Design and Analysis of Parallel Algorithms

Murray Cole

e-mail: mic room: 2508

What?

Devising algorithms which allow many processors to work collectively to solve

- the same problems, but faster
- bigger/more refined problems in the same time

when compared to a single processor.

Why?

Because it is an interesting intellectual challenge!

Because parallelism is everywhere and we need algorithms to exploit it.

- Global scale: computational grids, distributed computing
- Supercomputer scale: Top 500 HPC, scientific simulation & modelling, Google
- Desktop scale: commodity multicore PCs and laptops
- Specialised hardware: custom parallel circuits for key operations (encryption, multimedia)

How?

We will need

- machine model(s) which tell us what the basic operations are in a "reasonably" abstract way
- cost model(s) which tell us what these operations cost, in terms of resources we care about (usually time, sometimes memory)
- analysis techniques which help us map from algorithms to costs with "acceptable" accuracy
- metrics which let us discriminate between costs (e.g. speed v. efficiency)

History Lesson: Sequential Computing

Precise cost modelling of sequential algorithms is already very hard.

The impact of memory hierarchy (cache behaviour) and (dynamic) instruction scheduling can be considerable.

Compiler optimisations complicate our attempts even further.

Easier, but less accurate to just count operations of some interesting (problem dependent) kind (arithmetic ops, comparisons, memory accesses) and to simplify the memory model to the Random Access Machine (RAM), in which each memory access has the same unit cost.

Sequential matrix multiply in $\Theta\left(n^3\right)$ operations

```
for (i=0;i<N;i++){
   for (j=0;j<N;j++){
       for (k=0; k<N; k++){
           c[i][j] += a[i][k]*b[k][j];
       }
   }
}
```

Sequential matrix multiply in $\Theta\left(n^3\right)$ operations again, but much faster.

```
for (jj=0; jj<N; jj=jj+B)
  for (kk=0; kk<N; kk=kk+B)
     for (i=0; i<N; i++)
        for (j=jj; j < jj+B; j++) {
           pa = &a[i][kk]; pb = &b[kk][j];
           temp = (*pa++)*(*pb);
           for (k=kk+1; k < kk+B; k++) {
             pb = pb+N;
             temp += (*pa++)*(*pb);
           }
           c[i][j] += temp;
        }
}
```

Asymptotic Analysis

Even counting such operations exactly can be difficult (and pointless, given the inaccuracies already introduced).

Often we will settle for capturing the rough growth rate with which resources (memory, time, processors) are used as functions of problem size.

Sometimes run-time may vary for different inputs of the same size. In such cases we will normally consider the worst case run-time. Occasionally we will also consider the average case (ie averaged across all possible inputs).

Asymptotic Analysis

Asymptotic ("big-O") notation captures this idea as "upper", "lower", and "tight" bounds.

- $f(n) = O(g(n)) \Leftrightarrow \exists c > 0, n_0$ such that $f(n) \leq cg(n)$, $\forall n > n_0$ ("no more than")
- $f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$ ("at least")
- $f(n) = \Theta(g(n)) \Leftrightarrow$ both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ ("roughly")

Note that we are throwing away constant factors! In practice these are sometimes crucial. Asymptotics are useful as (precisely defined) rules of thumb.

Parallel Computer Structures

Dominant programming models reflect an underlying architectural divergence:

- the shared address space model allows threads (or "lightweight processes") to interact directly through common memory locations. Care is required to avoid races and unintended interactions.

- the message passing model gives each process its own address space. Care is required to distribute the data across these address spaces and to implement copying between them (by sending and receiving "messages") as appropriate.

How should we reflect these in our machine and cost model(s)?

Differences in structure between parallel architectures (even in the same "class") make the complications of sequential machines seem trivial!

Shared Address Space Parallelism

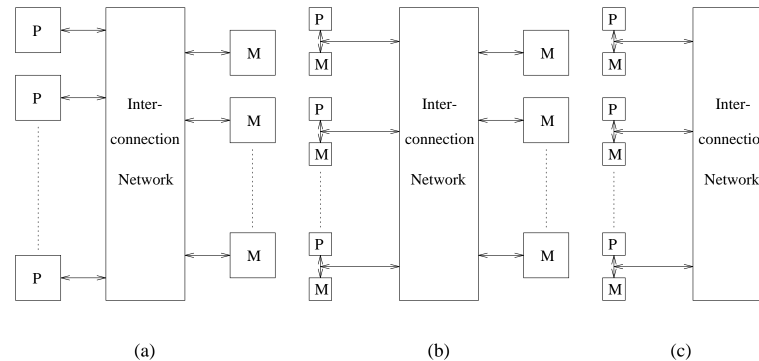


Figure 2.5 Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Non-uniform-memory-access shared-address-space computer with local and global memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

- Real costs are complicated by "cache coherence" support and congestion in the network (i.e. hidden communications).

- We will take a simplified view for algorithm design, using the PRAM model.

The Parallel Random Access Machine (PRAM)

The PRAM is an idealised model of a shared address space computer, developed as an extension of the RAM model used in sequential algorithm analysis.

- p processors, synchronized at each step, m shared memory locations with each step costing unit time (irrespective of p, m)
- memory clash resolution (reads before writes)
  - EREW (exclusive-read exclusive-write)
  - CREW (concurrent-read exclusive-write)
  - CRCW (concurrent-read concurrent-write), with common/arbitrary/priority/associative write resolution variants

Useful starting point developing more pragmatic (e.g. cache aware) algorithms.

Summing n integers

CRCW algorithm (resolution: associative with $+$)

```
int a[n], sum;
for i = 0 to n-1 do in parallel
  sum = a[i];
```

which is a $\Theta(1)$ (constant) time algorithm given $p = n$ processors.

Summing n integers

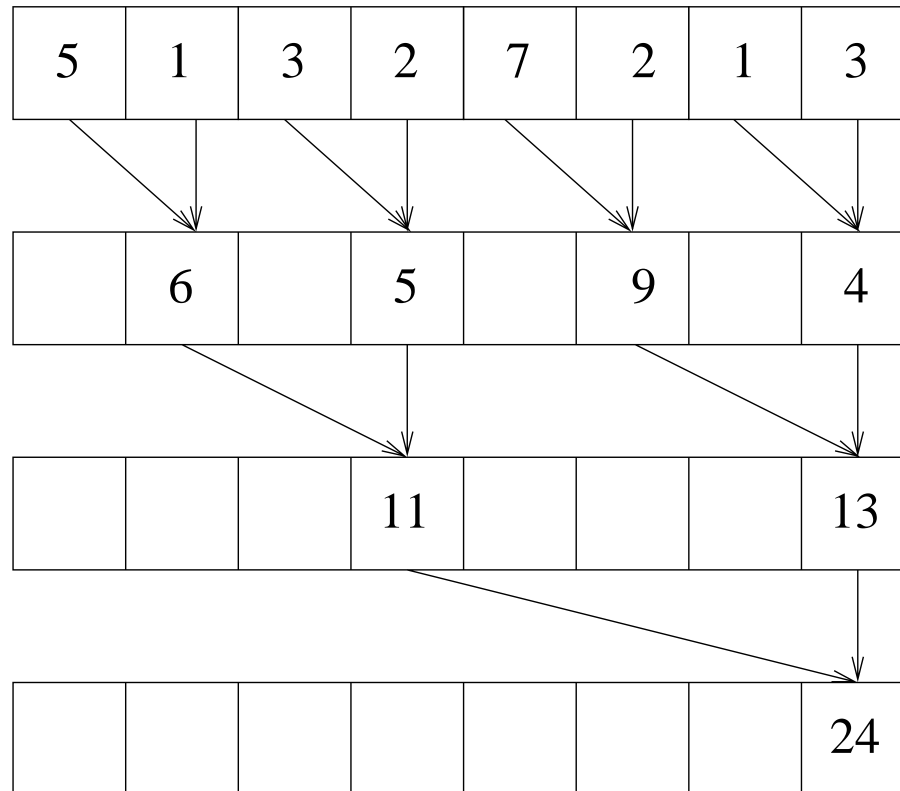
EREW algorithm

```
for i = 0 to n-1 do in parallel
  temp[i] = a[i];

for i = 0 to n-1 do in parallel
 for j = 1 to log n do
  if (i mod 2^j == 2^j - 1) then
   temp[i] = temp[i] + temp[i - 2^(j-1)];

sum = temp[n-1];
```

which is a $\Theta(\log n)$ time algorithm, given $p = n$ processors.


5
1
3
2
7
2
1
3
6
5
9
4
11
13
24

Too Powerful?

We have seen that the choice of PRAM variant can affect achievable performance asymptotically, and that the most powerful model can achieve surprising results.

For example, it is possible to devise a constant time sorting algorithm for the CRCW (associative with $+$) PRAM, given n^2 processors.

Whether or not we "believe in" one model or another, working from this starting point may help us to devise unusual algorithms which can subsequently be made more realistic.

Constant Time Sorting

```
for i = 0 to n-1 do in parallel
  for j = 0 to n-1 do in parallel
     if (A[i]>A[j]) or (A[i]=A[j] and i>j) then
       wins[i] = 1;
     else
       wins[i] = 0;
for i = 0 to n-1 do in parallel
  A[wins[i]] = A[i];
```

(The second clause in the conditional resolves ties)

Metrics for Parallel Algorithms

How shall we compare our parallel algorithms? We could just focus on run time, assuming $p = n$ and compare with run time of best sequential algorithm.

For any real system, p is fixed, but making it a constant means it would disappear asymptotically, and we would also like to have algorithms which scale well when we add processors.

As a compromise, we treat p as a variable in our analysis, and look for algorithms which perform well as p grows as various functions of n. Informally, such algorithms will tend to be good when p is more realistic. To make this more concrete we define cost and efficiency.

Metrics for Parallel Algorithms

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

We will usually talk about cost in asymptotic terms.

For example, the run time of sequential (comparison based) sorting is $\Theta\left(n \log n\right)$.

A good parallel sorting algorithm might have run time $t = \Theta\left(\frac{n \log n}{p} + \log n\right)$. This will be asymptotically cost optimal if $p = O\left(n\right)$.

Another parallel sorting algorithm might have run time $t = \Theta\left(\frac{n \log n}{p} + \log^2 n\right)$. This will be asymptotically cost optimal if $p = O\left(\frac{n}{logn}\right)$.

Our "fast" CRCW parallel sorting algorithm has $t = \Theta\left(1\right)$ (constant) but requires $p = \Theta\left(n^2\right)$. This is fastest but not cost optimal.

The best algorithms are fast and cost optimal, making them easily scalable.

Scaling down efficiently

We can design parallel algorithms for unrealistically large p and use round-robin scheduling to scale-down to realistic p'.

Each of the p' real processors does the work of $\frac{p}{p'}$ logical processors, performing one operation for each in turn.

Run time increases by a factor of $\frac{p}{p'}$, while number of processors used decreases by the same factor. The cost is unchanged.

Cost optimal algorithms remain cost optimal, but the inefficiency of non cost optimal algorithms is exposed.

We can also scale down non-optimal algorithms to improve cost and efficiency if there is some slack (unused processors during some steps).

Brent's Theorem

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

Note distinction between steps and operations.

The theorem tells us that an algorithm must exist, but we may still have to think hard to present it in a concise form.

Since counting operations in practice is tricky, we will tend to use Brent in asymptotic terms, as a hint that a better algorithm is possible.

Application to EREW Summation

The original algorithm is not cost optimal ($p = n$ and $t = \log n$ against a sequential cost of $\Theta(n)$), so simple round-robin emulation will not be effective.

Brent (in asymptotic form) tells us that a computation with $p = \Theta\left(\frac{n}{\log n}\right)$, $t = \Theta(\log n)$ is possible, (but we still have to work out how).

The new algorithm has each processor summing $\log n$ items sequentially, then runs the original parallel algorithm with these partial results (and thus fewer processors).

We compensate (asymptotically) for an inefficient complex parallel phase with an efficient simple phase (initial independent summations). Asymptotically, the run time is unchanged (twice as long), but the algorithm is cost optimal.

Message Passing Parallelism

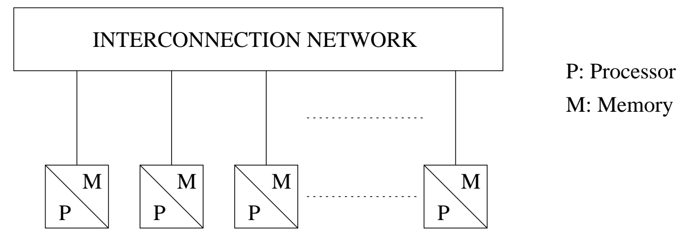


Figure 2.4 A typical message-passing architecture.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Could ignore network and model all process-process interactions as equal (with only message size counting), but we choose to open the lid a little, and consider network topology. As size gets large this can have an asymptotic impact. Also, some networks neatly match classes of algorithms and can help us in our algorithm design.

Many networks have been proposed and implemented, but we will focus on only two. Our first network is the mesh (usually 2-D) sometimes with wrap-around connections to make a torus.

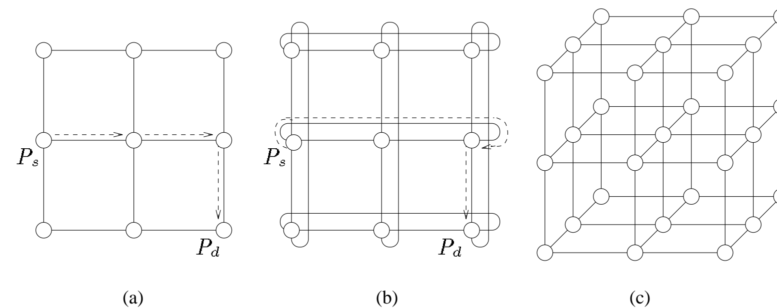


Figure 2.16 (a) A two-dimensional mesh with an illustration of routing a message from processor P_s to processor P_d; (b) a two-dimensional wraparound mesh with an illustration of routing a message from processor P_s to processor P_d; (c) a three-dimensional mesh.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

This has a natural association with many matrix based computations.

Our second network is the binary hypercube which has a natural association with many divide & conquer algorithms.

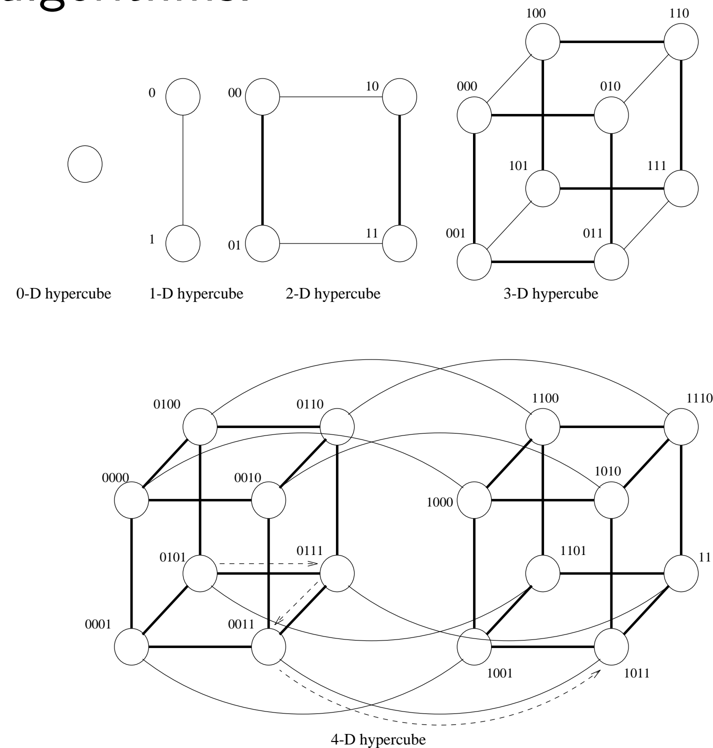
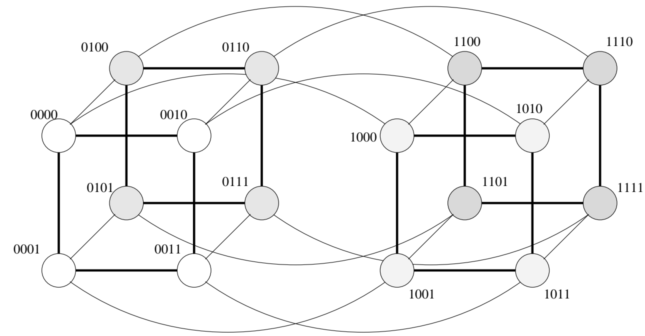
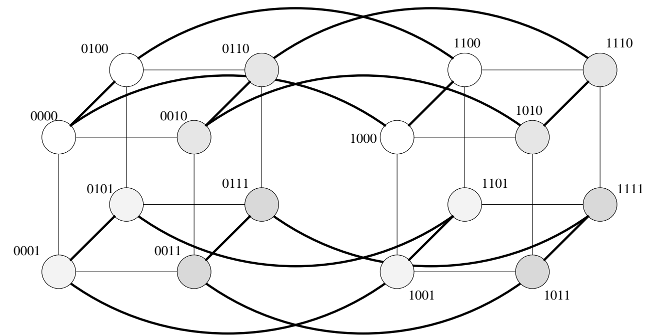


Figure 2.19 Hypercube-connected architectures of zero, one, two, three, and four dimensions. The figure also illustrates routing of a message from processor 0101 to processor 1011 in a four-dimensional hypercube.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.



(a)



(b)

Figure 2.21 The two-dimensional subcubes of a four-dimensional hypercube formed by fixing the two most significant label bits (a) and the two least significant bits (b). Processors within a subcube are connected by bold lines.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Message Passing Costs

Precise modelling of costs is impossible in practical terms.

Standard approximations respect two factors for messages passed between directly connected processors:

- every message incurs certain fixed start-up costs
- big messages takes longer than small ones

For messages involving several "hops" there are two standard models:

- Store-and-Forward (SF) routing
- Cut-Through (CT) routing

Message Passing Costs

In SF routing, we assume that the message is passed in its entirety between each pair of processors on the route, before moving to the next pair.

In CT routing, we assume that a message may be in transit across several links simultaneously. For a large enough message, this may allow routing time to be asymptotically independent of the number of links.

Message Passing Costs

For SF routing, in the absence of congestion, a message of size m words traversing l links takes time

- t_s (constant) in start-up overhead
- lmt_w for the transfer, where t_w is a constant which captures the inverse bandwidth of the connection

Asymptotically, this is just $\Theta(lm)$ (i.e. we lose the constants) so we should be aware that in practice this may not sufficiently penalise algorithms which use many very short messages.

Message Passing Costs

For CT routing, the situation is more complex (even in the absence of congestion), but time is modelled as proportional to $t_s + l + mt_w$. With a little hand-waving about small l and large m this is often simplified to $\Theta(m)$.

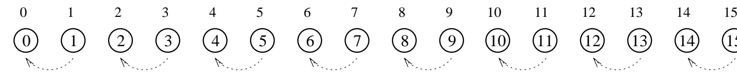
To avoid this unpleasantness all our algorithms (except one, to illustrate the point) are based on messages between directly connected processors.

To eliminate congestion, we further require that a processor sends/receives no more than a constant number (usually one) of messages at a time.

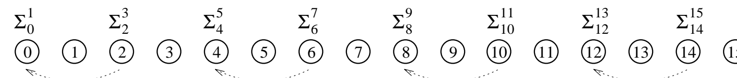
$$T_P = \Theta(\log n)$$

$$S = \Theta\left(\frac{n}{\log n}\right)$$

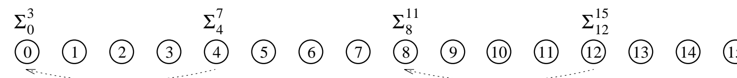
$$E = \Theta\left(\frac{1}{\log n}\right)$$



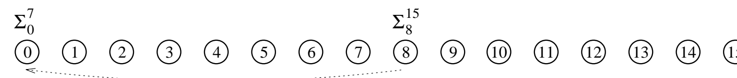
(a) Initial data distribution and the first communication step



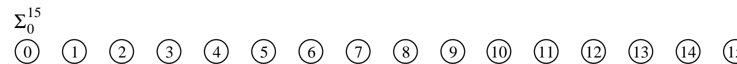
(b) Second communication step



(c) Third communication step



(d) Fourth communication step



(e) Accumulation of the sum at processor 0 after the final communication

Figure 4.1 Computing the sum of 16 numbers on a 16-processor hypercube. Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Scaling Down (Revisited)

Not quite so simple as for PRAM: need to think about where the data is now (can only efficiently emulate processors for which the data is locally available).

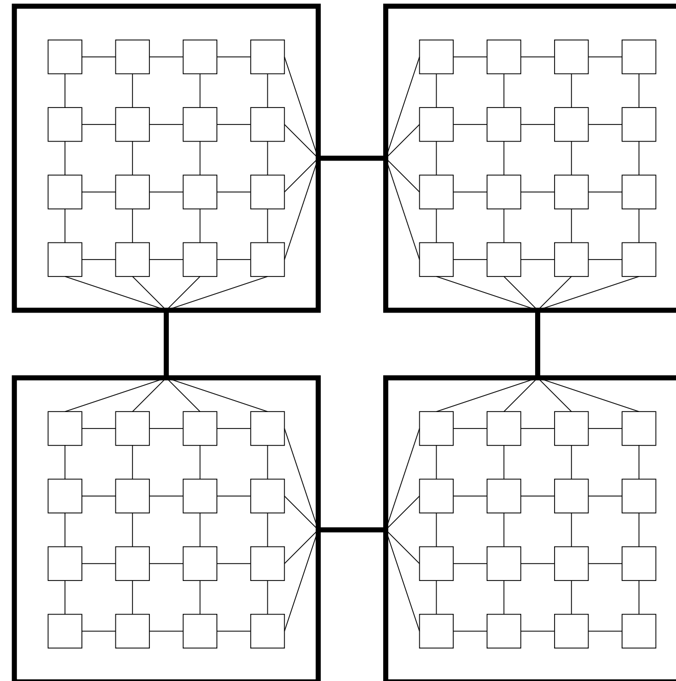
Quotient networks allow large instances to be embedded into smaller ones with

- balanced mapping of processors, with no edge dilation (edges map to at most single edges)
- 'round-robin' emulation with constant overhead (above the inevitable $\frac{p}{p'}$)

Thus, efficient algorithms for large p scale down to efficient algorithms for small p'.

Meshes and hypercubes are quotient networks.

Mesh



For hypercubes, remember the recursive construction.

Brent for Networks?

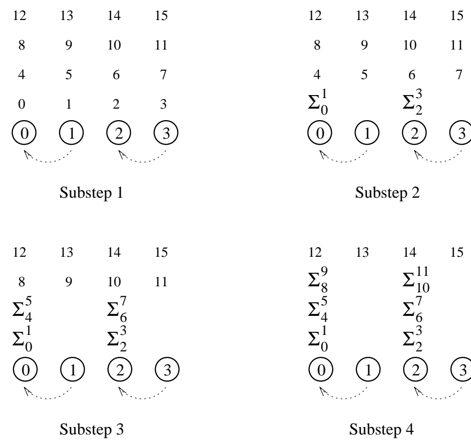
What if the large p algorithm isn't efficient?

Sometimes we can scale down to produce an efficient algorithm for smaller p' (saw this effect already for PRAM, via Brent's theorem)

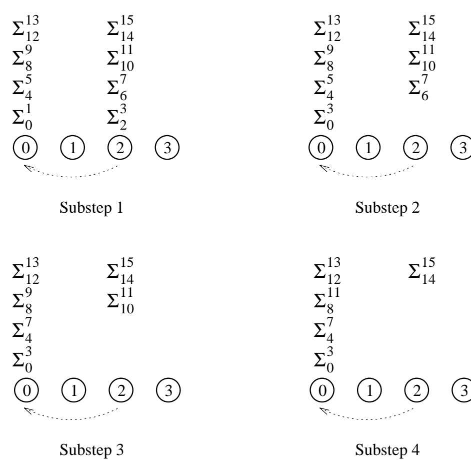
Data placement and efficiency of sequentialized components are now critical.

Consider adding n numbers on a p processor hypercube. We will see that the naive mapping is still inefficient but a smarter mapping is possible.

This doesn't always happen.



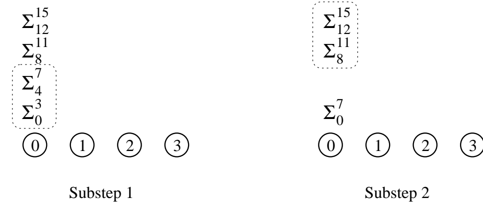
(a) Four processors simulating the first communication step of 16 processors



(b) Four processors simulating the second communication step of 16 processors

Figure 4.2 Four processors simulating 16 processors to compute the sum of 16 numbers (first two steps). Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.



(c) Simulation of the third step in two substeps

(d) Simulation of the fourth step

(e) Final result

Figure 4.3 (cont.) Four processors simulating 16 processors to compute the sum of 16 numbers (last three steps).

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

First $\log p$ steps take $\Theta\left(\frac{n}{p}\log p\right)$ time.

Remaining $\log n - \log p$ steps take $\Theta\left(\frac{n}{p}\right)$ time.

$C = \Theta(n \log p)$, not cost optimal.

Smarter scale down

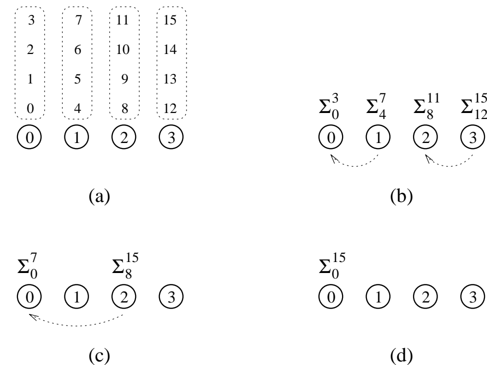


Figure 4.4 A cost-optimal way of computing the sum of 16 numbers on a four-processor hypercube. Copyright (r) 1994 Benjamin/Cummings Publishing Co.

First phase takes $\Theta\left(\frac{n}{p}\right)$ time and second phase takes $\Theta\left(\log p\right)$ time.

$C = \Theta\left(n + p\log p\right)$, which is cost optimal when $n = \Omega\left(p\log p\right)$.

Inter Model Emulation

Do we need to devise new algorithms for each message passing architecture? How about general purpose techniques for porting existing parallel algorithms?

We need to

- map processors to processors
- map links/memory to links/memory
- calculate overhead involved for each step and multiply run-time by this overhead

In special cases we make make optimisations which exploit specific aspects of a particular algorithm (e.g. frequency of usage of links).

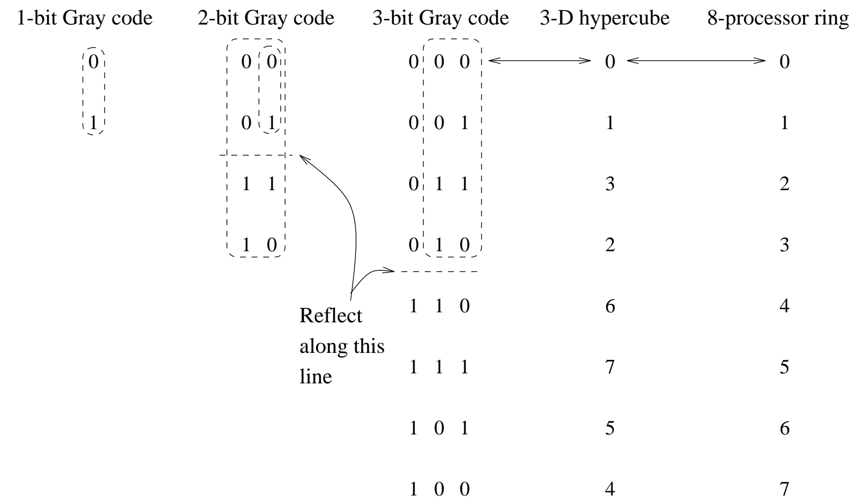
1-D Mesh (Array) to Hypercube

Array has 2^d procs indexed from 0 to $2^d - 1$.
Embed 1 to 1 into a d-dimensional hypercube, using binary reflected Gray code.

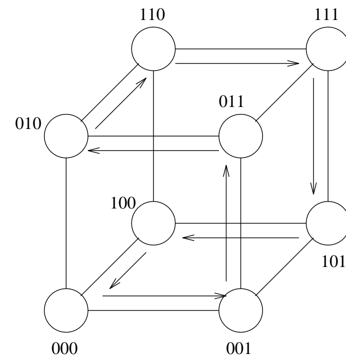
Array processor i maps to hypercube processor $G(i, d)$ where,

$$
\begin{aligned}
G(0,1) &= 0 \\
G(1,1) &= 1 \\
G(i, x+1) &= \begin{cases} G(i, x), & i < 2^x \\ 2^x + G(2^{x+1} - 1 - i, x), & i \geq 2^x \end{cases}
\end{aligned}
$$

Create table for $d+1$ from table for d by reflection and prefixing old half with 0, new half with 1.



(a)



(b)

Figure 2.22 A three-bit reflected Gray code ring (a) and its embedding into a three-dimensional hypercube (b).

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

2-D Mesh to Hypercube

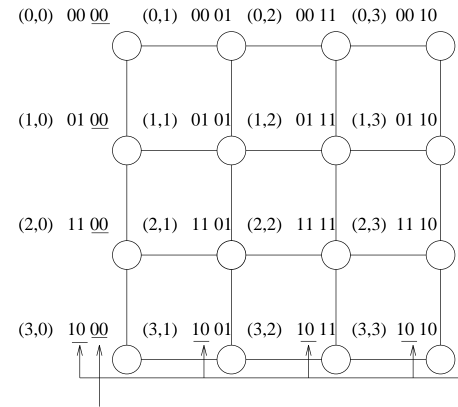
An extension of the 1-D technique.

Consider $2^r \times 2^s$ mesh in 2^{r+s} hypercube

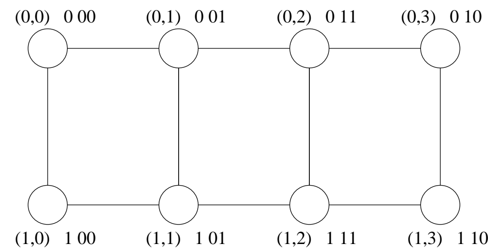
- proc (i, j) maps to proc $G(i, r)G(j, s)$ (i.e. the two codes concatenated)
- e.g. in a 2×4 mesh $(0, 1)$ maps to 001 etc

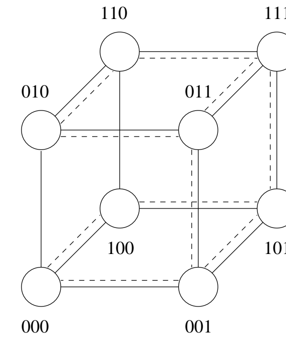
Fixing r bits in labels of $r + s$ dimensional cube produces an s dimensional subcube of 2^s processor, thus each row of the mesh maps to a distinct sub-cube, similarly with columns.

Useful when sub-algorithms work on rows/columns.



(a)





(b)

Figure 2.23 (a) A 4×4 mesh illustrating the mapping of mesh processors to processors in a four-dimensional hypercube; and (b) a 2×4 processor mesh embedded into a three-dimensional hypercube.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Designing Parallel Algorithms

There are no rules, only intuition, experience and imagination!

We consider design techniques, particularly top-down approaches, in which we find the main structure first, using a set of useful conceptual paradigms

We also look for useful primitives to compose in a bottom-up approach.

Designing Parallel Algorithms

A parallel algorithm emerges from a process in which a number of interlocked issues must be addressed:

- Where do the basic units of computation (tasks) come from? This is sometimes called "partitioning" or "decomposition". Sometimes it is natural to think in terms of partitioning the input or output data (or both). On other occasions a functional decomposition may be more appropriate (i.e. thinking in terms of a collection of distinct, interacting activities).

- How do the tasks interact? We have to consider the dependencies between tasks, perhaps as a DAG (directed acyclic graph). Dependencies will be expressed in implementations as communication, synchronisation and sharing (depending upon the machine model).

Designing Parallel Algorithms

- Are the natural tasks of a suitable granularity? Depending upon the machine, too many small tasks may incur high overheads in their interaction. Should they be agglomerated (collected together) into super-tasks? This is related to scaling-down.

- How should we assign tasks to processors? Again, in the presence of more tasks than processors, this is related to scaling down. The owner computes rule is natural for some algorithms which have been devised with a data-oriented partitioning. We need to ensure that tasks which interact can do so as efficiently (quickly) as possible.

These issues are often in tension with each other.

Divide & Conquer

Use recursive problem decomposition.

Create sub-problems of the same kind, which are solved recursively.

Combine sub-solutions to solve the original problem.

Define a base case for direct solution of simple instances.

Well-known examples include quicksort, mergesort, matrix multiply.

Parallelizing Divide & Conquer

There is an obvious tree of processes to be mapped to available processors.

There may be a sequential bottleneck at the root.

Producing an efficient algorithm may require us to parallelize it, producing nested parallelism.

Small problems may not be worth distributing - trade off between distribution costs and recomputation costs.

Pipelining

Modelled on a factory production line, applicable when some complex operation must be applied to a long sequence of data items.

Decompose operation into a sequence of p sub-operations and chain together processes for each sub-operation.

The single task completion time is at best the same (typically worse) but overall completion for n inputs is improved.

Essential that the sub-operation times are well balanced, in which case we can achieve nearly p fold speed-up for large n.

Step by Step Parallelization

Parallelizing sequential programs is difficult, because of many complex and even false dependencies.

Parallelizing sequential algorithms is sometimes easier

- consider coarse grain phases
- parallelize each phase
- keep sequential control flow between phases

Phases may sometimes be hard to parallelize (or even identify).

Amdahl's Law

If some fraction $0 \leq f \leq 1$ of a computation must be executed sequentially, then the speed up which can be obtained when the rest of the computation is executed in parallel, is bounded above $\frac{1}{f}$ irrespective of p.

Amdahl's Law

Proof: Call sequential run time T. Parallel version has sequential component fT and a parallel component $\geq \frac{(1-f)T}{p}$. Speed-up is then

$$
\begin{aligned}
& \frac{T}{fT+\frac{(1-f)T}{p}} \\
= \quad & \frac{1}{f+\frac{(1-f)}{p}} \\
& \longrightarrow \frac{1}{f} \; as \; p \longrightarrow \infty
\end{aligned}
$$

For example, if f is 0.1 then speed-up ≤ 10.

Useful Parallel Primitives

Support a bottom-up approach to algorithm design.

Identify a collection of common operations and devise fast parallel implementations for each architecture.

Design algorithms with these primitives as reusable components.

Corresponding programs benefit from heavily optimised implementations in a library.

Reduction and Parallel Prefix

Given values $x_1, x_2, x_3, ..., x_n$ and associative operator $\oplus$, reduction (for which we have already seen a fast PRAM implementation) computes

$$x_1 \oplus x_2 \oplus x_3 \oplus ... \oplus x_n$$

Similarly, prefix (scan) computes

$$x_1, x_1 \oplus x_2, x_1 \oplus x_2 \oplus x_3, ..., x_1 \oplus x_2 ... \oplus x_n$$

and the PRAM implementation is similar to that for reduction.

Hypercube prefix

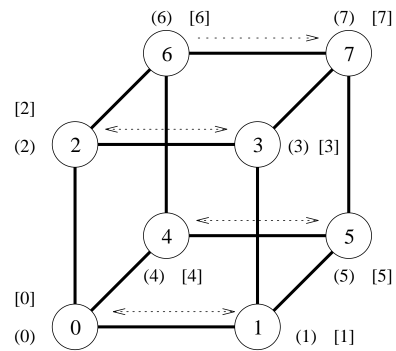
Assume $p = n$. Both data and results distributed one per processor.

```
1.  procedure PREFIX_SUMS_HCUBE(my_id, my_number, d, result)
2.  begin
3.      result := my_number;
4.      msg := result;
5.      for i := 0 to d do
6.      begin
7.          partner := my_id XOR 2^i;
8.          send msg to partner;
9.          receive number from partner;
10.         msg := msg + number;
11.         if (partner < my_id) then result := result + number;
12.     endfor;
13. end PREFIX_SUMS_HCUBE
```

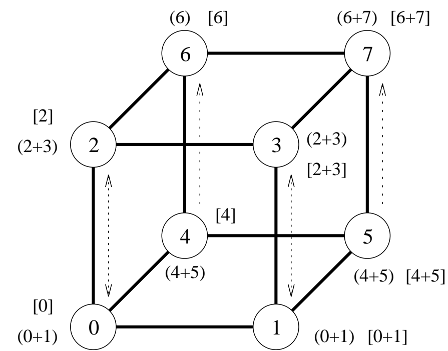
Program 3.7 Prefix sums on a d-dimensional hypercube.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

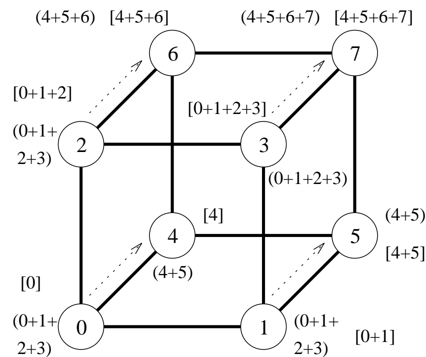
There are $\log n$ iterations and the run time depends upon size of communications and complexity of the operation.



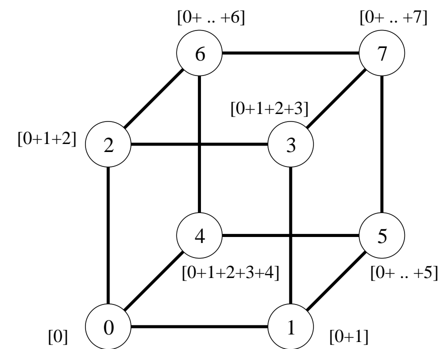
(a) Initial distribution of values



(b) Distribution of sums before second step



(c) Distribution of sums before third step



(d) Final distribution of prefix sums

Figure 3.13 Computing prefix sums on an eight-processor hypercube. At each processor, square brackets show the local prefix sum accumulated in a buffer and parentheses enclose the contents of the outgoing message buffer for the next step.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

(Fractional) Knapsack Problem

Given n objects with weights w_i and values v_i and a 'knapsack' of capacity c, find objects (or fractions of objects) which maximise knapsack value without exceeding capacity.

Assume value, weight distributed evenly through divided objects.

For example, given $\{(5, 100), (4, 20), (9, 135), (2, 26)\}$ (a set of objects as (w, v) pairs) and $c = 15$, choose all of $(5, 100)$ and $(9, 135)$ and half of $(2, 26)$ for a profit of 248.

There is a standard sequential algorithm.

```
sort by decreasing profitability v/w;
usedweight = 0;  i = 0;
while (usedweight < c) {
   if (usedweight + (sorteditems[i].w) < c) {
      include sorteditem[i];
   } else {
      include the fraction of sorteditem[i]
      which brings usedweight up to c;
   }
   i=i+1;
}
```

It has $\Theta(n \log n)$ run time (dominated by the sort), which we will parallelize step by step.

```
calculate profitabilities (independently in parallel);

sort in parallel;          // covered later

compute prefix sum of weights;
for each object independently in parallel
    if (myprefix <= c)
        object is completely included;
    else if (myprefix > c && left neighbour's prefix < c)
        an easily determined fraction of the object is included;
    else
        object not included;
```

Run time is sum of step times, e.g. $\Theta(\log n)$ time on n processor CREW PRAM.

Pointer Jumping

Suppose a PRAM has data in a linked list. We know the locations of objects, but not their ordering which is only implicitly known from the collected pointers

The last object has a null "next" pointer.

We can still do some operations fast in parallel, including prefix (!), by using pointer jumping.

For example, we can find each object's distance from the end of the list (a form of list ranking) in $\Theta(\log n)$ time, where a simplistic "follow the pointers" approach would take $\Theta(n)$ time.

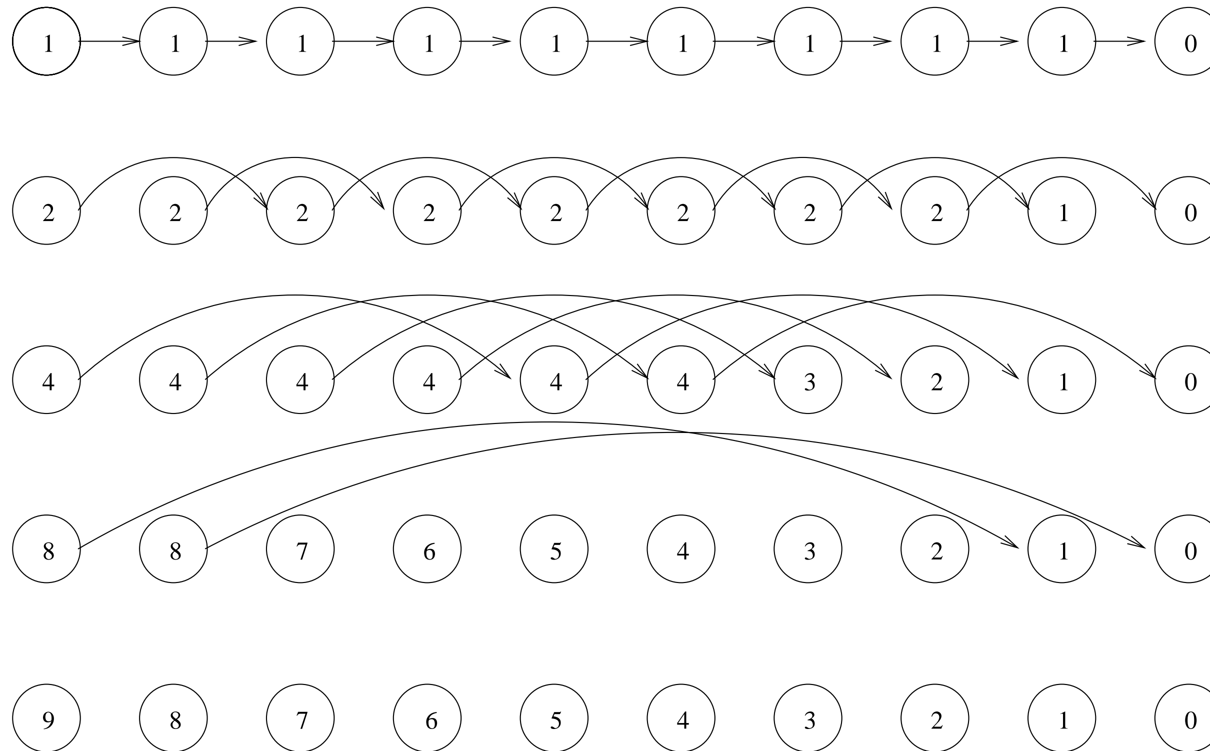
Pointer Jumping

```
for all objects in parallel {
  this.jump = this.next;          // copy the list

  if (this.jump == NULL) then this.d = 0;
  else this.d = 1;                // initialise

  while (this.jump != NULL) {
    this.d +=  (this.jump).d;     // accumulate
    this.jump = (this.jump).jump; // move on
  }
}
```

(assuming PRAM synchronization between statements of the loop)


1 1 1 1 1 1 1 1 1 0
2 2 2 2 2 2 2 2 1 0
4 4 4 4 4 4 3 2 1 0
8 8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0

PRAM List Prefix

We can similarly execute a prefix computation across the list, with an arbitrary associative operator.

In contrast to the ranking algorithm we now operate on our successor's value (rather than our own).

PRAM List Prefix

```
for all objects in parallel {
  this.jump = this.next; this.pf   = this.x;

  while (this.jump != NULL) {
    this.jump.pf = Op(this.pf, this.jump.pf);
    this.jump = (this.jump).jump;
  }
}
```

Communications Primitives

Many algorithms are based around a common set of collective communication operations, such broadcasts, scatters and gathers.

We can build these into a useful library, with fast implementations for each architecture. This gives our algorithms a degree of machine independence and avoids "reinventing the wheel".

For example, one-to-all broadcast has one process sending the same data (of size m) to all others.

A naive approach would send $p - 1$ individual messages, but we can do better.

Ring One-to-All Broadcast (SF)

Send message off in both directions, which receivers copy and forward.

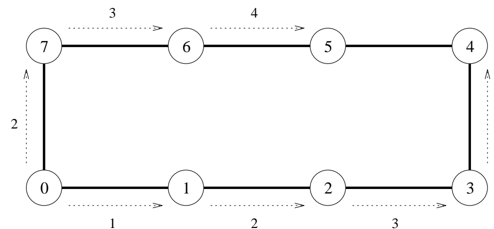


Figure 3.2 One-to-all broadcast on an eight-processor ring with SF routing. Processor 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during which the message is transferred.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

$$T_{one-to-all} = (t_s + t_w m)\lceil p/2 \rceil = \Theta\left(mp\right)$$

Ring One-to-All Broadcast (CT)

Use recursive doubling: one processor, then two, then four ..., choosing partners to avoid congestion.

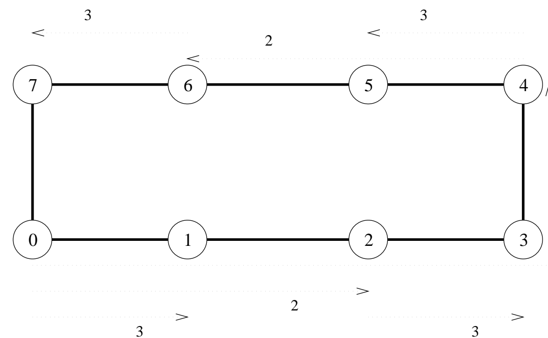


Figure 4.2 One-to-all broadcast on an eight-node ring. Node 0 is the source of the broadcast. Each message transfer step is shown by a numbered, dotted arrow from the source of the message to its destination. The number on an arrow indicates the time step during which the message is transferred.

$$T_{one-to-all} = \Theta\left(m \log p\right)$$

[NB: All our results from now on will be for SF routing]

2-D Mesh One-to-All Broadcast

First broadcast along source row then broadcast concurrently along each column.

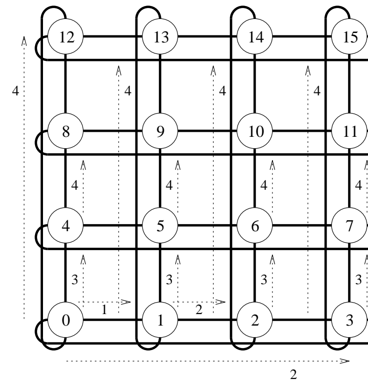


Figure 3.4 One-to-all broadcast on a 16-processor mesh with SF routing. Copyright (r) 1994 Benjamin/Cummings Publishing Co.

$T_{one-to-all} = 2(t_s + t_w m)\lceil \sqrt{p}/2 \rceil$ and similarly for higher dimensional meshes.

Hypercube One-to-All Broadcast

A 2^d processor hypercube is a d-dimensional mesh with 2 processors in each dimension, so we generalise from the 2-D mesh algorithm.

```
1.   procedure ONE_TO_ALL_BC(d, my_id, X)
2.   begin
3.       mask := 2^d - 1;              /* Set all d bits of mask to 1 */
4.       for i := d - 1 downto 0 do    /* Outer loop */
5.       begin
6.           mask := mask XOR 2^i;   /* Set bit i of mask to 0 */
7.           if (my_id AND mask) = 0 then
             /* If the lower i bits of my_id are 0 */
8.               if (my_id AND 2^i) = 0 then
9.               begin
10.                  msg_destination := my_id XOR 2^i;
11.                  send X to msg_destination;
12.              endif
13.              else
14.              begin
15.                  msg_source := my_id XOR 2^i;
16.                  receive X from msg_source;
17.              endelse;
18.      endfor;
19.  end ONE_TO_ALL_BC
```

Program 3.1 One-to-all broadcast of a message X from processor 0 of a d-dimensional hypercube. AND and XOR are bitwise logical-and and exclusive-or operations, respectively.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

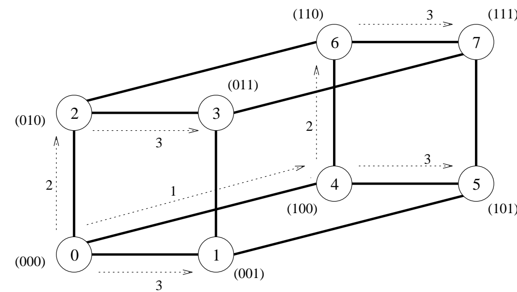


Figure 3.5 One-to-all broadcast on a three-dimensional hypercube. The binary representations of processor labels are shown in parentheses.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

This works when the source is process 0. See Kumar for generalisation to any source.

Execution time is $T_{one-to-all} = (t_s + t_w m)\log p$

All-to-All Broadcast

Every processor has its own message of size m to broadcast.

The challenge is to beat the obvious sequence of p one-to-all broadcasts.

For a ring algorithm, we note and exploit unused links in one-to-all case.

This gives a pipelined effect (but more complex) with $p-1$ steps.

$T_{all-to-all} = (t_s + t_w m)(p-1)$ which is twice the time for a single one-to-all.

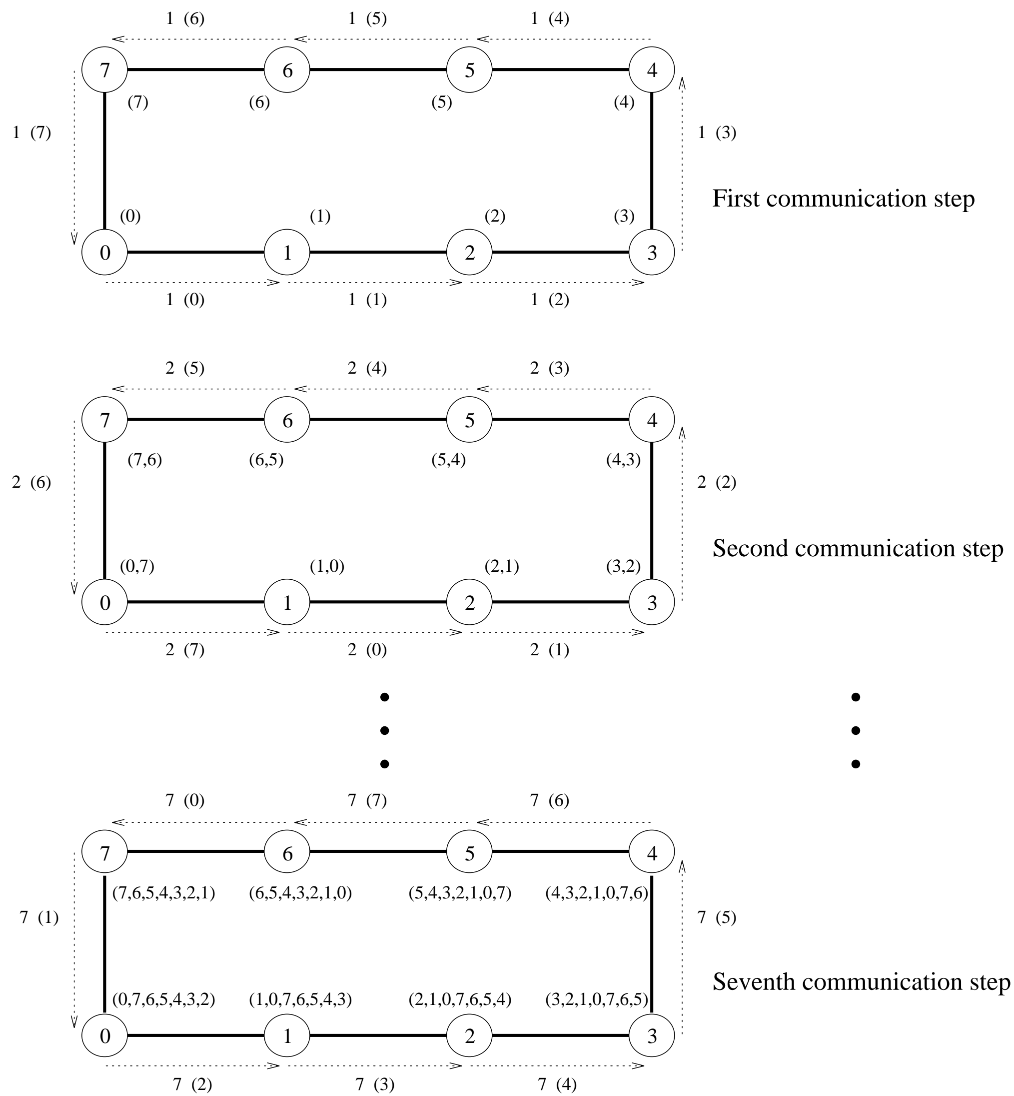


Figure 3.10 All-to-all broadcast on an eight-processor ring with SF routing. In addition to the time step, the label of each arrow has an additional number in parentheses. This number labels a message and indicates the processor from which the message originated in the first step. The number(s) in parentheses next to each processor are the labels of processors from which data has been received prior to the communication step. Only the first, second, and last communication steps are shown.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Mesh/Hypercube All-to-All Broadcast

First mesh phase, independent ring all-to-all in rows in time $(t_s + t_w m)(\sqrt{p} - 1)$.

Second mesh phase is similar in columns, message size $m\sqrt{p}$, giving time $(t_s + t_w m\sqrt{p})(\sqrt{p} - 1)$.

Total time is $2t_s(\sqrt{p} - 1) + t_w m(p - 1)$, which is roughly a factor of $\sqrt{p}$ worse than one-to-all. This is an asymptotic lower bound because of link capacity.

Hypercube version generalizes, with message size $2^{i-1}m$ in the ith step of d.

$$\begin{aligned} T_{all-to-all} &= \sum_{i=1}^{\log p}(t_s + 2^{i-1}t_w m) \\ &= t_s \log p + t_w m(p - 1) \end{aligned}$$

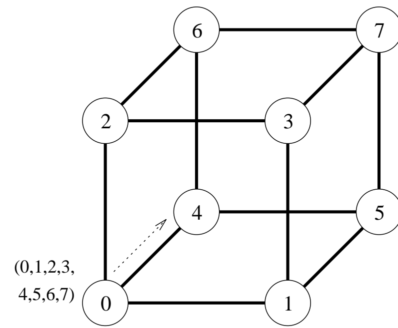
Scatter

Sometimes called one-to-all personalised communication, we have a source processor with a distinct item for every processor.

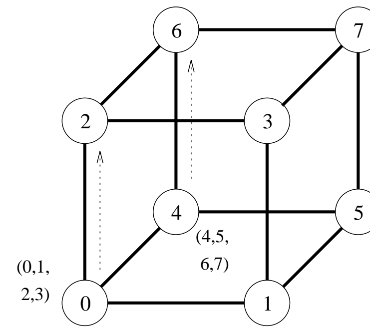
We will need $\Omega\left(t_w m(p-1)\right)$ time to get data off source (a lower bound).

Algorithms have the familiar structure

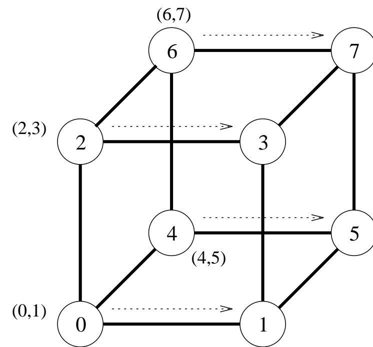
- ring in time $(t_s + t_w m)(p-1)$
- mesh in time $2t_s(\sqrt{p}-1) + t_w m(p-1)$
- hypercube in time $t_s \log p + t_w m(p-1)$



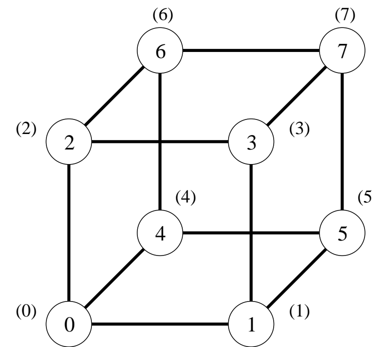
(a) Initial distribution of messages



(b) Distribution before the second step



(c) Distribution before the third step



(d) Final distribution of messages

Figure 3.16 One-to-all personalized communication on an eight-processor hypercube.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Summary

	One − All BC	All − All BC	Personal
Ring	$\Theta(mp)$	$\Theta(mp)$	$\Theta(mp)$
Mesh	$\Theta(m\sqrt{p})$	$\Theta(mp)$	$\Theta(mp)$
Hypercube	$\Theta(m \log p)$	$\Theta(mp)$	$\Theta(mp)$

Remember, this is all for SF routing - the book also discusses CT routing, so be careful to check the context when reading!

Sorting

A simple (but not simplistic), widely applicable problem which allows us to see several of our techniques in action.

Given $S = (a_1, a_2, ..., a_n)$ produce $S' = (a'_1, a'_2, ..., a'_n)$ such that $a'_1 <= a'_2 <= ..... <= a'_n$ and S' is a permutation of S.

We consider only integers, but techniques are generic.

Sequential sorting is $\Omega\left(n \log n\right)$ time.

Sequential Sorting Algorithms

In enumeration sort we work out final positions (ranks) first then move data. Takes $\Theta\left(n^2\right)$ time.

Mergesort is a well-known divide & conquer algorithm with all work in the combine step. Overall it takes $\Theta\left(n \log n\right)$ time.

Quicksort is another divide & conquer algorithm, but with all the work in the divide step. They key idea is to partition data into 'smaller', 'larger' and items with respect a pivot value. Choice of pivot is crucial. Average case run time is $\Theta\left(n \log n\right)$, The worst case $O\left(n^2\right)$ but randomized pivot choice gives 'almost always' good behaviour.

PRAM Algorithms

We have already seen a constant time n^2 processor CRCW algorithm, but this is only efficient with respect to enumeration sort.

In CREW Mergesort, we parallelize the divide-and-conquer structure, and parallelize the merge to avoid a $\Theta(n)$ bottleneck.

We borrow from enumeration sort, noting that rank in a merged sequence $=$ own rank $+$ rank with respect to other sequence.

We compute "other sequence" rank with binary search, where concurrent searches require the CREW PRAM model.

We have to make minor adjustments if we want to allow for the possibility of duplicate elements.

Analysis of CREW Mergesort

Standard binary search is $O(\log s)$ in s items, and using one processor per item, the ith step merges sequences of length 2^{i-1}.

Summing across $\log n$ steps gives $\Theta\left(\log^2 n\right)$ run time, which is not cost optimal.

A more complex merge can use $O\left(\frac{n}{\log n}\right)$ processors (sorting sequentially first) for $\Theta\left(\frac{n}{p}\log n + \log^2 n\right)$ time, which is cost optimal.

School of informatics

CRCW Quicksort - A Tree of Pivots

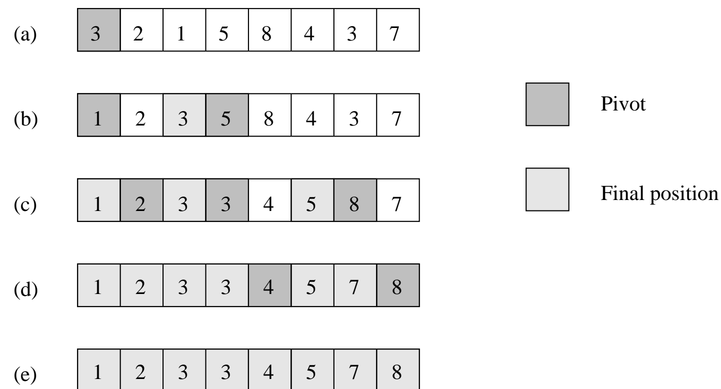


Figure 6.15 Example of the quicksort algorithm sorting a sequence of size $n = 8$.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

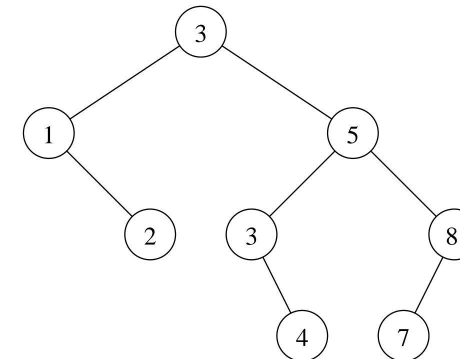


Figure 6.16 A binary tree generated by the execution of the quicksort algorithm. Each level of the tree represents a different array-partitioning iteration. If pivot selection is optimal, then the height of the tree is $\Theta(\log n)$, which is also the number of iterations.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

In CRCW (arbitrary resolution) quicksort we construct the pivot tree, which is then used to compute final positions (ranks), which are used to move each item to its correct position.

We describe tree structure in terms of array indices, with a shared variable root (of the whole tree), arrays leftchild and rightchild and a local variable parent for each item.

Arbitrary write mechanism randomizes pivot choice, duplicate values are split between sides to balance workload.

Average tree depth (and so run time) is $\Theta(\log n)$.

```
1.    procedure BUILD_TREE (A[1...n])
2.    begin
3.        for each processor i do
4.        begin
5.            root := i;
6.            parent_i := root;
7.            leftchild[i] := rightchild[i] := n + 1;
8.        end for
9.        repeat for each processor i ≠ root do
10.       begin
11.           if (A[i] < A[parent_i]) or
                  (A[i] = A[parent_i] and i <parent_i) then
12.           begin
13.               leftchild[parent_i] := i;
14.               if i = leftchild[parent_i] then exit
15.               else parent_i := leftchild[parent_i];
16.           end for
17.           else
18.           begin
19.               rightchild[parent_i] := i;
20.               if i = rightchild[parent_i] then exit
21.               else parent_i := rightchild[parent_i];
22.           end else
23.       end repeat
24.   end BUILD_TREE
```

Program 6.6 The binary tree construction procedure for the CRCW PRAM parallel quicksort formulation.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

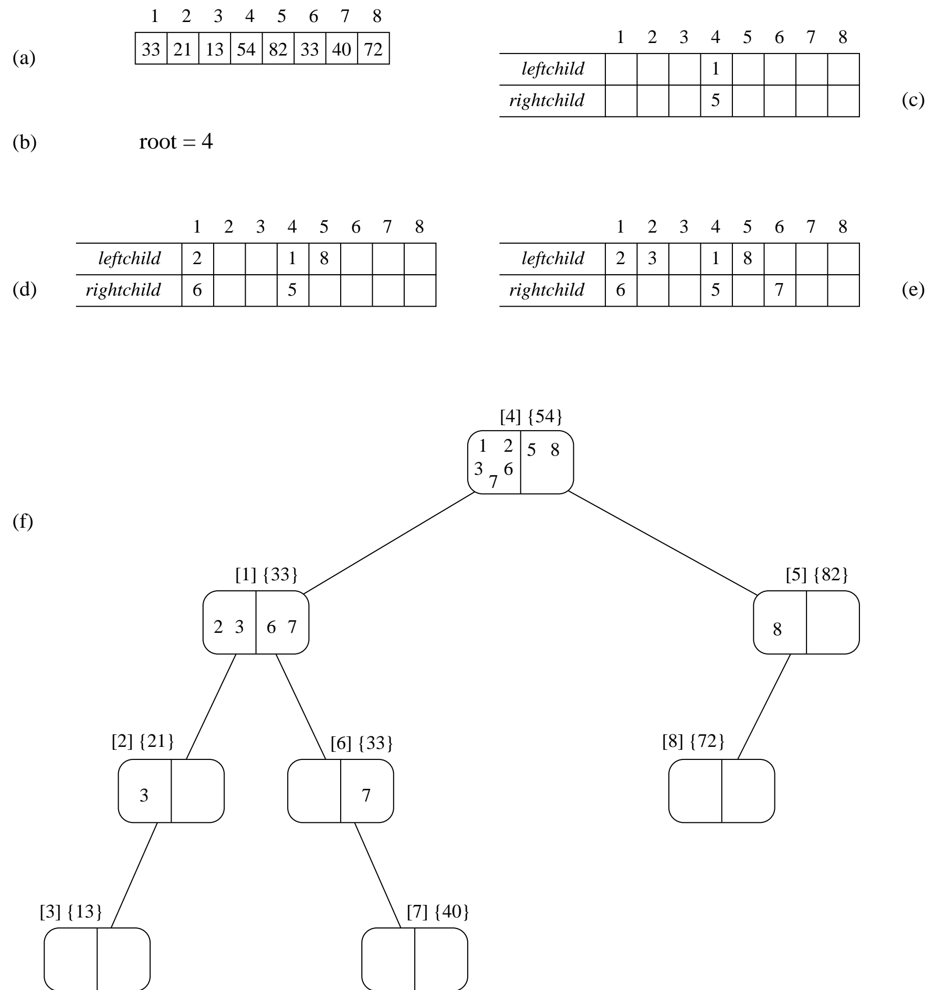


Figure 6.17 The execution of the PRAM algorithm on the array shown in (a). The arrays *leftchild* and *rightchild* are shown in (c), (d), and (e) as the algorithm progresses. Figure (f) shows the binary tree constructed by the algorithm. Each node is labeled by the processor (in square brackets), and the element is stored at that processor (in curly brackets). The element is the pivot. In each node, processors with smaller elements than the pivot are grouped on the left side of the node, and those with larger elements are grouped on the right side. These two groups form the two partitions of the original array. For each partition, a pivot element is selected at random from the two groups that form the children of the node.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Given the tree, we compute the ranks in two steps:

1. Compute the size of (number of nodes in) each sub-tree. For node `i`, store the size of its sub-trees in `leftsize[i]` and `rightsize[i]`

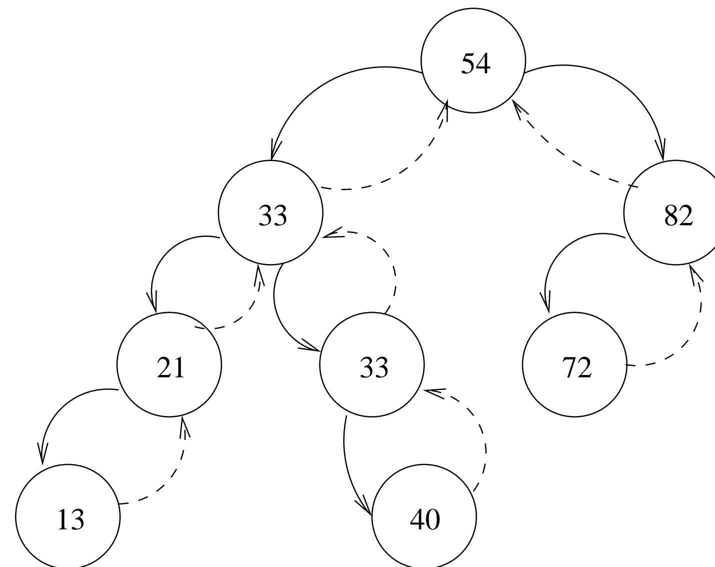
2. Compute the rank for a node with the following sweep down the tree, from root to leaves

This takes $\Theta\left(tree\ depth\right)$ so expected $\Theta\left(\log n\right)$.

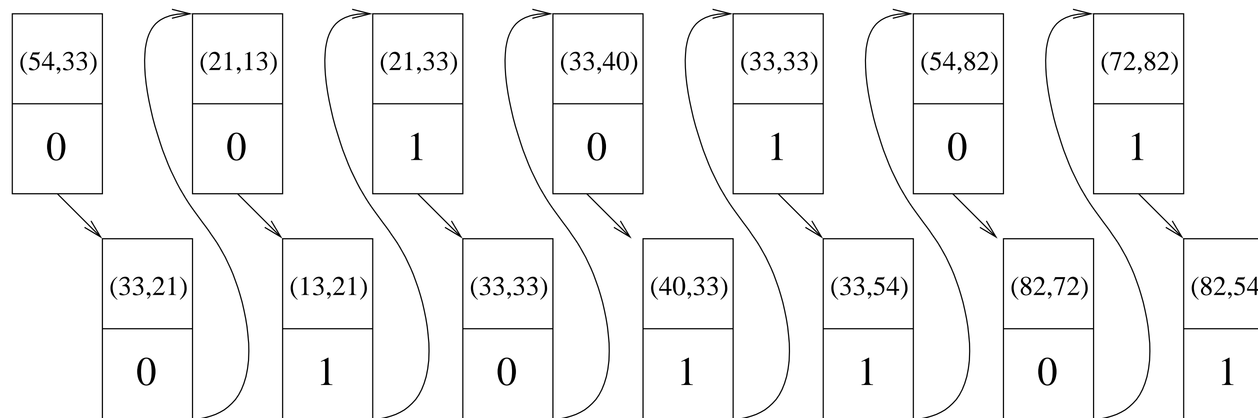
```
for each node n in parallel do rank[n] = 0;
rank[root] = leftsize[root];
for each node n != root in parallel repeat {
  parentrank = rank[parent];
  if parentrank != 0 {
   if n is a left child {
     rank[n] = parentrank - rightsize[n] - 1;
   } else {
     rank[n] = parentrank + leftsize[n]  + 1;
   }
   exit;
  }
}
```

Computing sub-tree sizes

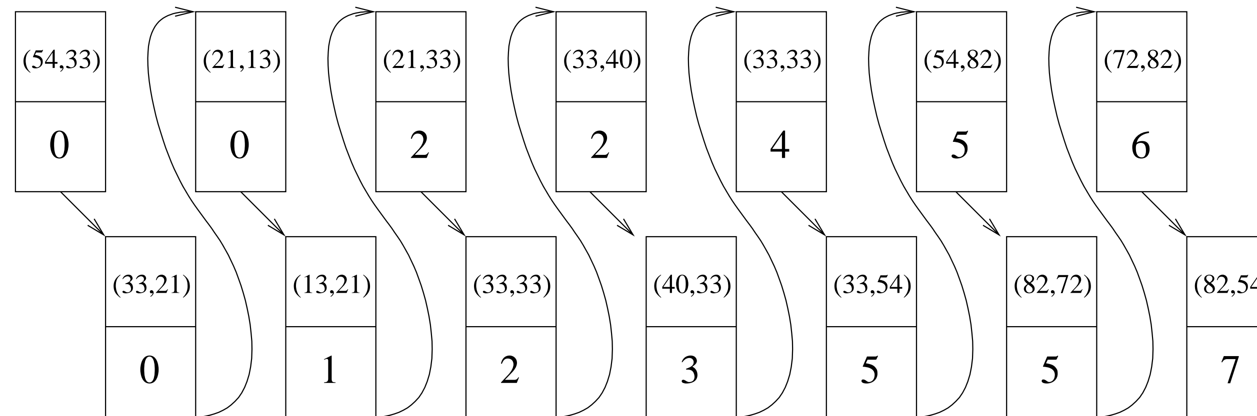
Think of the each edge in the tree as two edges, one down and one up.



Think of the sequences of edges used by a depth first tree traversal as a list, additionally labelled 0 for down edges and 1 for up edges.



The size a (sub)tree is the number of up edges in the list between the down edge which first enters the (sub)tree and (inclusive of) the up edge which finally leaves it. We can compute this by performing a prefix with addition of the list then subtracting the count on entry to a node from the count on leaving it.



For example

- the upper node containing 33 roots a subtree of size 5 (ie. $5 - 0$)
- the node containing 82 roots a subtree of size 2 (ie. $7 - 5$)

The root is an easy special case (final value in prefix $+ 1$).

Parallel Sorting with Message Passing

For message passing algorithms there is an extra issue of how to define the problem. What does it mean to have data sorted when there is more than one memory?

Our assumption will be that each processor initially stores a fair share ($\frac{n}{p}$) of the items, and that sorting means rearranging these so that each processor still has a fair share but with the smallest $\frac{n}{p}$ stored in sorted order by processor 0 etc.

The diameter of the underlying network defines a lower bound on time for $n = p$.

Odd-even transposition sort

A variant of sequential bubblesort mapped to the 1-D array which performs a number of non-overlapping neighbour swaps in parallel using a pairwise compare-exchange step as the building block.

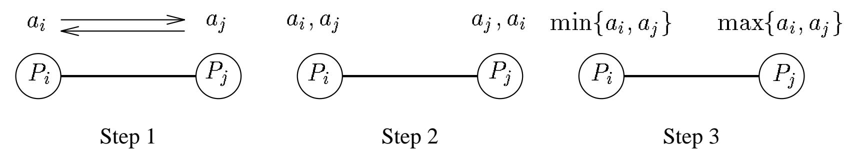


Figure 6.1 A parallel compare-exchange operation. Processors P_i and P_j send their elements to each other. Processor P_i keeps $\min\{a_i, a_j\}$, and P_j keeps $\max\{a_i, a_j\}$.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Alternate the positioning of the swap windows to allow progress (i.e. so that items can move in the right direction).

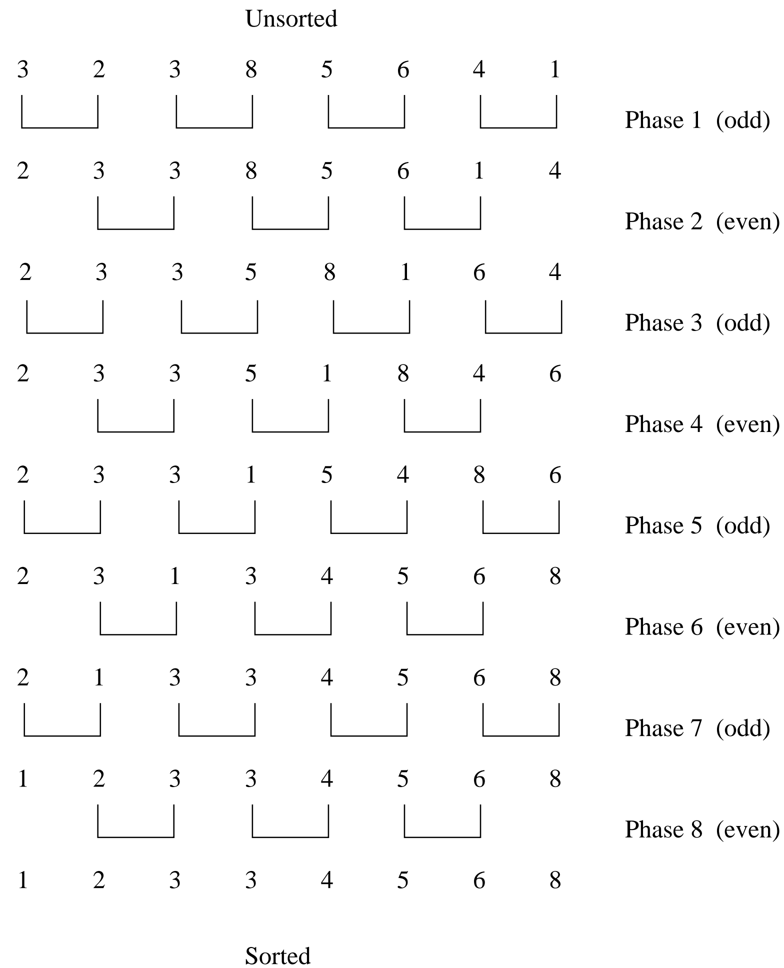


Figure 6.13 Sorting $n = 8$ elements, using the odd-even transposition sort algorithm. During each phase, $n = 8$ elements are compared.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

```
1.  procedure ODD-EVEN_PAR(n)
2.  begin
3.      id := processor's label
4.      for i := 1 to n do
5.      begin
6.          if i is odd then
7.              if id is odd then
8.                  compare-exchange_min(id + 1);
9.              else
10.                 compare-exchange_max(id − 1);
11.         if i is even then
12.             if id is even then
13.                 compare-exchange_min(id + 1);
14.             else
15.                 compare-exchange_max(id − 1);
16.     end for
17. end ODD-EVEN_PAR
```

Program 6.4 The parallel formulation of odd-even transposition sort on an n-processor ring.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Analysis

Time is $\Theta(n)$ on n processors, which is optimal for the architecture, but not cost optimal.

Using $p < n$ processors

- each processor sorts $\frac{n}{p}$ items sequentially
- p iterations, replacing compare-exchanges with compare-split, costing $\Theta\left(\frac{n}{p}\right)$ time each

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

To ensure asymptotic cost-optimality we will need $p = O(\log n)$ (ie $n = \Omega(2^p)$).

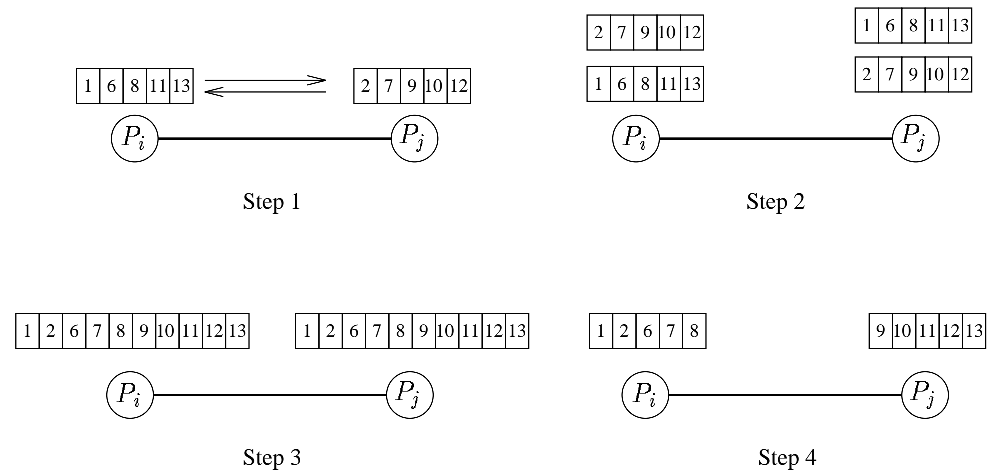


Figure 6.2 A compare-split operation. Each processor sends its block of size n/p to the other processor. Each processor merges the received block with its own block and retains only the appropriate half of the merged block. In this example, processor P_i retains the smaller elements and processor P_j retains the larger elements.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Bitonic Mergesort

A sorting network is a connected collection of comparators, each of which outputs its two inputs in ascending (or descending) order.

A sorting network is sequence of interconnected columns of such comparators.

The run time of a sorting network is proportional to number of columns, sometimes called its depth.

We could implement such algorithms directly in VLSI, or use them as a basis for parallel algorithms for conventional architectures.

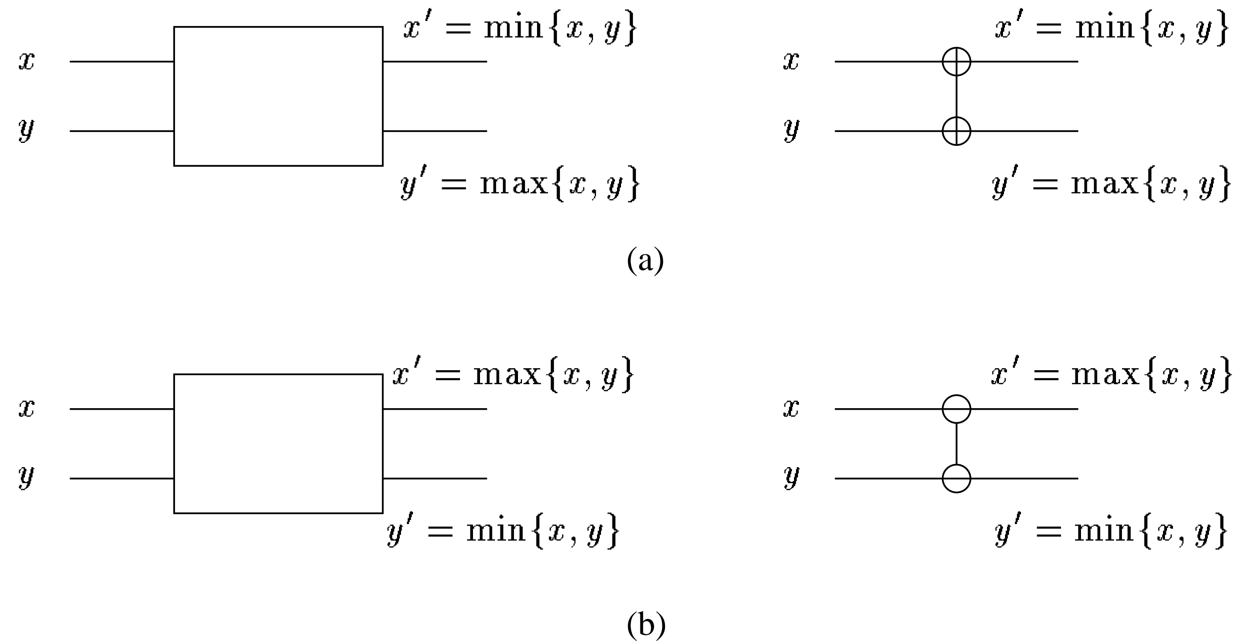


Figure 6.3 A schematic representation of comparators: (a) an increasing comparator, and (b) a decreasing comparator.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

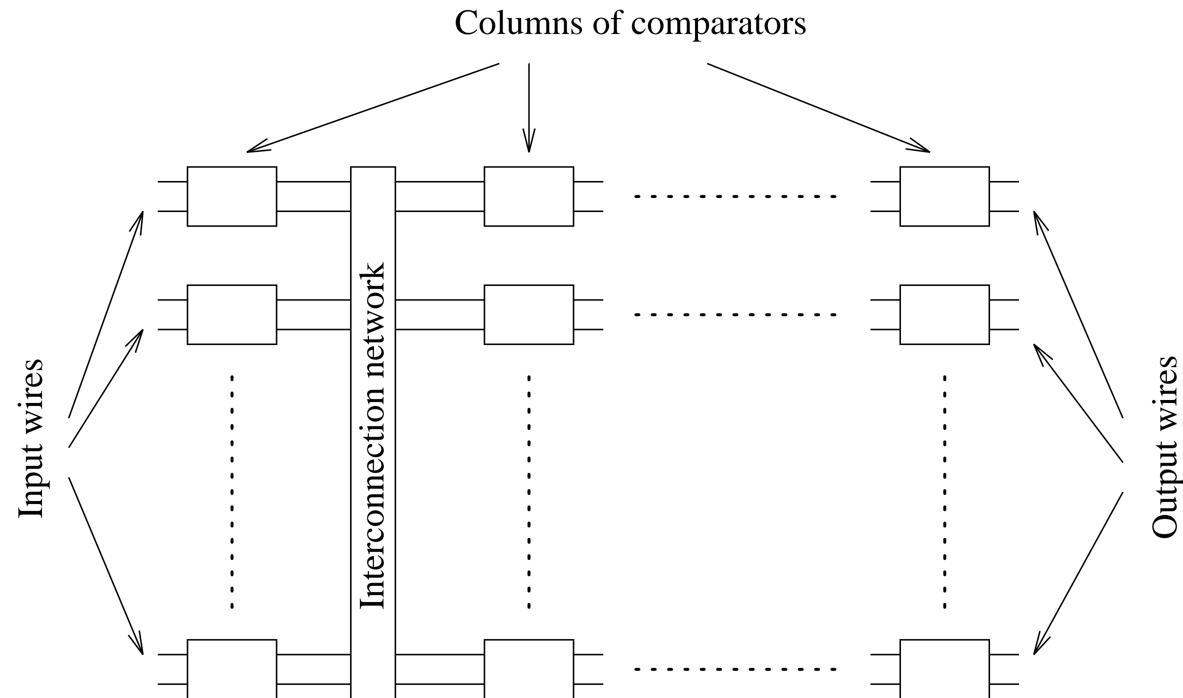


Figure 6.4 A typical sorting network. Every sorting network is made up of a series of columns, and each column contains a number of comparators connected in parallel.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

A sequence $a_0, a_1, ..., a_{n-1}$ is bitonic if there is i, $0 \leq i \leq n-1$ such that

- $a_0..a_i$ is monotonically increasing
- $a_{i+1}..a_{n-1}$ is monotonically decreasing

or it can be shifted cyclically to be so.

For example, sequences 1, 2, 4, 7, 6, 0 and 8, 9, 2, 1, 0, 4 are bitonic.

For simplicity, we now argue with n a power of 2, and $i = \frac{n}{2}$, but this applies for any bitonic sequence.

Consider the sequences

$min(a_0, a_{n/2}), min(a_1, a_{n/2+1}), ...min(a_{n/2-1}, a_{n-1})$

$max(a_0, a_{n/2}), max(a_1, a_{n/2+1}), ...max(a_{n/2-1}, a_{n-1})$

Both sequences are bitonic and all items in the 'min' sequence are $\leq$ all items in the 'max' sequence.

Recurse independently on each sequence until we have n 'bitonic' sequences of length 1, in ascending order (i.e. sorted!)

We will need $\log n$ such levels.

Original sequence	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
1st Split	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
2nd Split	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3rd Split	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
4th Split	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

Figure 6.5 Merging a 16-element bitonic sequence through a series of $\log 16$ bitonic splits.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

School of informatics

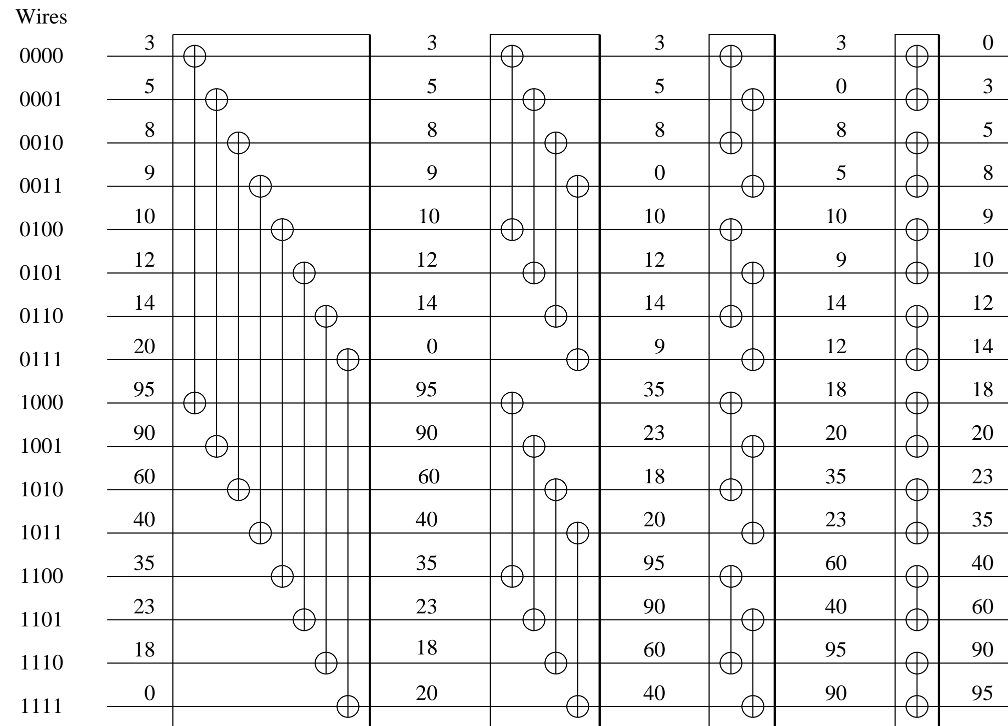


Figure 6.6 A bitonic merging network for $n = 16$. The input wires are numbered $0, 1 \ldots, n-1$, and the binary representation of these numbers is shown. Each column of comparators is drawn separately; the entire figure represents a $\oplus$BM[16] bitonic merging network. The network takes a bitonic sequence and outputs it in sorted order.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

To sort, create length $\frac{n}{2}$ ascending-descending sequences, then feed these to a bitonic merger.

We can create the required smaller sequences by sorting (bitonically).

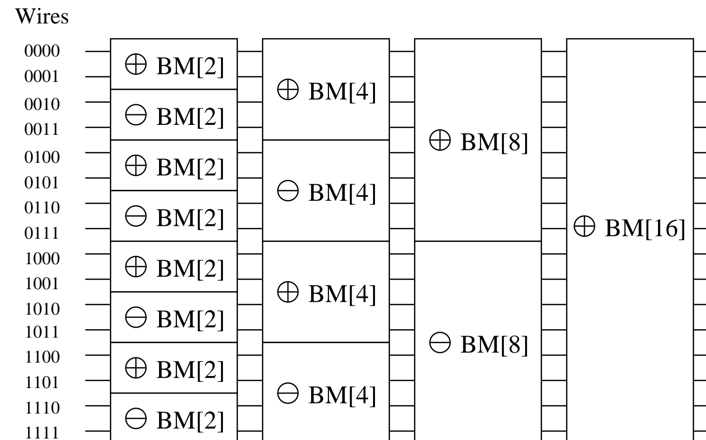


Figure 6.7 A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example, ⊕BM[k] and ⊖BM[k] denote bitonic merging networks of input size k that use ⊕ and ⊖ comparators, respectively. The last merging network (⊕BM[16]) sorts the input. In this example, $n = 16$.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

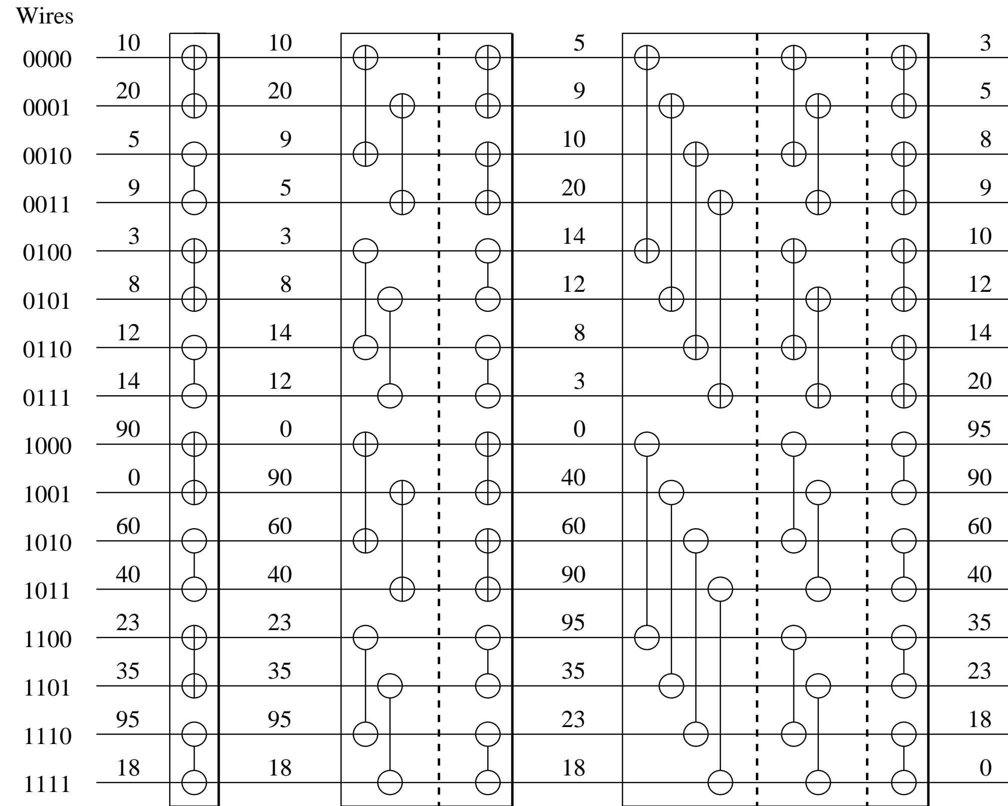


Figure 6.8 The comparator network that transforms an input sequence of 16 unordered numbers into a bitonic sequence. In contrast to Figure 6.6, the columns of comparators in each bitonic merging network are drawn in a single box, separated by a dashed line.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

We now consider the algorithm's circuit depth (run-time), $d(n)$.

The final bitonic merge has depth $\log n$ and is preceded by a complete bitonic sorter for $\frac{n}{2}$ items (actually two running one above the other in parallel), giving a recurrence for depth of

$$\begin{aligned} d(n) &= d(\tfrac{n}{2}) + \log n \\ d(2) &= 1 \end{aligned}$$

which has solution $\frac{(\log^2 n + \log n)}{2} = \Theta\left(\log^2 n\right)$.

Notice that the algorithm is pipelineable. We could use it as a sorting algorithm for an unusual parallel machine, or map it to more conventional machines.

Mapping Bitonic Sort to Other Networks

First consider the simple case with $n = p$ so we need to map wires 1-1 to processors.

The columns are $\frac{n}{2}$ pairs of compare-exchanges and we map to make these as local as possible.

Notice that wires involved always differ in exactly one bit in binary representation.

In the algorithm, wires differing in i^{th} least significant bit are involved in $\log n - i + 1$ steps.

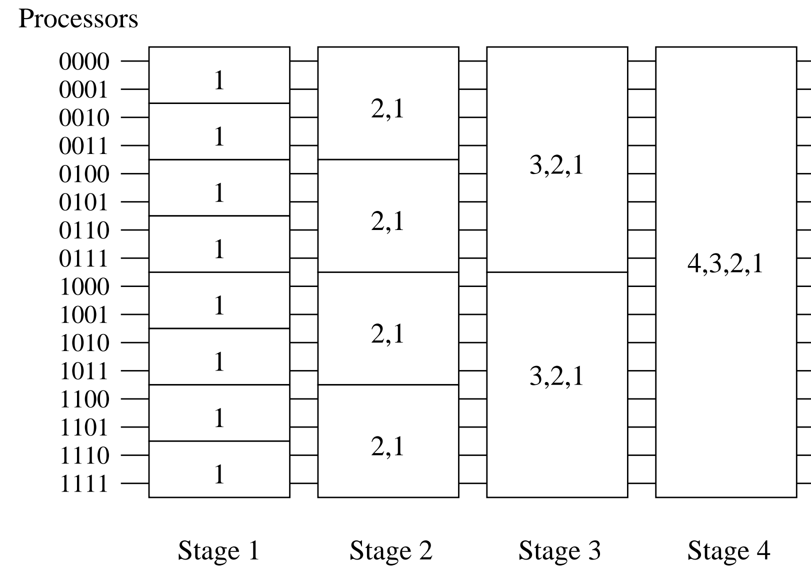


Figure 6.10 Communication characteristics of bitonic sort on a hypercube. During each stage of the algorithm, processors communicate along the dimensions shown.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Mapping to the Hypercube

The pairs of processors involved are all directly connected. Notice that stages of the algorithm map to larger and larger subcubes.

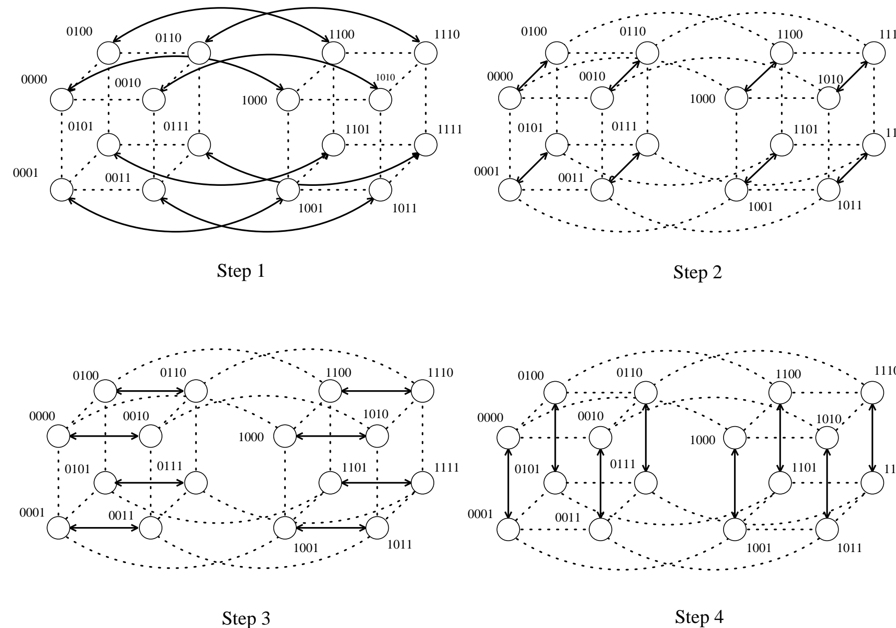


Figure 6.9 Communication during the last stage of bitonic sort. Each wire is mapped to a hypercube processor; each connection represents a compare-exchange between processors.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

This perfect mapping means that run-time is unaffected, $\Theta\left(\log^2 n\right)$.

The tricky part in implementation is to get the 'polarity' of the comparators right.

```
1.  procedure BITONIC_SORT(label, d)
2.  begin
3.      for i := 0 to d − 1 do
4.          for j := i downto 0 do
5.              if (i + 1)st bit of label ≠ jth bit of label then
6.                  comp_exchange_max(j);
7.              else
8.                  comp_exchange_min(j);
9.  end BITONIC_SORT
```

Program 6.1 Parallel formulation of bitonic sort on a hypercube with $n = 2^d$ processors. In this algorithm, $label$ is the processor's label and d is the dimension of the hypercube.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Mapping to the 2-D Mesh

The required wire pairs can't all be directly connected (consider the degree of the nodes).

We should map more frequently paired wires to shorter paths.

We could consider various labelling schemes, but want wires differing in bit i to be mapped to processors which are at least as close physically as those to which wires differing in bit $i+1$ are mapped.

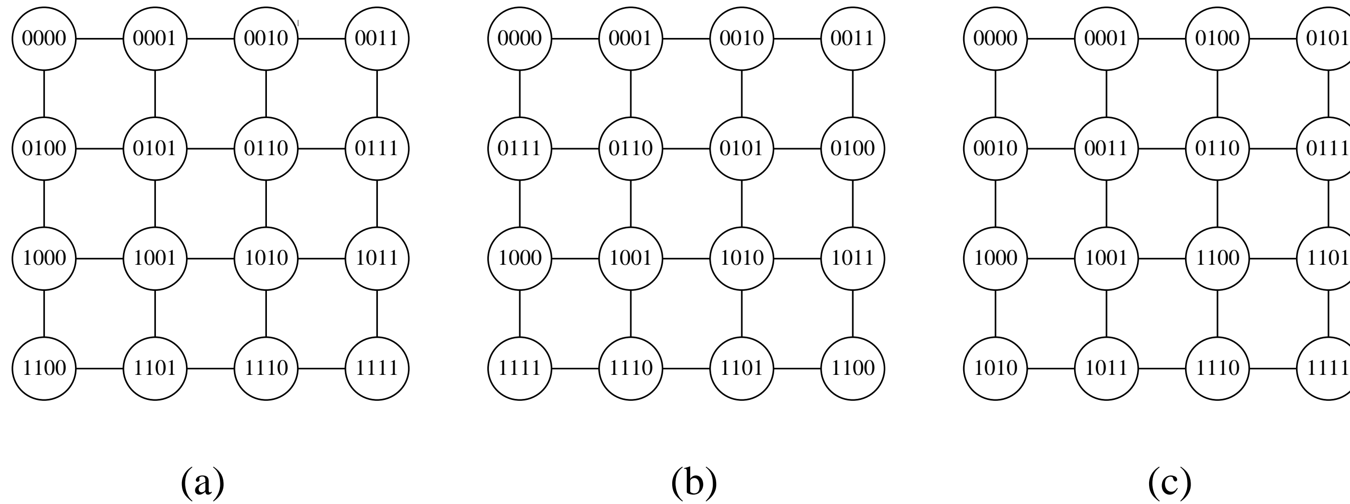


Figure 6.11 Different ways of mapping the input wires of the bitonic sorting network to a mesh of processors: (a) row-major mapping, (b) row-major snakelike mapping, and (c) row-major shuffled mapping.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Wire to Mesh Mappings

We want to map $n = 2^d$ wires to a $2^{\frac{d}{2}} \times 2^{\frac{d}{2}}$ $(\sqrt{n} \times \sqrt{n})$ processor mesh.

Mesh processor (r, c) in row $r = r_{\frac{d}{2}-1}r_{\frac{d}{2}-2}...r_1r_0$ and column $c = c_{\frac{d}{2}-1}c_{\frac{d}{2}-2}...c_1c_0$ is physically closest to processors sitting in adjacent rows and columns (i.e. those for which the higher order bits of r and c are the same, and which differ in the low order bits of r and/or c).

Row-major mapping assigns wire with identifier $w_{d-1}w_{d-2}...w_1w_0$ to processor $(w_{d-1}w_{d-2}...w_{\frac{d}{2}},\quad w_{\frac{d}{2}-1}...w_1w_0)$ (top bits become the row index, bottom bits are the column index), which tends to map wires close to other wires which differ from them in the low order bits $(..w_1w_0)$, and/or the middle order bits $(...w_{\frac{d}{2}+1}w_{\frac{d}{2}})$.

Row-major Shuffled Indexing

"we want wires differing in bit i to be mapped to processors which are at least as close physically as those to which wires differing in bit $i+1$ are mapped."

In contrast, shuffled row-major mapping assigns wire $w_{d-1}w_{d-2}...w_1w_0$ to processor $(w_{d-1}w_{d-3}...w_5w_3w_1, \quad w_{d-2}w_{d-4}...w_4w_2w_0)$

(as though the mesh process row and column bits have been shuffled to create the wire identifier)

In other words, the low order bits of the wire identifier are used to create the low order bits of the row and column identifiers and so a process is mapped closer to its more frequent partners in the algorithm. In general, wires differing in bit i are mapped to processors $2^{\lfloor \frac{i}{2} \rfloor}$ links apart.

Analysis

Computation still costs $\Theta\left(\log^2 n\right)$ time, while communications cost

$$
\begin{aligned}
& (t_s + t_w) \sum_{i=0}^{\log n - 1} \sum_{j=0}^{i} 2^{\lfloor \frac{j}{2} \rfloor} \\
= \quad & \Theta\left(\sqrt{n}\right)
\end{aligned}
$$

[This is quite hard to show - I won't expect you to do so]

Overall run time is $\Theta\left(\sqrt{n}\right)$, which is not cost optimal, but is optimal for the architecture.

Useful fact: data can be routed to any other 'reasonable' indexing scheme in $\Theta\left(\sqrt{n}\right)$ time.

Block based variants

As for odd-even transposition sort, we can use the same algorithm but work with compare-split for compare-exchange, and introducing an initial sequential sort of each $\frac{n}{p}$ item block.

The analysis is identical in terms of stages, but with different costs for communications and computation at each node and the addition of the initial local sorts.

Dense Matrix Algorithms

We will deal with dense, square $n \times n$ matrices, in which there are 'not enough zeros to exploit systematically'.

Generalizations to rectangular matrices are straightforward but messy.

For message passing algorithms, data partitioning (distributing matrices across processors) is a key issue.

Standard approaches are to distribute by block or cyclically using rows, column, or checkerboard.

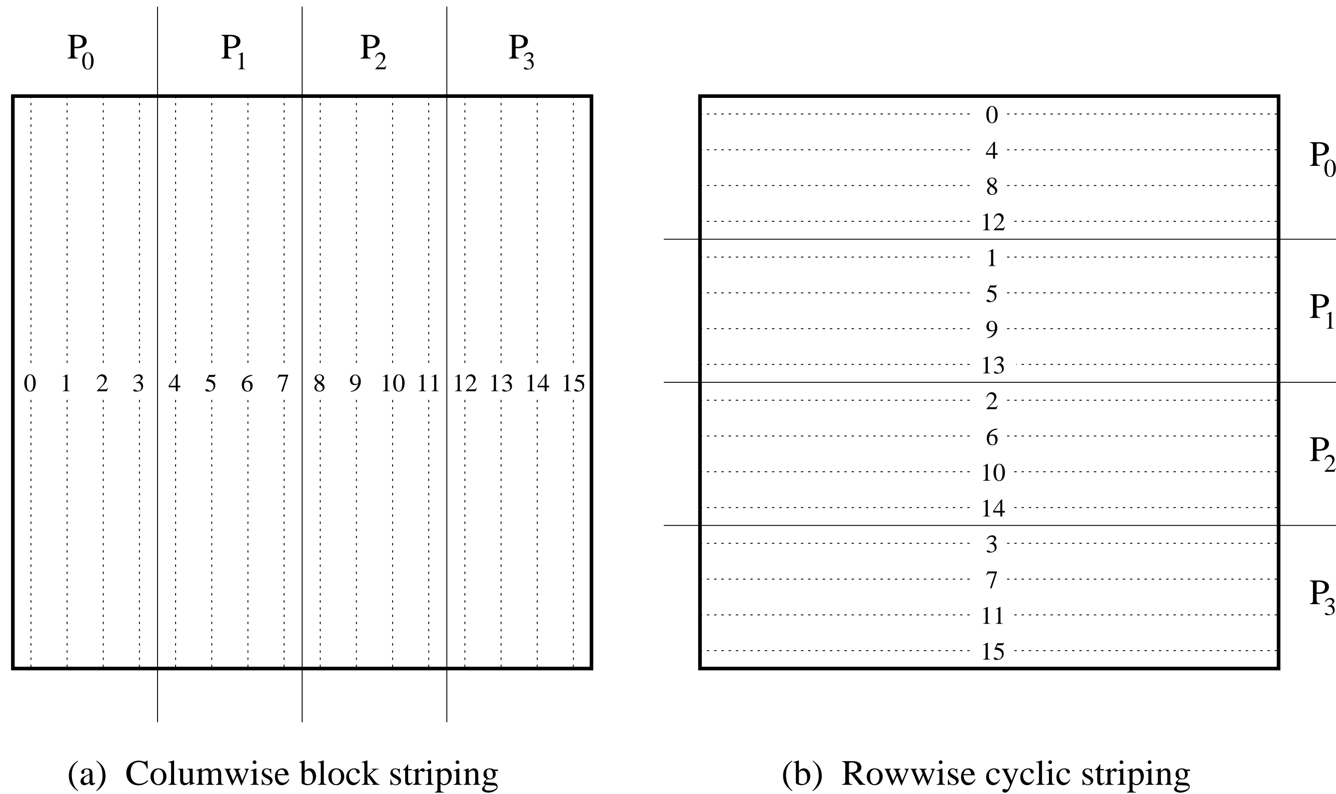


Figure 5.1 Uniform striped partitioning of 16 $\times$ 16 matrices on 4 processors.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

(0,0) (0,1) P_0 (1,0) (1,1)	(0,2) (0,3) P_1 (1,2) (1,3)	(0,4) (0,5) P_2 (1,4) (1,5)	(0,6) (0,7) P_3 (1,6) (1,7)
(2,0) (2,1) P_4 (3,0) (3,1)	(2,2) (2,3) P_5 (3,2) (3,3)	(2,4) (2,5) P_6 (3,4) (3,5)	(2,6) (2,7) P_7 (3,6) (3,7)
(4,0) (4,1) P_8 (5,0) (5,1)	(4,2) (4,3) P_9 (5,2) (5,3)	(4,4) (4,5) P_{10} (5,4) (5,5)	(4,6) (4,7) P_{11} (5,6) (5,7)
(6,0) (6,1) P_{12} (7,0) (7,1)	(6,2) (6,3) P_{13} (7,2) (7,3)	(6,4) (6,5) P_{14} (7,4) (7,5)	(6,6) (6,7) P_{15} (7,6) (7,7)

(a) Block-checkerboard partitioning

(0,0) (0,4) P_0 (4,0) (4,4)	(0,1) (0,5) P_1 (4,1) (4,5)	(0,2) (0,6) P_2 (4,2) (4,6)	(0,3) (0,7) P_3 (4,3) (4,7)
(1,0) (1,4) P_4 (5,0) (5,4)	(1,1) (1,5) P_5 (5,1) (5,5)	(1,2) (1,6) P_6 (5,2) (5,6)	(1,3) (1,7) P_7 (5,3) (5,7)
(2,0) (2,4) P_8 (6,0) (6,4)	(2,1) (2,5) P_9 (6,1) (6,5)	(2,2) (2,6) P_{10} (6,2) (6,6)	(2,3) (2,7) P_{11} (6,3) (6,7)
(3,0) (3,4) P_{12} (7,0) (7,4)	(3,1) (3,5) P_{13} (7,1) (7,5)	(3,2) (3,6) P_{14} (7,2) (7,6)	(3,3) (3,7) P_{15} (7,3) (7,7)

(b) Cyclic-checkerboard partitioning

Figure 5.2 Checkerboard partitioning of 8×8 matrices on 16 processors.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Matrix Multiplication

The standard sequential algorithm is $\Theta\left(n^3\right)$ time.

```
1.  procedure MAT_MULT (A, B, C)
2.  begin
3.      for i := 0 to n - 1 do
4.          for j := 0 to n - 1 do
5.              begin
6.                  C[i, j] := 0;
7.                  for k := 0 to n - 1 do
8.                      C[i, j] := C[i, j] + A[i, k] × B[k, j];
9.              endfor;
10. end MAT_MULT
```

Program 5.2 The conventional serial algorithm for multiplication of two $n \times n$ matrices.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Blocked Matrix Multiplication

This can be expressed in blocked form:

1. **procedure** BLOCK_MAT_MULT (A, B, C)
2. **begin**
3. **for** $i := 0$ **to** $q-1$ **do**
4. **for** $j := 0$ **to** $q-1$ **do**
5. **begin**
6. Initialize all elements of $C_{i,j}$ to zero;
7. **for** $k := 0$ **to** $q-1$ **do**
8. $C_{i,j} := C_{i,j} + A_{i,k} \times B_{k,j}$;
9. **endfor**;
10. **end** BLOCK_MAT_MULT

Program 5.3 The block matrix multiplication algorithm for $n \times n$ matrices with a block size of $(n/q) \times (n/q)$.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Simple Parallelization (block checkerboard)

$\sqrt{p} \times \sqrt{p}$ processors, conceptually in a 2-D mesh, with $P_{i,j}$ storing blocks $A_{i,j}$ and $B_{i,j}$ and computing $C_{i,j}$.

Gather all the data required then compute locally.

$P_{i,j}$ requires $A_{i,k}$ and $B_{k,j}$ for all $0 \leq k < \sqrt{p}$.

Achieve this with two parallel all-to-all broadcast steps (in rows, then columns).

Hypercube implementation

Recall the mesh to hypercube embedding which maps rows/columns to sub-cubes, to produce total communication time of $2(t_s \log \sqrt{p} + t_w \frac{n^2}{p}(\sqrt{p}-1))$.

Computation time is $\Theta\left(\sqrt{p}(\frac{n}{\sqrt{p}})^3\right) = \Theta\left(\frac{n^3}{p}\right)$ and so cost is $\Theta\left(n^3 + p\log\sqrt{p} + n^2\sqrt{p}\right)$

Technically we should compare cost with the best known sequential algorithm, $\Theta\left(n^{2.376}\right)$ (Coppersmith & Winograd), but in practical terms the normal $\Theta\left(n^3\right)$ algorithm is actually "best". Thus, in that practical sense, our algorithm is cost optimal for $p = O\left(n^2\right)$.

Cannon's Algorithm

Similar in structure, but lighter on memory use (avoids duplication)

Within a row/column use each block in a different place at different times, which is achieved by cycling blocks along rows/columns, interleaving communication and computation.

A preliminary phase skews the alignment of blocks of A (B) cyclically i (j) places left (up) in row i (column j) and multiplies co-located blocks.

This is followed by $\sqrt{p} - 1$ iterations of a step which computes (local matrix multiply and add) then communicates with a block cyclic shift left (up).

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

(a) Initial alignment of A

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

(b) Initial alignment of B

$A_{0,0}$ $B_{0,0}$	$A_{0,1}$ $B_{1,1}$	$A_{0,2}$ $B_{2,2}$	$A_{0,3}$ $B_{3,3}$
$A_{1,1}$ $B_{1,0}$	$A_{1,2}$ $B_{2,1}$	$A_{1,3}$ $B_{3,2}$	$A_{1,0}$ $B_{0,3}$
$A_{2,2}$ $B_{2,0}$	$A_{2,3}$ $B_{3,1}$	$A_{2,0}$ $B_{0,2}$	$A_{2,1}$ $B_{1,3}$
$A_{3,3}$ $B_{3,0}$	$A_{3,0}$ $B_{0,1}$	$A_{3,1}$ $B_{1,2}$	$A_{3,2}$ $B_{2,3}$

(c) A and B after initial alignment

$A_{0,1}$ $B_{1,0}$	$A_{0,2}$ $B_{2,1}$	$A_{0,3}$ $B_{3,2}$	$A_{0,0}$ $B_{0,3}$
$A_{1,2}$ $B_{2,0}$	$A_{1,3}$ $B_{3,1}$	$A_{1,0}$ $B_{0,2}$	$A_{1,1}$ $B_{1,3}$
$A_{2,3}$ $B_{3,0}$	$A_{2,0}$ $B_{0,1}$	$A_{2,1}$ $B_{1,2}$	$A_{2,2}$ $B_{2,3}$
$A_{3,0}$ $B_{0,0}$	$A_{3,1}$ $B_{1,1}$	$A_{3,2}$ $B_{2,2}$	$A_{3,3}$ $B_{3,3}$

(d) Submatrix locations after first shift

$A_{0,2}$ $B_{2,0}$	$A_{0,3}$ $B_{3,1}$	$A_{0,0}$ $B_{0,2}$	$A_{0,1}$ $B_{1,3}$
$A_{1,3}$ $B_{3,0}$	$A_{1,0}$ $B_{0,1}$	$A_{1,1}$ $B_{1,2}$	$A_{1,2}$ $B_{2,3}$
$A_{2,0}$ $B_{0,0}$	$A_{2,1}$ $B_{1,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{3,3}$
$A_{3,1}$ $B_{1,0}$	$A_{3,2}$ $B_{2,1}$	$A_{3,3}$ $B_{3,2}$	$A_{3,0}$ $B_{0,3}$

(e) Submatrix locations after second shift

$A_{0,3}$ $B_{3,0}$	$A_{0,0}$ $B_{0,1}$	$A_{0,1}$ $B_{1,2}$	$A_{0,2}$ $B_{2,3}$
$A_{1,0}$ $B_{0,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{2,2}$	$A_{1,3}$ $B_{3,3}$
$A_{2,1}$ $B_{1,0}$	$A_{2,2}$ $B_{2,1}$	$A_{2,3}$ $B_{3,2}$	$A_{2,0}$ $B_{0,3}$
$A_{3,2}$ $B_{2,0}$	$A_{3,3}$ $B_{3,1}$	$A_{3,0}$ $B_{0,2}$	$A_{3,1}$ $B_{1,3}$

(f) Submatrix locations after third shift

Figure 5.10 The communication steps in Cannon's algorithm on 16 processors.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Analysis

For 2-D Mesh first alignment achieved in $\Theta\left(\frac{n^2}{p}\sqrt{p}\right)$ time (SF routing), with $\Theta\left((\frac{n}{\sqrt{p}})^3\right)$ time for first computation.

Subsequent $\sqrt{p}-1$ steps each in $\Theta\left((\frac{n}{\sqrt{p}})^3\right)$ time.

Total time as for 'simple' algorithm, but without memory overhead.

Solving Systems of Linear Equations

Find $x_0, x_1, ..., x_{n-1}$ such that

$$\begin{array}{cccccccc}
a_{0,0}x_0 & + & a_{0,1}x_1 & \ldots & + & a_{0,n-1}x_{n-1} & = & b_0 \\
a_{1,0}x_0 & + & a_{1,1}x_1 & \ldots & + & a_{1,n-1}x_{n-1} & = & b_1 \\
. & & . & & & . & & . \\
. & & . & & & . & & . \\
. & & . & & & . & & . \\
a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & \ldots & + & a_{n-1,n-1}x_{n-1} & = & b_{n-1}
\end{array}$$

Solving Systems of Linear Equations

First reduce to upper triangular form (Gaussian elimination).

$$\begin{array}{ccccccc}
x_0 & + & u_{0,1}x_1 & \ldots & + & u_{0,n-1}x_{n-1} & = & y_0 \\
 & & x_1 & \ldots & + & u_{1,n-1}x_{n-1} & = & y_1 \\
 & & & & & . & & . \\
 & & & & & . & & . \\
 & & & & & . & & . \\
 & & & & & x_{n-1} & = & y_{n-1}
\end{array}$$

Finally solve by back substitution.

Gaussian Elimination

Assume that the matrix is 'non-singular' and ignore numerical stability concerns (a course in itself!).

Working row by row, adjust active row so that diagonal element is 1, then adjust subsequent rows to zero all items directly beneath this.

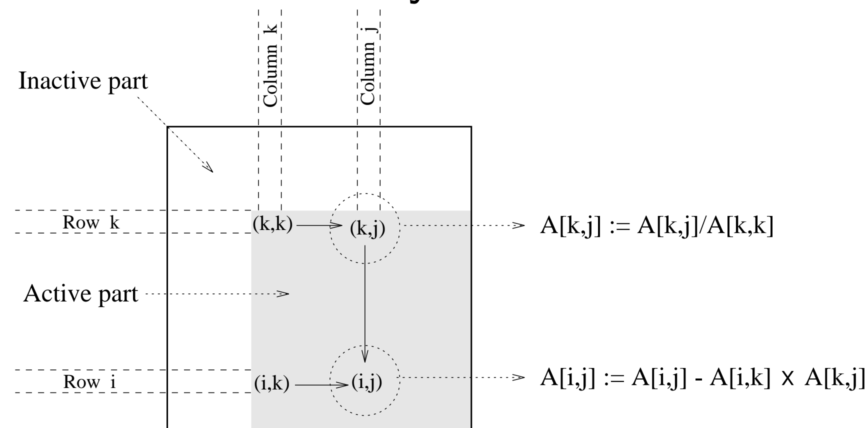


Figure 5.13 A typical computation in Gaussian elimination.
Copyright (r) 1994 Benjamin/Cummings Publishing Co.

```
1.   procedure GAUSSIAN_ELIMINATION (A, b, y)
2.   begin
3.      for k := 0 to n - 1 do                    /* Outer loop */
4.      begin
5.         for j := k + 1 to n - 1 do
6.            A[k, j] := A[k, j]/A[k, k];   /* Division step */
7.         y[k] := b[k]/A[k, k];
8.         A[k, k] := 1;
9.         for i := k + 1 to n - 1 do
10.        begin
11.           for j := k + 1 to n - 1 do
12.              A[i, j] := A[i, j] - A[i, k] × A[k, j]; /* Elimination step */
13.           b[i] := b[i] - A[i, k] × y[k];
14.           A[i, k] := 0;
15.        endfor;          /* Line 9 */
16.     endfor;             /* Line 3 */
17.  end GAUSSIAN_ELIMINATION
```

Program 5.4 A serial Gaussian elimination algorithm that converts the system of linear equations $Ax = b$ to a unit upper-triangular system $Ux = y$. The matrix U occupies the upper-triangular locations of A. This algorithm assumes that $A[k, k] \neq 0$ when it is used as a divisor on lines 6 and 7.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Sequential time (work) is $\Theta\left(n^3\right)$ (because of the triply nested loops).

Row-Striped Parallel Algorithm

Given a distribution of one row of A (and item of b) per processor, the kth (outer) iteration is $\Theta(n-k-1)$ computation time concurrently in each subsequent row, giving $\Theta\left(n^2\right)$ overall.

The second step requires broadcast of relevant part of updated 'active' row to all processors beneath it (a one-all broadcast, in the first case), and so total communications time is for n one-all broadcasts (slight overkill, but not asymptotically), with reducing message size.

On a hypercube $\sum_{k=0}^{n-1}(t_s + t_w(n-k-1))\log n \ = \ \Theta\left(n^2 \log n\right)$

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(a) Computation:

(i) A[k,j] := A[k,j]/A[k,k] for k < j < n

(ii) A[k,k] := 1

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(b) Commmunication:

One-to-all brodcast of row A[k,*]

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(c) Computation:

(i) A[i,j] := A[i,j] - A[i,k] × A[k,j]
for k < i < n and k < j < n

(ii) A[i,k] := 0 for k < i < n

Figure 5.14 Gaussian elimination steps during the iteration corresponding to $k = 3$ for an 8×8 matrix striped rowwise on 8 processors.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Pipelined Version

Consider a 1-D array version of the previous algorithm. As soon as a processor has forwarded data in a broadcast, it can start its work for that iteration, producing a pipelined effect across (outer) iterations.

n phases, begun at $O(n)$ intervals, with the last taking constant time, hence $\Theta(n^2)$ time overall.

This is asymptotically cost optimal (and similarly for any other architecture which can easily mimic the array).

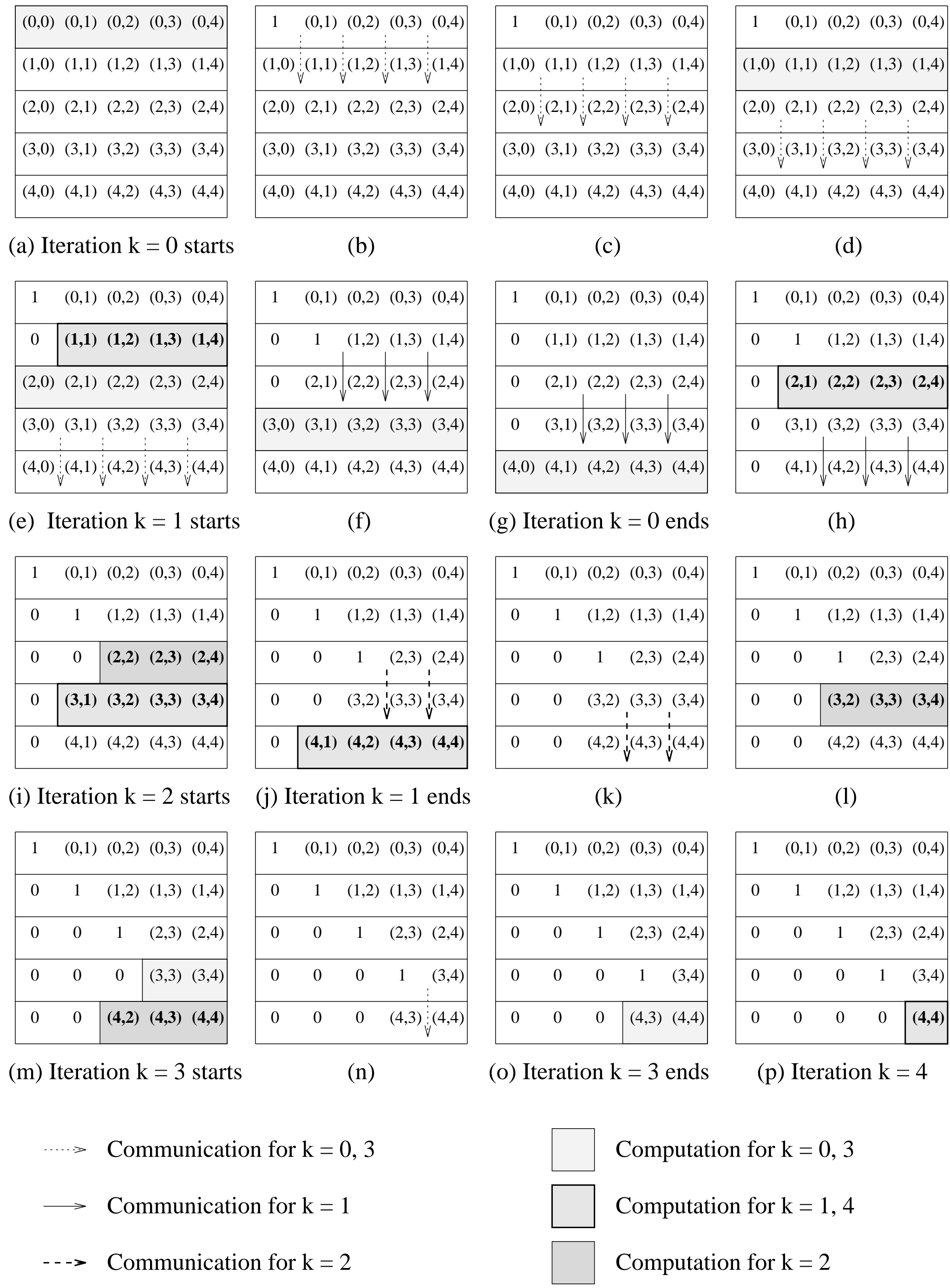


Figure 5.15 Pipelined Gaussian elimination on a 5×5 matrix stripe-partitioned with one row per processor.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

```
1.  procedure BACK_SUBSTITUTION (U, x, y)
2.  begin
3.     for k := n - 1 downto 0 do   /* Main loop */
4.        begin
5.           x[k] := y[k];
6.           for i := k - 1 downto 0 do
7.              y[i] := y[i] - x[k] × U[i, k];
8.        endfor;
9.  end BACK_SUBSTITUTION
```

Program 5.5 A serial algorithm for back-substitution. U is an upper-triangular matrix with all entries of the principal diagonal equal to one, and all subdiagonal entries equal to zero.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.

For a parallel 1-D array version, we pipeline broadcast of results up columns as before. This completes in $\Theta(n)$ time, which is cost-optimal (and negligible compared to triangularization).