. Better.

CRCW: Concurrent read, concurrent write. Maximum concurrency. Can simulate on a EREW PRAM (exercise)

ERCW: Exclusive read, concurrent write. Unusual?

## Concurrent Write Semantics

Arbitration is needed to define a unique semantics of concurrent write in CRCW and ERCW PRAMs

*Common* All values to be written are the same

*Arbitrary* Pick one writer at random

*Priority* All processors have a preassigned priority

*Reduce* Write the (generalised) sum of all values attempting to be written. 'Sum' can be *any* associative and commutative operator – cf. 'reduce' or 'fold' of functional languages.



## PRAM role

- Natural extension of the von Neumann model with zero cost communication (via shared memory)
- We will use the PRAM to assess the complexity of some parallel algorithms
- Gives an upper bound on performance, e.g. minimum achievable latency



## Static Interconnection Networks

1. Completely connected
   - direct link between every pair of processors
   - ideal performance but complex and expensive
2. Star
   - all communication through a special 'central' processor
   - central processor liable to become a bottleneck
   - logically equivalent to a bus – associated with shared memory machines (dynamic network)



## Static Interconnection Networks (2)

3. Linear array and ring
   - connect processors in tandem
   - with wrap-around gives a ring
   - communication via multiple 'hops' over links through intermediate processors
   - basis for quantitative analysis of many other common networks

# Static Interconnection Networks (3)

4. Mesh
   - generalisation of linear array (or ring with wrap-around) to more than one dimension
   - processors labelled by rectilinear coordinates
   - links between adjacent processors on each coordinate axis (i.e. in each dimension)
   - multiple paths between source and destination processors



# Static Interconnection Networks (4)

5. Tree
   - unique path between any pair of processors
   - processors reside at the leaves of the tree
   - Internal nodes may be processors (typical in static network) or switches (typical in dynamic networks)
   - bottlenecks higher up the tree
   - can alleviate by increasing bandwidth at higher levels $\rightarrow$ *fat tree* (e.g. in CM5)



# Cube Networks

In a $k$-*ary* $d$-*cube* topology – of dimension $d$ and radix $k$ – each processor is connected to $d$ others (with wrap-around) and there are $k$ processors along each dimension

- Regular $d$-dimensional mesh with $k^d$ processors
- Processors labelled by $d$ digit number with radix $k$
- Ring of $p$ processors is a $p$-ary 1-cube
- Wrap-around mesh of $p$ processors is a $\sqrt{p}$-ary 2-cube



# Hypercubes

- A $k$-ary $d$-cube can be formed from $k$ $k$-ary $(d-1)$-cubes by connecting corresponding nodes into rings
  e.g. composition of rings to form a wrap-around mesh
- Hypercube $\equiv$ binary $d$-cube
  - nodes labelled by binary numbers of $d$ digits
  - each node connected directly to $d$ others
  - adjacent nodes differ in exactly one bit

# Embeddings into Hypercubes

Hypercube is the most richly connected topology we have considered (apart from completely connected) so can we consider other topologies as *embedded subnetworks*?

1. Ring of $2^d$ nodes
   - Need to find a sequence of adjacent nodes, with wraparound, in a $d$-hypercube
   - Adjacent node labels differ in exactly one bit position



# Mapping: ring $\rightarrow$ hypercube

- Assign processor $i$ in the ring to node $G(i, d)$ in the hypercube where $G$ is the *binary reflected Gray code* (RGC) defined by: $G(0,1) = 0, \quad G(1,1) = 1$ and

$$G(i, n+1) = \begin{cases} G(i,n) & i < 2^n \\ 2^n + G(2^{n+1} - 1 - i, n) & i \geq 2^n \end{cases}$$

- This is easily seen recursively, by concatenating the mapping for a $(d-1)$-hypercube with its reverse and pre- (or app-)ending a 0 onto one mapping and a 1 onto the other .....



# Why is this true?

Proof by induction: a sketch (all that is necessary here) is:

1. Certainly true for $d = 1$, when $0 \mapsto 0$ and $1 \mapsto 1$
2. For $d \geq 0$, *assume* successive node addresses in any $d$-cube ring mapping differ in only one bit
3. Hence same applies in each half of the RGC for a $(d+1)$-cube
4. But because of the reflection, the same holds for adjacent nodes in *different* halves.



# Mapping: mesh $\rightarrow$ hypercube

- The mapping for an $m$ dimensional mesh is obtained by concatenating the RGCs for each individual dimension
- Thus node $(i_1, \ldots, i_m)$ in a $2^{r_1} \times \ldots \times 2^{r_m}$ mesh maps to node

$$G(i_1, r_1) <> \ldots <> G(i_m, r_m)$$

- E.g. in an $8 \times 8$ square mesh, the node at coordinate $(2, 7)$ maps to hypercube node $(0, 1, 1, 1, 0, 0)$.

## Mapping: tree $\longrightarrow$ hypercube

- Consider a (complete) binary tree of depth $d$ with processors at the leaves only
- This embeds into a $d$-hypercube as follows, via a many-to-one mapping that maps *every* node
  1. map the root (level 0) to any node, e.g. $(0, \ldots, 0)$
  2. For each node at level $j$, if mapped to hypercube node $\vec{k}$, map the left child to $\vec{k}$ and the right child to $\vec{k}$ with bit $j$ inverted.
  3. repeat for $j = 1, \ldots, d$



## Monotonicity of the mapping

- Distance between two tree-nodes is $2n$ for some $n \geq 1$ (difference between $d$ and the level of the lowest common ancestor)
- The corresponding distance in the hypercube is $n$ – think of bit-changes
- Nodes further apart in the hypercube must be further apart in the tree, but the *converse may not hold*:
  - because of richer hypercube connectivity
  - some bits might flip back
  - distant tree-nodes might *happen* to be closer in the hypercube: $d$ are adjacent



## Communication Costs

Time spent sending data between processors in a parallel algorithm is a significant overhead – *communication latency* – defined by the *switching mechanism* and parameters:

1. *Startup time*, $t_s$ : message preparation, route initialisation etc. Incurred *once per message*.
2. *Per-hop time*, or node latency, $t_h$ : time for header to pass between directly connected processors. Incurred for *every link* in a path.
3. *Per-word transfer time*, $t_w$ : $t_w = 1/r$ for channel bandwidth $r$ words per second. Relates message length to latency.



## Switching Mechanisms

1. Store-and-forward routing

- Each intermediate processor on a communication path receives an entire message and *only then* sends it on to the next node on the path
- For a message of size $m$ words, the communication latency on a path of $l$ links is:

$$t_{\text{comm}} = t_s + (mt_w + t_h)l$$

- Typically $t_h$ is small and so we often approximate $t_{\text{comm}} = t_s + mt_w l$

# Switching Mechanisms (2)

2. Cut-through routing
   - Reduce idle time of resources by 'pipelining' messages along a path 'in pieces'
   - Messages are advanced to the out-link of a node *as they arrive* at the in-link
   - Wormhole routing splits messages into *flits* (flow-control digits) which are then pipelined



# Wormhole Routing

- As soon as a flit is completely received, it is sent on to the next node in the message's path (same path for all flits)
- No need for buffers for *whole messages* – unless asynchronous multiple inputs are allowed for the same out-link
- Hence more time-efficient *and* more memory efficient
- But in a bufferless system, messages may become *blocked* (waiting for a processor already transmitting another message) $\Rightarrow$ possible *deadlock*



# Wormhole Routing (2)

- On an $l$-link path, header flit latency $= lt_h$
- An $m$-word message will all arrive $mt_w$ after the header
- For a message of size $m$ words, the communication latency on a path of $l$ links is therefore:

$$t_{\text{comm}} = t_s + mt_w + lt_h$$

  - $\Theta(m + l)$ for cut-through vs. $\Theta(ml)$ for store-and-forward
  - similar for small $l$ (identical for $l = 1$)



# Communication Operations

- Certain types of computation occur in many parallel algorithms
- Some are implemented naturally by particular communication patterns
- We consider the following patterns of communication – where the *dual operations*, with the direction of the communication reversed, are shown in brackets ...

# Communication Patterns

- simple message transfer between two processors (same for dual)
- one-to-all broadcast (single node accumulation)
- all-to-all broadcast (multi-node accumulation)
- one-to-all personalised (single node gather)
- all-to-all personalised, or 'scatter' (multi-node gather)
- more exotic patterns, e.g. permutations



# Simple Message Transfer

- Most basic type of communication
- Dual operation is of the same type
- Latency for single message is :
  - $T_{\text{smt-sf}} = t_s + t_w ml + t_h l$ for store-and-forward routing
  - $T_{\text{smt-ct}} = t_s + t_w m + t_h l$ for cut-through routing
- where $l$ is the number of hops ...



# Number of hops, $l$

This depends on the network topology – $l$ is at most:

- $\lfloor p/2 \rfloor$ for a *ring*
- $2\lfloor \sqrt{p}/2 \rfloor$ for a *wrap-around* square mesh of $p$ processors ($\lfloor a/2 \rfloor + \lfloor b/2 \rfloor$ for an $a \times b$ mesh)
- $\log p$ for a *hypercube*

So for a hypercube with cut-through,

$$T_{\text{smt-ct-h}} = t_s + t_w m + t_h \log p$$



# Comparison of SF and CT

- If message size $m$ is very small, latency is similar for SF and CT
- If message size is large, i.e. $m >> l$, CT becomes asymptotically independent of path length $l$
  - CT *much faster* than SF
  - $T_{\text{smt-ct}} \simeq t_w m \simeq$ *single hop latency under SF*

# One-to-All Broadcast (OTA)

- Single processor sends data to *all* or *a subset of* other processors
- E.g. matrix-vector multiplication: broadcast each element of the vector over its corresponding column
- In the dual operation, single node accumulation, data may not only be collected but also mapped by an associative operator
  - e.g. sum a list of elements initially distributed over processors
  - cf. concurrent write in PRAM



# All-to-All Broadcast (ATA)

- Each processor performs (simultaneously) one-to-all broadcast *with its own data*
- Used in matrix operations, e.g. matrix multiplication, reduction and parallel-prefix
- In the dual operation – multinode accumulation – each processor receives single-node accumulation
- Could implement ATA by sequentially performing $p$ OTAs
- Far better to proceed in parallel and catenate incoming data



# Reduction

To broadcast the reduction of the data held in all processors with an associative operator, we can:

1. ATA broadcast the data and then reduce locally in every node . . . *inefficient*
2. Single node accumulation at one node followed by OTA broadcast . . . *better*
3. Modify ATA broadcast so that instead of catenating messages, the incoming data and the current accumulated value are operated on by the associative operator – e.g. summed – the result overwriting the accumulated value ..... the most efficient



# Parallel Prefix

- The *Parallel Prefix* of a function $f$ over a non-null list $[x_1, \ldots, x_n]$ is the list of reductions of $f$ over all sublists $[x_1, \ldots, x_i]$ for $1 \leq i \leq n$, where $\texttt{reduce}f[x_1] = x_1$ for all $f$
- Could implement as $n$ reductions
- Better to modify the third reduction method by *only updating the accumulator at each node when data comes in from the appropriate nodes* (otherwise it is just passed on)

# All-to-All Personalised

- Every processor sends a distinct message of size $m$ to every other processor – 'total exchange'
- E.g. in matrix transpose, FFT, database join
- Communication *patterns* identical to ATA
- Label messages by pairs $(x, y)$ where $x$ is the source processor and $y$ is the destination processor: *uniquely determines the message contents*
- List of $n$ messages denoted $[(x_1, y_1), \ldots, (x_n, y_n)]$



# Performance Metrics

1. Run Time, $T_p$
   - A parallel algorithm is hard to justify without improved run-time
   - $T_p$ = Elapsed time on $p$ processors: between the start of computation on the *first processor to start*, and termination of computation on the *last processor* to finish



# Performance Metrics (2)

2. Speed-up, $S_p$

$$S_p = \frac{\text{serial run-time of "best" sequential algorithm}}{T_p}$$

   - "best" algorithm is the *optimal* one for the problem, if known, or the fastest known, if not
   - often in practice (always in this course) $T_1$
   - $S_p \geq 1 \quad \ldots \quad$ usually!



# Example – addition on hypercube

- Add up $p = 2^d$ numbers on a $d$-hypercube
- Use *single node accumulation*
- Each single-hop communication combined with one addition operation

$\Rightarrow S_p = \Theta(p/\log p)$

# Performance Metrics (2)

3. Efficiency, $E_p$

$$E_p = \frac{S_p}{p}$$

- Fraction of time for which a processor is doing useful work
- $E_p = \Theta(1/\log p)$ in above example



# Performance Metrics (3)

4. Cost, $C_p$

$$C_p = p \times T_p \quad \text{so that} \quad E_p = \frac{\text{best serial run-time}}{C_p}$$

- A parallel algorithm is *cost-optimal* if $C_p \propto$ best serial run time
- Equivalently if $E_p = \Theta(1)$
- Above example is *not* cost-optimal since best serial run time is $\Theta(p)$



# Granularity

- "Amount of work allocated to each processor"
- Few processors, relatively large processing load on each $\Rightarrow$ *coarse-grained parallel algorithm*
- Many processors, relatively small processing load on each $\Rightarrow$ *fine-grained parallel algorithm*
  - e.g. our hypercube algorithm to add up $p$ numbers
  - typically many small communications, often in parallel



# Increasing the granularity

- Let each processor "simulate" $k$ processors in a finer-grained parallel algorithm
- Computation at each processor increases by a factor $k$
- Communication time increases by factor $\leq k$
  - typically $<< k$
  - but may have much larger message *sizes*, e.g. $k$ parallel communications may map to a single communication $k$ times bigger
- Hence $T_{p/k} \leq k \times T_p$ and so $C_{p/k} \leq C_p$
- Cost-optimality *preserved* – may be *created*?

## Addition on Hypercube Again

- Add $n$ numbers on a $d$-hypercube of $p = 2^d$ processors
- Let each processor simulate $k = n/p$ processes (assuming $p \mid n$)
- Each processor adds locally $k$ numbers in $\Theta(k)$ time
- $p$ partial sums are added in $\Theta(\log p)$ time
- $T_p = \Theta(k + \log p)$ and $C_p = \Theta(n + p \log p)$
- Cost optimal if $n = \Theta(p \log p)$



## Addition on Hypercube Again (2)

Alternatively, try communication in the first $\log p$ steps, followed by local addition of $n/p$ numbers

- $T_p = \Theta((n/p) \log p)$
- So $C_p = \Theta((n) \log p) = \log p \times \Theta(C_1)$
- *never cost-optimal*



## Scalability

- Efficiency decreases as the number of processors increases
- Consequence of *Amdahl's Law*:

$$S_p \leq \frac{\text{problem size}}{\text{size of serial part of problem}}$$

where size is the *number of basic computation steps in the best serial algorithm*



## Scalability (2)

- A parallel system is scalable if it can maintain the efficiency of a parallel algorithm by simultaneously increasing the number of procesors and problem size
- E.g. in the above example, efficiency remains at 80% if $n$ is increased with $p$ as $8p \log p$
- But you can't tell me why *yet*!

# The Isoefficiency Metric

- Measure of the extent to which a parallel system is scalable
- Define the *overhead*, $O_p$ to be the amount of computation not performed in the best serial algorithm

$$O_p = C_p - W$$

  where $W$ is the problem size
- $O_p$ includes setup overheads and possible changes to an algorithm to make it parallel, but usually (100% in this course) comprises the *communication latency*



# Overhead in Hypercube-Addition

- For the above addition on a hypercube example, at granularity $k = n/p$

$$T_p = n/p + 2\log p$$

- assuming time 1 for addition and single hop communication. Then

$$O_p = 2p\log p$$



# Isoefficiency

- For a *scalable system*, the *isoefficiency function* $I$ determines $W$ in terms of $p$ and $E$ such that *efficiency*, $E_p$, *is fixed at some specified constant value* $E$

$$\begin{aligned} E &= S_p/p = W/C_p \\ &= \frac{W}{W + O_p(W)} \\ &= \frac{1}{1 + O_p(W)/W} \end{aligned}$$



# Isoefficiency (2)

- Rearranging, $1 + O_p(W)/W = 1/E$, and so

$$W = \frac{E}{1-E} O_p(W)$$

- This is the *Isoefficiency Equation*
- Setting $K = E/(1-E)$ for our given $E$, let the solution of this equation (assuming it exists, i.e. for a scalable system) be $W = I(p, K)$ – the *Isoefficiency function*

## Back to Hypercube-Addition

- For the addition on a hypercube example it's easy:
$$I(p, K) = 2Kp \log p$$
- More generally, $O_p$ varies with $W$ and the isoefficiency equation is non-trivial, e.g. non-linear
- Plenty of examples in the rest of the course!



## Cost-optimality and Isoefficiency

- A parallel system is cost-optimal *if and only if* $C_p = \Theta(W)$, i.e. its cost is asymptotically the same as the cost of the serial algorithm
- This implies the *upper bound* on the overhead
$$O_p(W) = O(W)$$
or *lower bound* on the problem size
$$W = \Omega(O_p(W))$$
- Not surprising – you don't want a bigger overhead than the computation of the solution itself!



## Cost-optimality and Isoefficiency (2

For the above example $W = \Theta(n)$ and $O_p(W) = 2p \log p$ so that the system cannot be cost-optimal unless $n = \Omega(p \log p)$

- the condition for cost-optimality already derived
- the system is then scalable – its isoefficiency function is $\Theta(p \log p)$



## Minimum Run-Time

- Assuming differentiability of the expression for $T_p$, find $p = p_0$ such that
$$\frac{dT_p}{dp} = 0$$
giving $T_p = T_p^{\text{min}}$
- For the above example,
$$\begin{aligned} T_p &= n/p + 2 \log p \\ p_0 &= n/2 \Rightarrow T_p^{\text{min}} = 2 \log n \end{aligned}$$
- *Not cost-optimal*

# Minimum Cost-Optimal Run-Time

- For isoefficiency function $\Theta(f(p))$ (at any efficiency), $W = \Omega(f(p))$ or $p = O(f^{-1}(W))$
- Then minimum *cost-optimal run time* is

$$T_p^{\text{min-cost-opt}} = \Omega(W/f^{-1}(W))$$

- For our example, $n = f(p) = p \log p$ and we find $p = f^{-1}(n) = n/\log p \simeq n/\log n$ so that

$$T_p^{\text{min-cost-opt}} \simeq 3 \log n - 2 \log \log n$$

- here, same asymptotic complexity as $T_p^{\text{min}}$