Parallelization Strategies
ASD Distributed Memory HPC Workshop

Computer Systems Group

Research School of Computer Science
Australian National University
Canberra, Australia

November 01, 2017

Day 3 – Schedule

Australian National University

Distributed Memory HPC

Search Distributed Memory HPC

DISTRIBUTED MEMORY HPC

1. Messaging and Networks
2. Advanced Messaging
3. Parallelization Strategies
4. PGAS Paradigm
5. Distributed HPC Systems

Home » 3. Parallelization Strategies

Day 3: Parallelization Strategies

Time	Lecture Topics	Hands-On Exercise	Instructor
9:00	Embarrassingly Parallel Problems	Load Balancing Embarrassingly Parallel Problems	Peter Strazdins
10:30	COFFEE BREAK		
11:00	Parallelisation via Data Partitioning	Bucket Sort	
12:30	LUNCH		
13:30	Synchronous Computations	Synchronous Computations	
15:00	AFTERNOON TEA		
15:30	Parallel Matrix Algorithms	Matrix Multiply	

Parallelization Strategies lecture slides (pdf)

Outline

Outline: Embarrassingly Parallel Problems

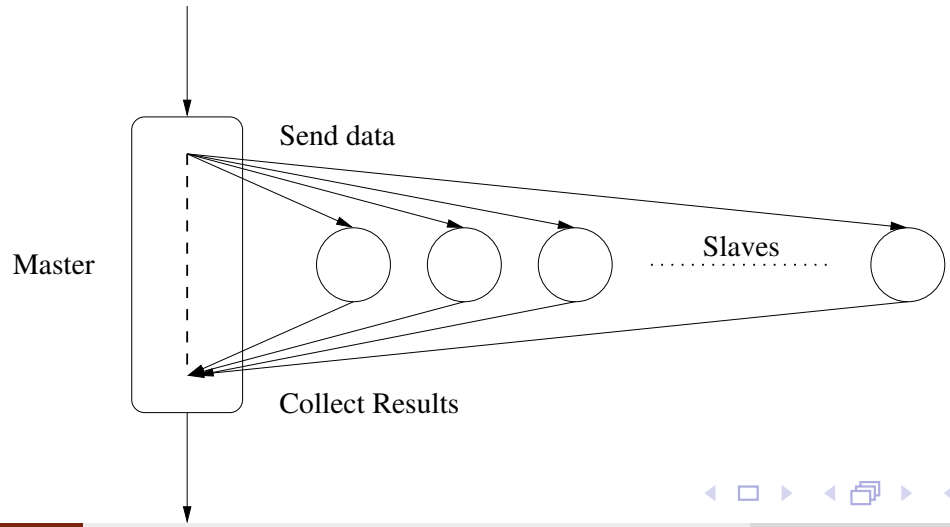
Australian National University

- what they are
- Mandelbrot Set computation
  - cost considerations
  - static parallelization
  - dynamic parallelizations and its analysis
- Monte Carlo Methods
- parallel random number generation

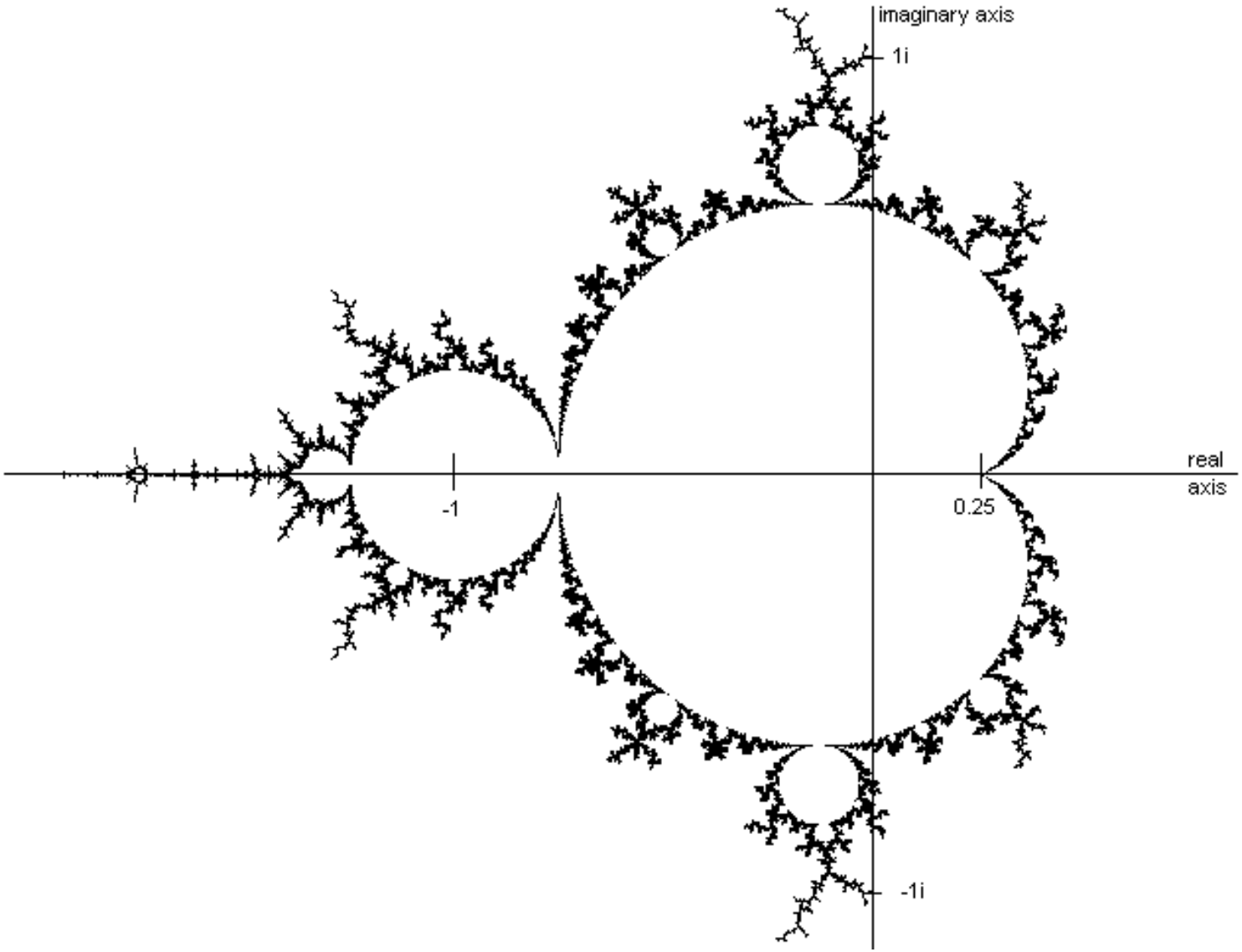
Embarrassingly Parallel Problems

Australian National University

- computation can be divided into completely independent parts for execution by separate processors (correspond to totally disconnected computational graphs)
  - infrastructure: Blocks of Independent Computations (BOINC) project
  - SETI@home and Folding@Home are projects solving very large such problems
- part of an application may be embarrassingly parallel
- distribution and collection of data are the key issues (can be non-trivial and/or costly)
- frequently uses the **master/slave** approach ($p - 1$ speedup)



Example#1: Computation of the Mandelbrot Set



The Mandelbrot Set

- set of points in complex plane that are quasi-stable
- computed by iterating the function

$$z_{k+1} = z_k^2 + c$$

  - z and c are complex numbers ($z = a + bi$)
  - z initially zero
  - c gives the position of the point in the complex plane
- iterations continue until $|z| > 2$ or some arbitrary iteration limit is reached

$$|z| = \sqrt{a^2 + b^2}$$

- enclosed by a circle centered at (0,0) of radius 2

Evaluating 1 Point

Australian National University

```
typedef struct complex{float real, imag;} complex;
const int MaxIter = 256;

int calc_pixel(complex c){
  int count = 0;
  complex z = {0.0, 0.0};
  float temp, lengthsq;
  do {
    temp = z.real * z.real - z.imag * z.imag + c.real
    z.imag = 2 * z.real * z.imag + c.imag;
    z.real = temp;
    lengthsq = z.real * z.real + z.imag * z.imag;
    count++;
  } while (lengthsq < 4.0  &&  count < MaxIter);
  return count;
}
```

Building the Full Image

Define:

- min. and max. values for c (usually -2 to 2)
- number of horizontal and vertical pixels

```
for (x = 0; x < width; x++)
  for (y = 0; y < height; y++){
    c.real = min.real + ((float) x * (max.real - min.real)/width);
    c.imag = min.imag + ((float) y * (max.imag - min.imag)/height);
    color = calc_pixel(c);
    display(x, y, color);
}
```

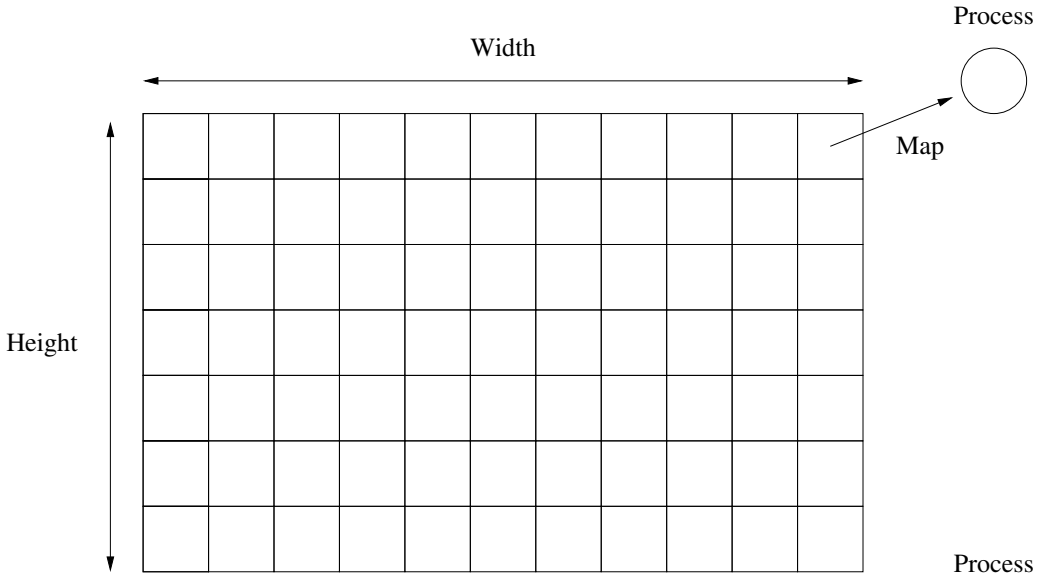
Summary:

- `width` × `height` totally independent tasks
- each task can be of different length

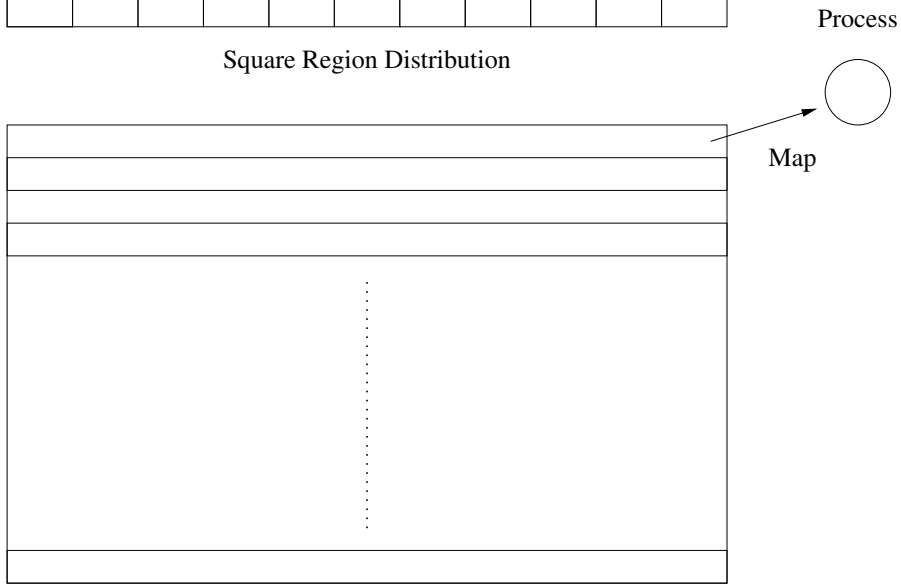
Cost Considerations on NCI's Raijin

- 10 flops per iteration
- maximum 256 iterations per point
- approximate time on one Raijin core:
  $10 \times 256/(8 \times 2.6 \times 10^9) \approx 0.12\text{usec}$
- between two nodes the time to communicate single point to slave and receive result $\approx 2 \times 2\text{usec}$ (latency limited)
- conclusion: cannot parallelize over individual points
- also must allow time for master to send to all slaves before it can return to any given process

Parallelisation: Static



Square Region Distribution



Row Distribution

Static Implementation

Australian National University

Master:

```
for (slave = 1, row = 0; slave < nproc; slave++) {
  send(&row, slave);
  row = row + height/nproc;
}
for (npixel = 0; npixel < (width * height); npixel++) {
  recv(&x, &y, &color, any_processor);
  display(x, y, color);
}
```

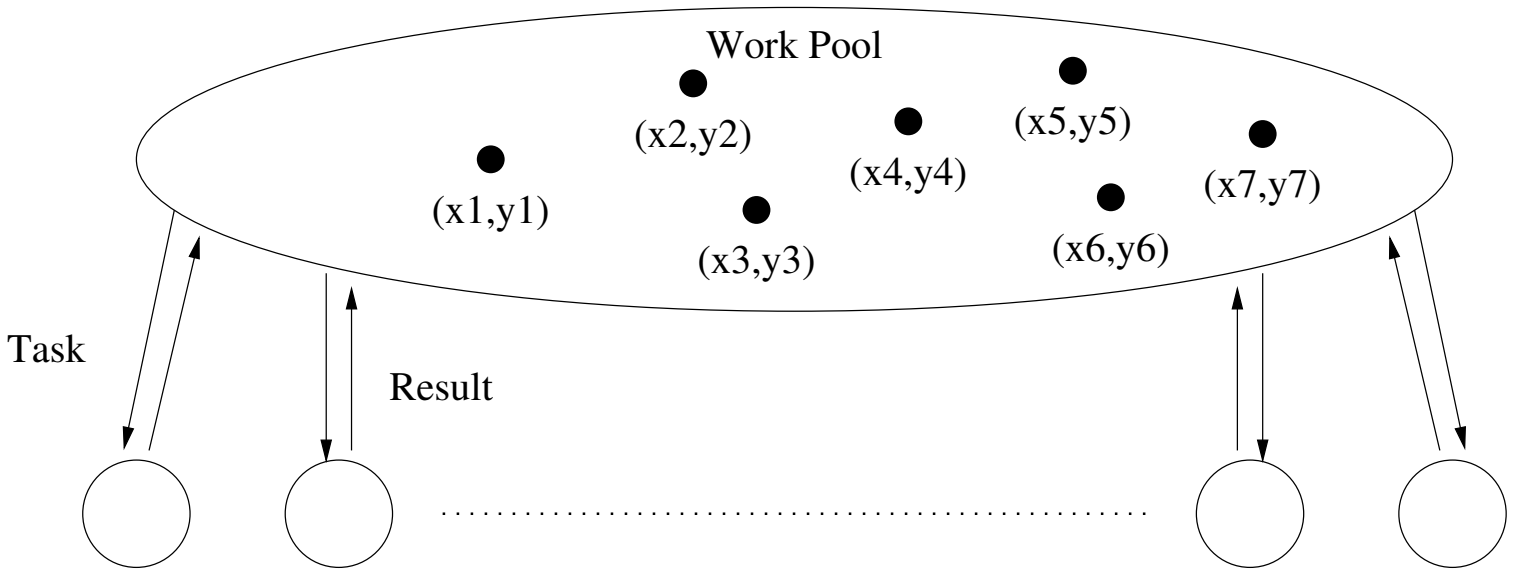
Slave:

```
const int master = 0; // proc. id
recv(&firstrow, master);
lastrow = MIN(firstrow + height/nproc, height);
for (x = 0; x < width; x++)
  for (y = firstrow; y < lastrow; y++) {
    c.real = min.real + ((float) x * (max.real - min.real)/width);
    c.imag = min.imag + ((float) y * (max.imag - min.imag)/height);
    color = calc_pixel(c);
    send(&x, &y, &color, master);
  }
```

Dynamic Task Assignment

Australian National University

- discussion point: why would we expect static assignment to be sub-optimal for the Mandelbrot set calculation? Would any regular static decomposition be significantly better (or worse)?
- usa a pool of **over-decomposed** tasks that are dynamically assigned to next requesting process:



Processor Farm: Master


```
count = 0;
row = 0;
for (slave = 1; slave < nproc; k++){
  send(&row, slave, data_tag);
  count++;
  row++;
}
do {
  recv(&slave, &r, &color, any_proc, result_tag);
  count--;
  if (row < height) {
     send(&row, slave, data_tag);
     row++;
     count++;
  } else
    send(&row, slave, terminator_tag);
  display_vector(r, color);
} while (count > 0);
```

Processor Farm: Slave


```
const int master = 0; //proc id.
recv(&y, master, any_tag, source_tag);
while (source_tag == data_tag) {
  c.imag = min.imag + ((float) y * (max.imag - min.imag)/height);
  for (x = 0; x < width; x++) {
    c.real = min.real + ((float) x * (max.real - min.real)/width);
    color[x] = calc_pixel(c);
  }
  send(&myid, &y, color, master, result_tag);
  recv(&y, master, source_tag);
}
```

Analysis

Australian National University

Let p, m, n, I denote `nproc`, `height`, `width`, `MaxIter`:

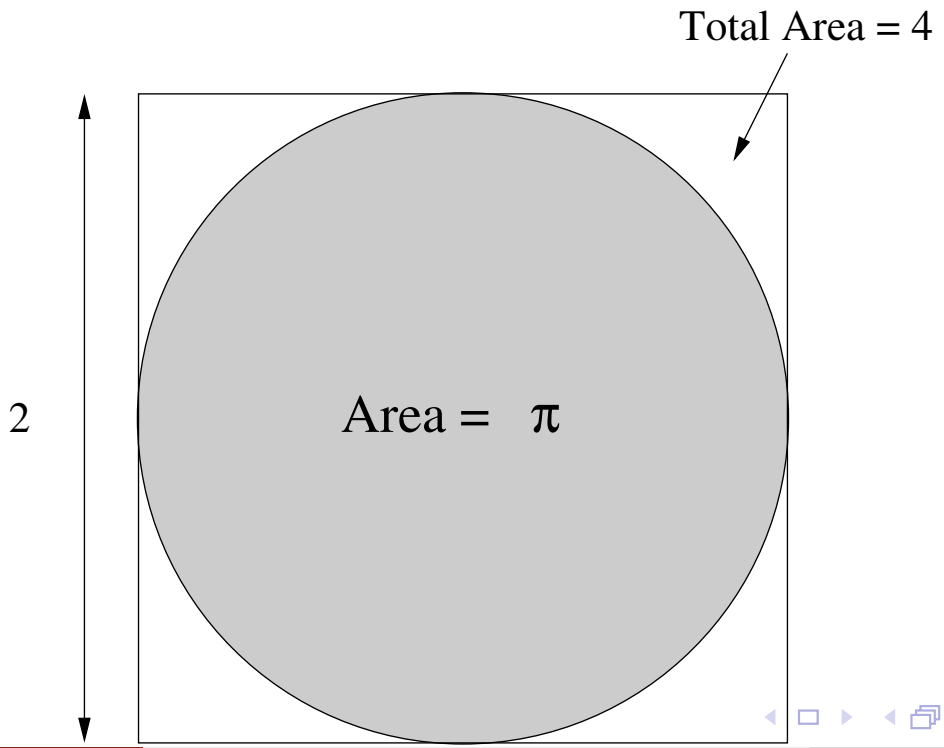
- sequential time: (t_f denotes floating point operation time)
  $t_{\text{seq}} \leq I \cdot mn \cdot t_f = O(mn)$
- parallel communication 1: (neglect t_h term, assume message length of 1 word)
  $t_{\text{com1}} = 2(p-1)(t_s + t_w)$
- parallel computation:
  $t_{\text{comp}} \leq \frac{I \cdot mn}{p-1} t_f$
- parallel communication 2:
  $t_{\text{com2}} = \frac{m}{p-1}(t_s + t_w)$
- overall:
  $t_{\text{par}} \leq \frac{I \cdot mn}{p-1} t_f + (p - 1 + \frac{m}{p-1})(t_s + t_w)$

Discussion point: What assumptions have we been making here? Are there any situations where we might still have poor performance, and how could we mitigate this?

Example#2: Monte Carlo Methods

- use random numbers to solve numerical/physical problems
- evaluation of π by determining if random points in the unit square fall inside a circle

$$\frac{\text{area of circle}}{\text{area of square}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$$



Monte Carlo Integration

Australian National University

- evaluation of integral $(x_1 \leq x_i \leq x_2)$

$$\text{area} = \int_{x_1}^{x_2} f(x)\,dx = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} f(x_i)(x_2 - x_1)$$

- example

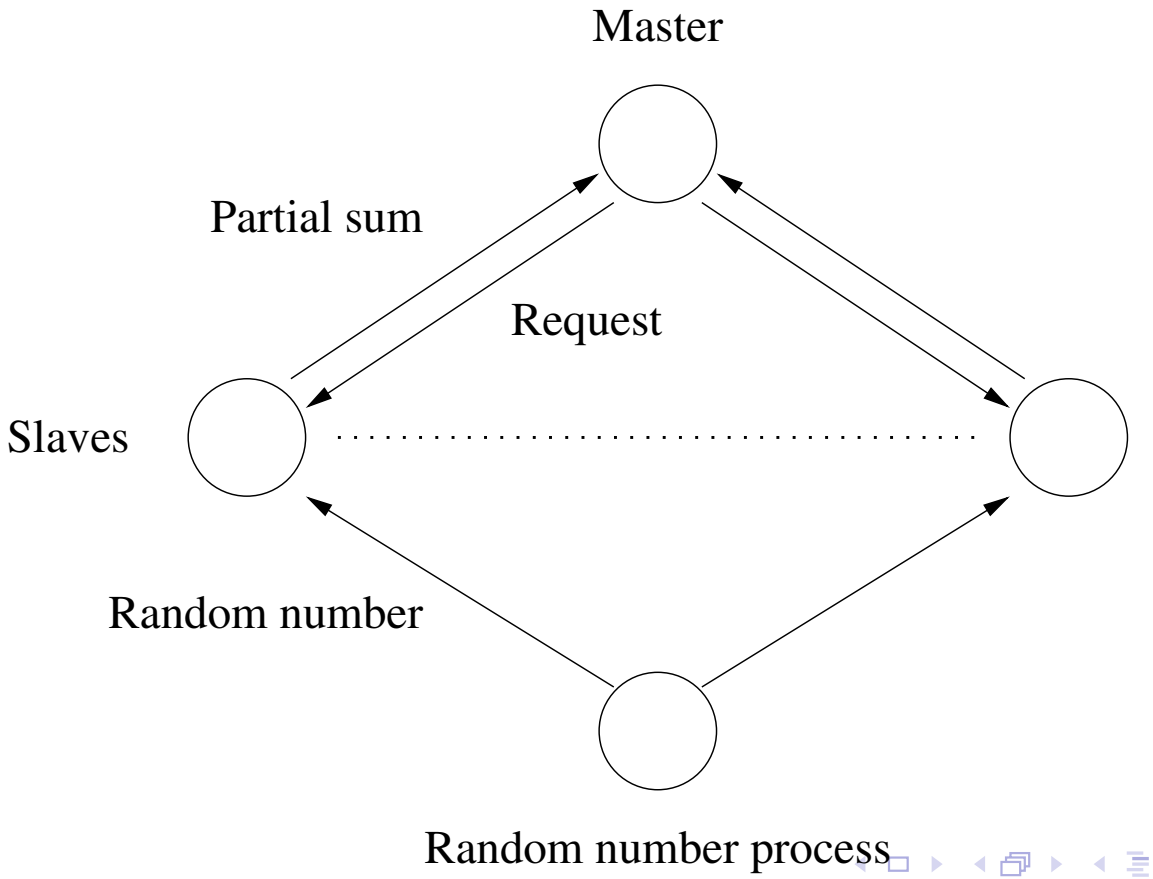
$$I = \int_{x_1}^{x_2} (x^2 - 3x)\,dx$$

```
sum = 0;
for (i = 0; i < N; i++) {
    xr = rand_v(x1, x2);
    sum += xr * xr - 3 * xr;
}
area = sum * (x2 - x1) / N;
```

- where `rand_v(x1, x2)` computes a pseudo-random number between `x1` and `x2`

Parallelization

- only problem is ensuring each process uses a different random number and that there is no correlation
- one solution is to have a unique process (maybe the master) issuing random numbers to the slaves



Parallel Code: Integration

Australian National University

Master (process 0):

```
for (i = 0; i < N/n; i++) {
  for (j = 0; j < n; j++)
    xr[j] = rand_v(x1, x2);
  recv(any_proc, req_tag, &p_src)
      ;
  send(xr, n, p_src, comp_tag);
}
for (i=1; i < nproc; i++) {
   recv(i, req_tag);
   send(i, stop_tag);
}
sum = 0;
reduce_add(&sum, p_group);
```

Slave:

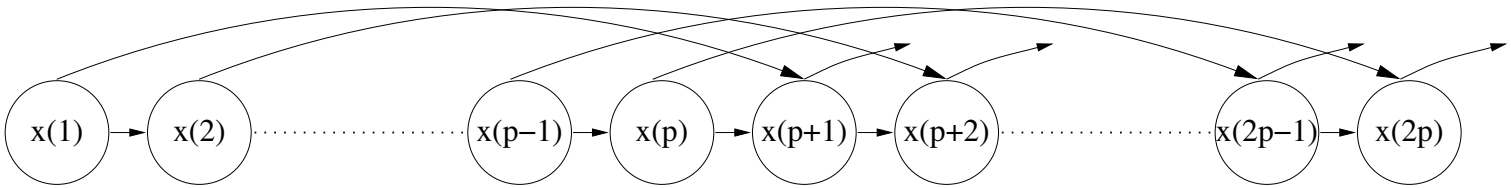
```
const int master = 0; //proc id.
sum = 0;
send(master, req_tag);
recv(xr, &n, master, tag);
while (tag == comp_tag) {
  for (i = 0; i < n; i++)
    sum += xr[i]*xr[i] - 3*xr[i]
  send(0, req_tag);
  recv(xr, n, master, &tag);
}
reduce_add(&sum, p_group);
```

Question: performance problems with this code?

Parallel Random Numbers

Australian National University

- linear congruential generators $x_{i+1} = (ax_i + c) \bmod m$ (a, c, and m are constants)
- using property $x_{i+p} = (A(a, p, m)x_i + C(c, a, p, m)) \bmod m$, we can generate the first p random numbers sequentially to repeatedly calculate the next p numbers in parallel



Summary: Embarrassingly Parallel Problems

Australian National University

- defining characteristic: tasks do not need to communicate
- non-trivial however: providing input data to tasks, assembling results, load balancing, scheduling, heterogeneous compute resources, costing
  - **static task assignment** (lower communication costs) vs. **dynamic task assignment** + **overdecomposition** (better load balance)
- **Monte Carlo** or **ensemble simulations** are a big use of computational power!
- the field of grid computing arose to solve this issue

Hands-on Exercise: Embarrassingly || Problems

Australian National University

Outline

Outline: Parallelisation via Data Partitioning

- partitioning strategies
- vector summation via partitioning, via divide-and-conquer
- binary trees (divide-and-conquer)
- bucket sort
- numerical integration - adaptive techniques
- N-body problems

Challenge from PS1: can you write a well-balanced parallel Mandelbrot set program using static task assignment?

Partitioning Strategies

Australian National University

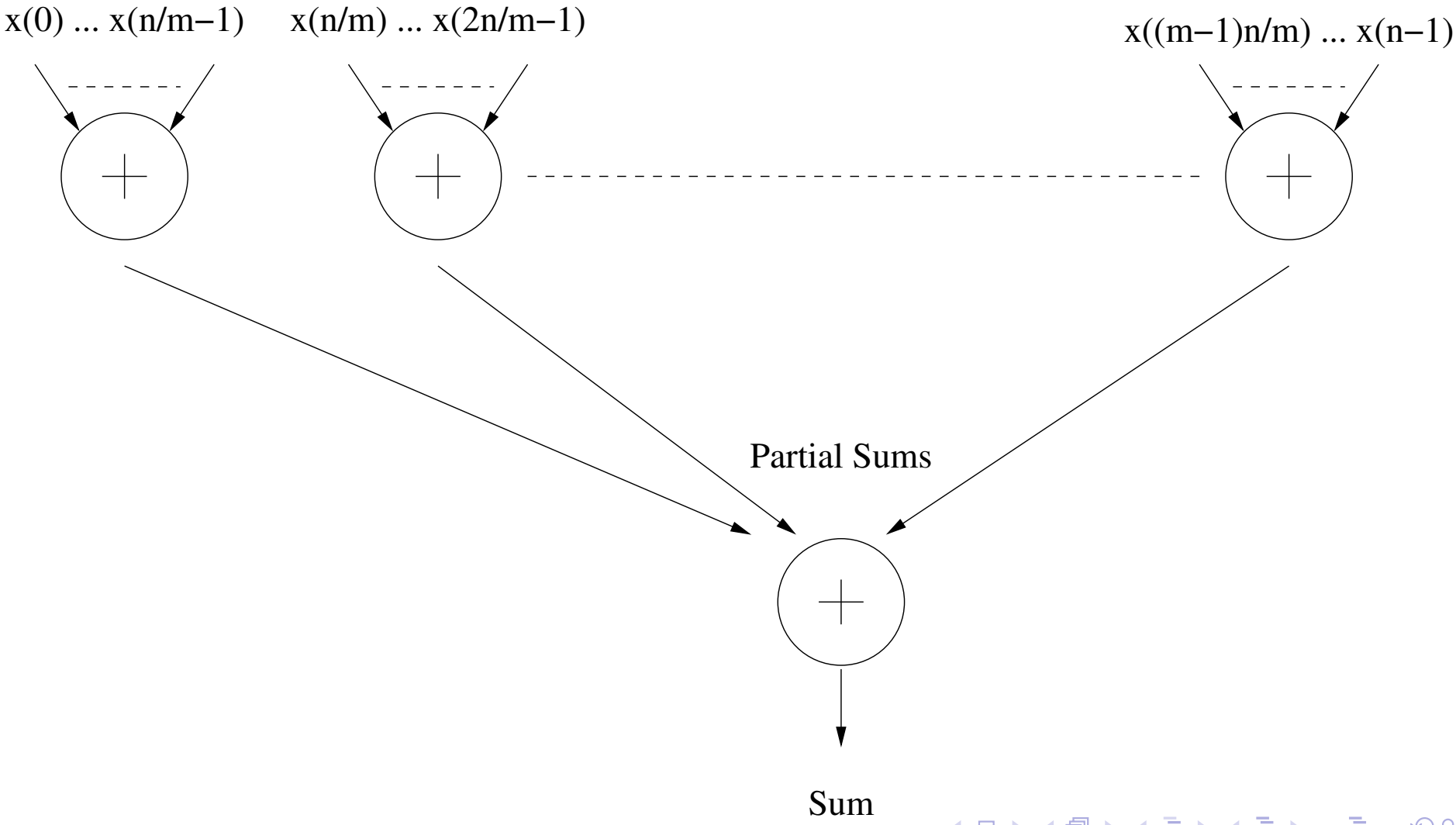
- replicated data approach (no partitioning)
  - each process has entire copy of data but does subset of computation
- partition program data to different processes
  - most common
  - strategies: **domain decomposition**, **divide-and-conquer**
- partitioning of program functionality
  - much less common
  - **functional decomposition**
- consider the addition of numbers

$$s = \sum_{i=0}^{n-1} x_i$$

Example#1: Simple Summation of Vector

Australian National University

- divide numbers into *m* equal parts



Master/Slave Send/Recv Approach

Master:

```
s = n / m;
for (i = 0, x = 0; i < m; i++, x = x + s)
  send(&numbers[x], s, i+1 /*slave id*/);

sum = 0;
for (i = 0; i < m; i++) {
  recv(&part_sum, any_proc);
  sum = sum + part_sum;
}
```

Slave:

```
recv(numbers, s, master);
part_sum = 0;
for (i = 0; i < s; i++)
  part_sum = part_sum + numbers[i];
send(&part_sum, master);
```

Using MPI_Scatter and MPI_Reduce

See `man MPI_Scatter` and `man MPI_Reduce`

```
1  s = n/m;
   MPI_Scatter(numbers, s /*sendcount*/, MPI_FLOAT /*send data*/,
3              numbers, s /*recvcount*/, MPI_FLOAT /*recv data*/,
               0 /*root*/, MPI_COMM_WORLD);
5
   for (i = 0; i < s; i++)
7     part_sum = part_sum + numbers[i];

9  MPI_Reduce(&part_sum, &sum, 1 /*count*/, MPI_FLOAT,
              MPI_SUM, 0 /*root*/, MPI_COMM_WORLD);
```

- NOT **master/slave**
- the root sends data to all processes (including itself)
- note related MPI calls:
  - `MPI_Scatterv()`: scatters variable lengths
  - `MPI_Allreduce()`: returns result to all processors

Analysis

Australian National University

Sequential:

- $n-1$ additions thus $O(n)$

Parallel ($p = \mathtt{m}$):

- communication #1: $t_{\text{scatter}} = p(t_s + \frac{n}{p} t_w)$
- computation #1: $t_{\text{partialsum}} = (\frac{n}{p}) t_f$
- communication #2: $t_{\text{reduce}} = p(t_s + t_w)$
- computation #2: $t_{\text{finalsum}} = (p-1) t_f$
- overall: $t_{\text{p}} = 2pt_s + (n+p)t_w + (n/p + p - 1)t_f = O(n+p)$
- **worse than sequential code!!**

Discussion point: in this example, we are assuming the associative property of addition ($+$)? Is this strictly true for floating point numbers? What impact does this have for such parallel algorithms?

Domain Decomposition via Divide-and-Conquer

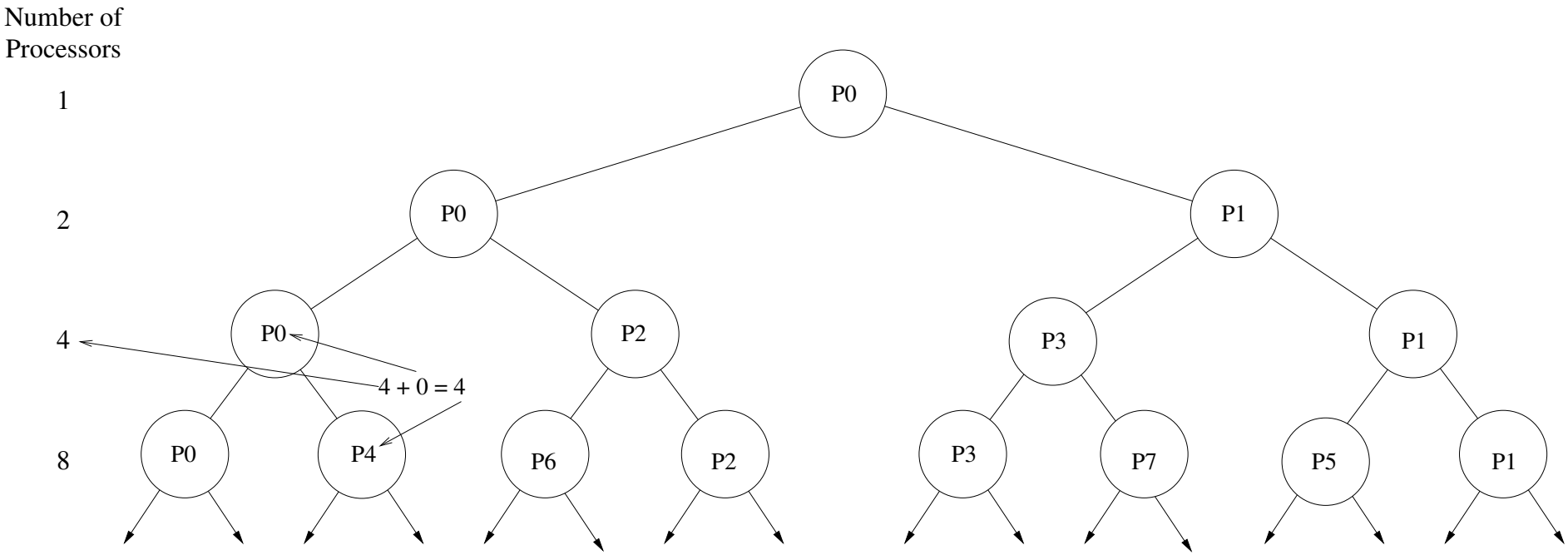
- problems that can be recursively divided into smaller problems of the same type
- recursive implementation of the summation problem:

```
int add(int *s) {
  if (numbers(s) == 1)
    return (s[0]);
  else {
    divide(s, s1, s2);
    part_sum1 = add(s1);
    part_sum2 = add(s2);
    return (part_sum1 + part_sum2);
  }
}
```

Binary Tree

Australian National University

- **divide-and-conquer** with binary partitioning
- note number of working processors decreases going up the tree



Simple Binary Tree Code


```
/* Binary Tree broadcast
  a) 0->1
  b) 0->2, 1->3
  c) 0->4, 1->5, 2->6, 3->7
  d) 0->8, 1->9, 2->10, 3->11, 4->12, 5->13, 6->14, 7->15
*/
  lo = 1;
  while (lo < nproc) {
    if (me < lo) {
      id = me + lo;
      if (id < nproc)
        send(buf, lenbuf, id);
    }
    else if (me < 2*lo) {
      id = lo;
      recv(buf, lenbuf, id);
    }
    lo *= 2;
  }
```

This is used to scatter the vector; the reverse algorithm combines the partial sums.

Analysis

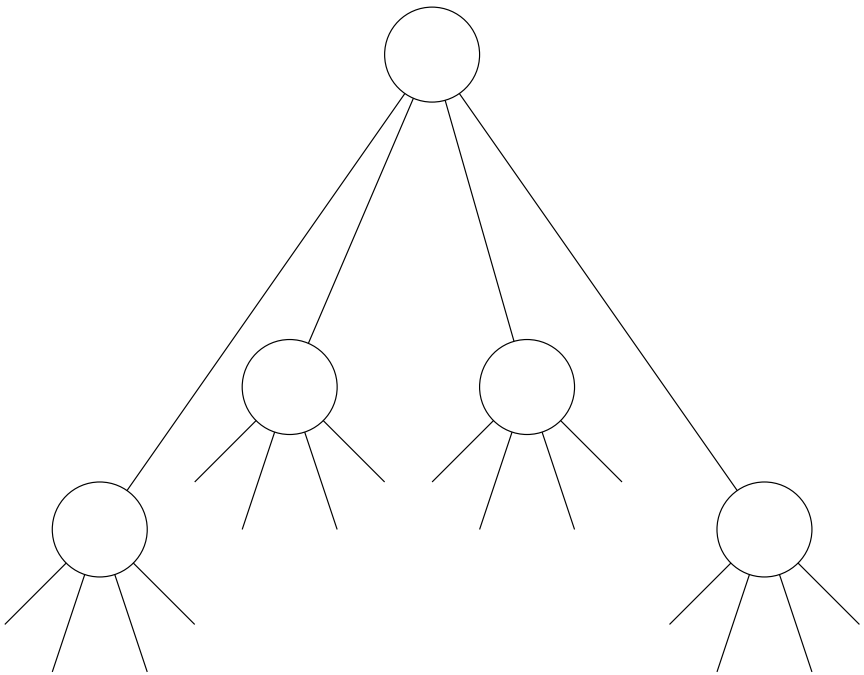
Australian National University

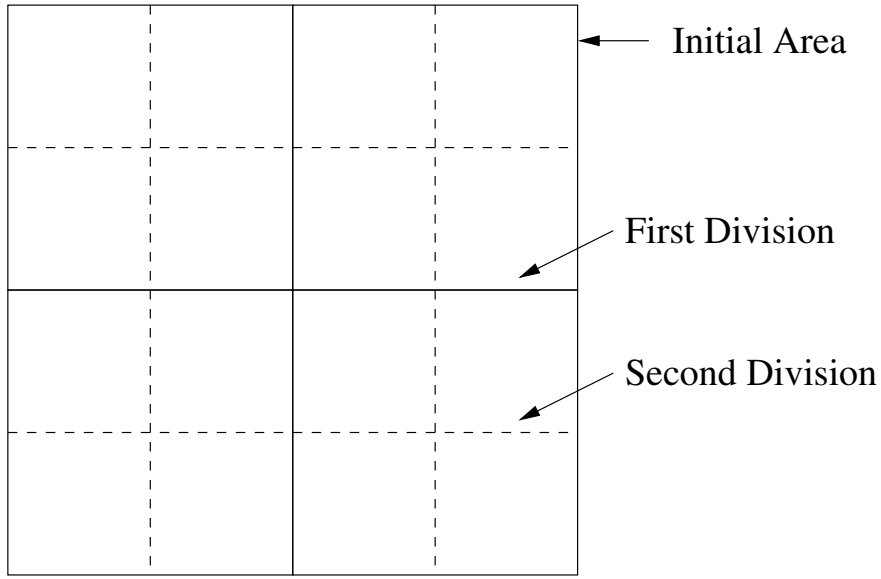
- assume n is a power of 2 and ignoring t_s
- communication#1: divide
  $t_{\text{divide}} = \frac{n}{2}t_w + \frac{n}{4}t_w + \frac{n}{8}t_w + \cdots \frac{n}{p}t_w = \frac{n(p-1)}{p}t_w$
- communication#2: combine
  $t_{\text{combine}} = \lg p \cdot t_w$
- computation:
  $t_{\text{comp}} = (\frac{n}{p} + \lg p)t_f$
- total:
  $t_{\text{p}} = (\frac{n(p-1)}{p} + \lg p)t_w + (\frac{n}{p} + \lg p)t_f$
- slightly better than before - as $p \to n$, cost $\to O(n)$

Higher Order Trees

Australian National University

- possible to divide data into higher order trees, e.g. a quad tree





Example#2: Bucket Sort

Australian National University

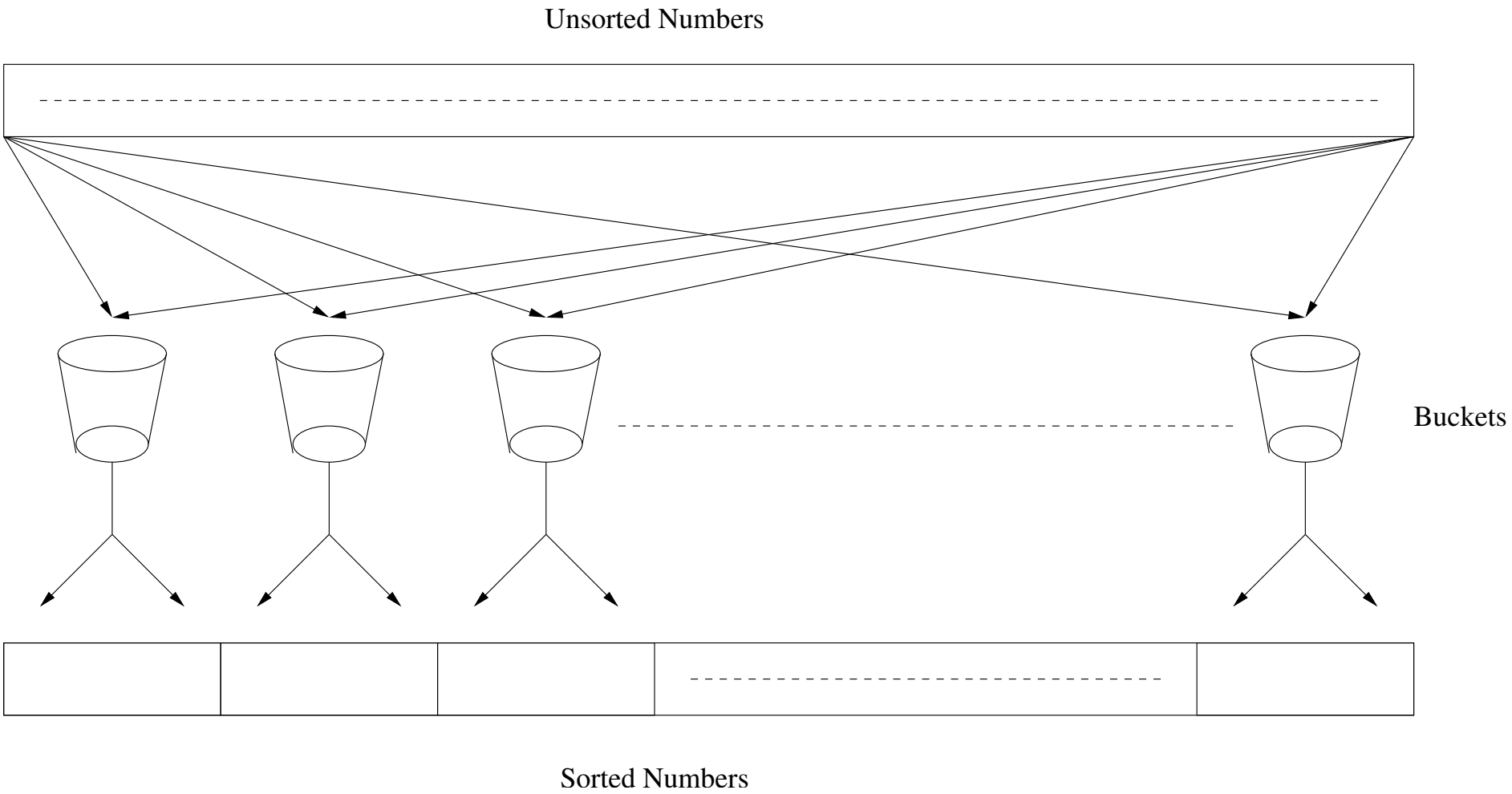
- Divide number range (a) into m equal regions

$$\left(0 \to \frac{a}{m} - 1\right), \left(\frac{a}{m} \to 2\frac{a}{m} - 1\right), \left(2\frac{a}{m} \to 3\frac{a}{m} - 1\right), \cdots$$

- assign one bucket to each region
- stage 1: numbers are placed into appropriate buckets
- stage 2: each bucket is sorted using a traditional sorting algorithm
- works best if numbers are evenly distributed over the range a
- sequential time

$$t_s = n + m((n/m)\lg(n/m)) = n + n\lg(n/m) = O(n\lg(n/m))$$

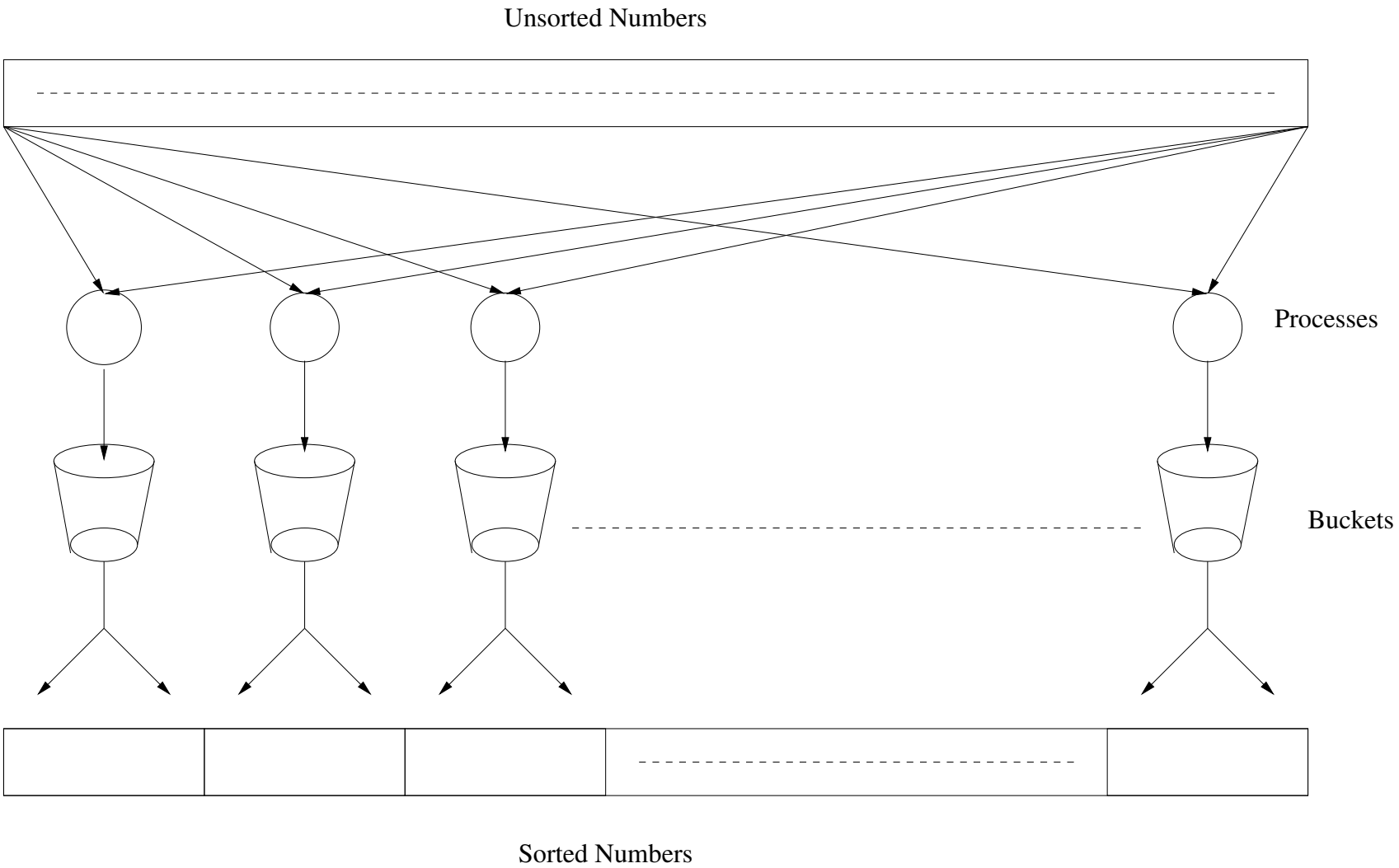
Sequential Bucket Sort



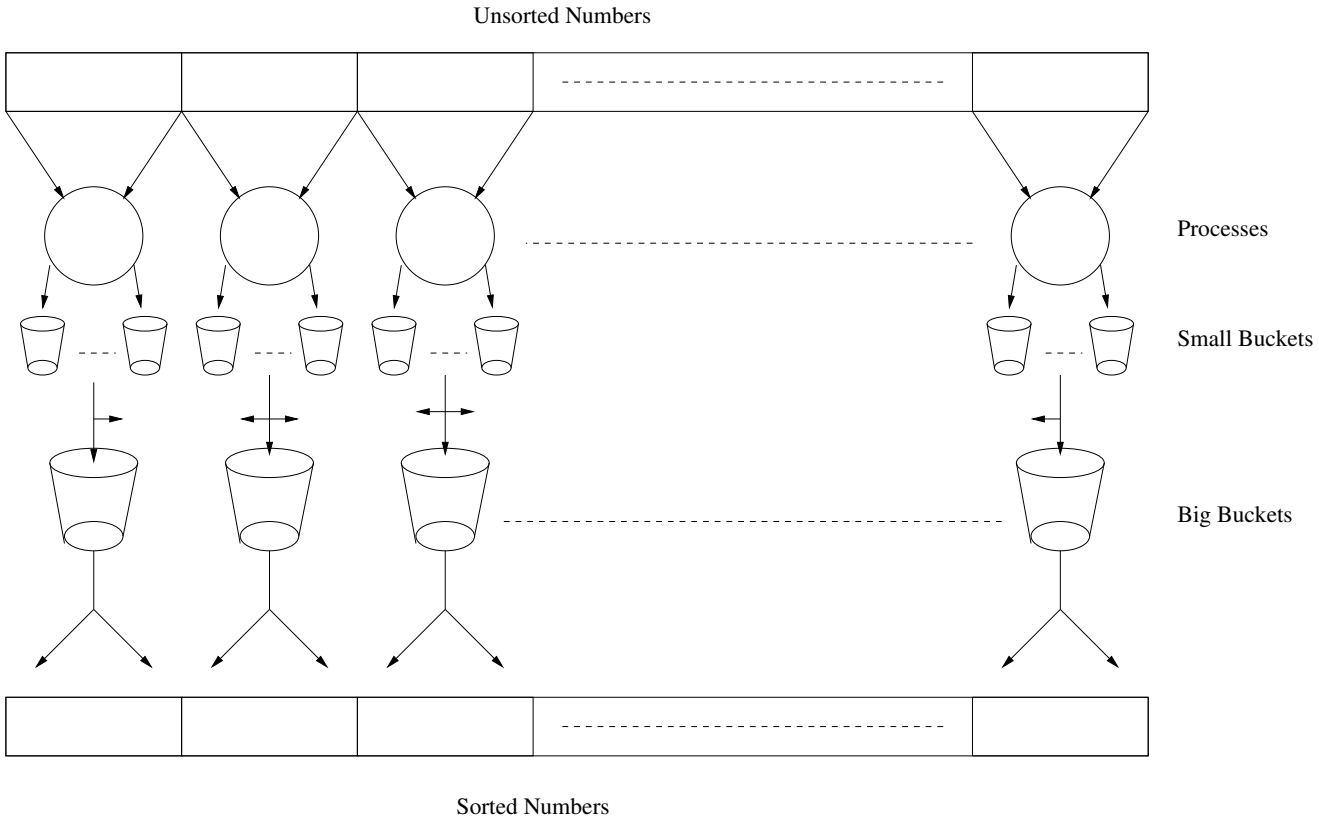
Parallel Bucket Sort#1

Australian National University

- assign one bucket to each process:



Parallel Bucket Sort#2



- assign p small buckets to each process
- note possible use of `MPI_Alltoall()`

```
MPI_Alltoall(void* sendbuf, int sendct, MPI_Datatype sendtype,
2             void* recvbuf, int recvct, MPI_Datatype recvtype,
              MPI_Comm comm)
```

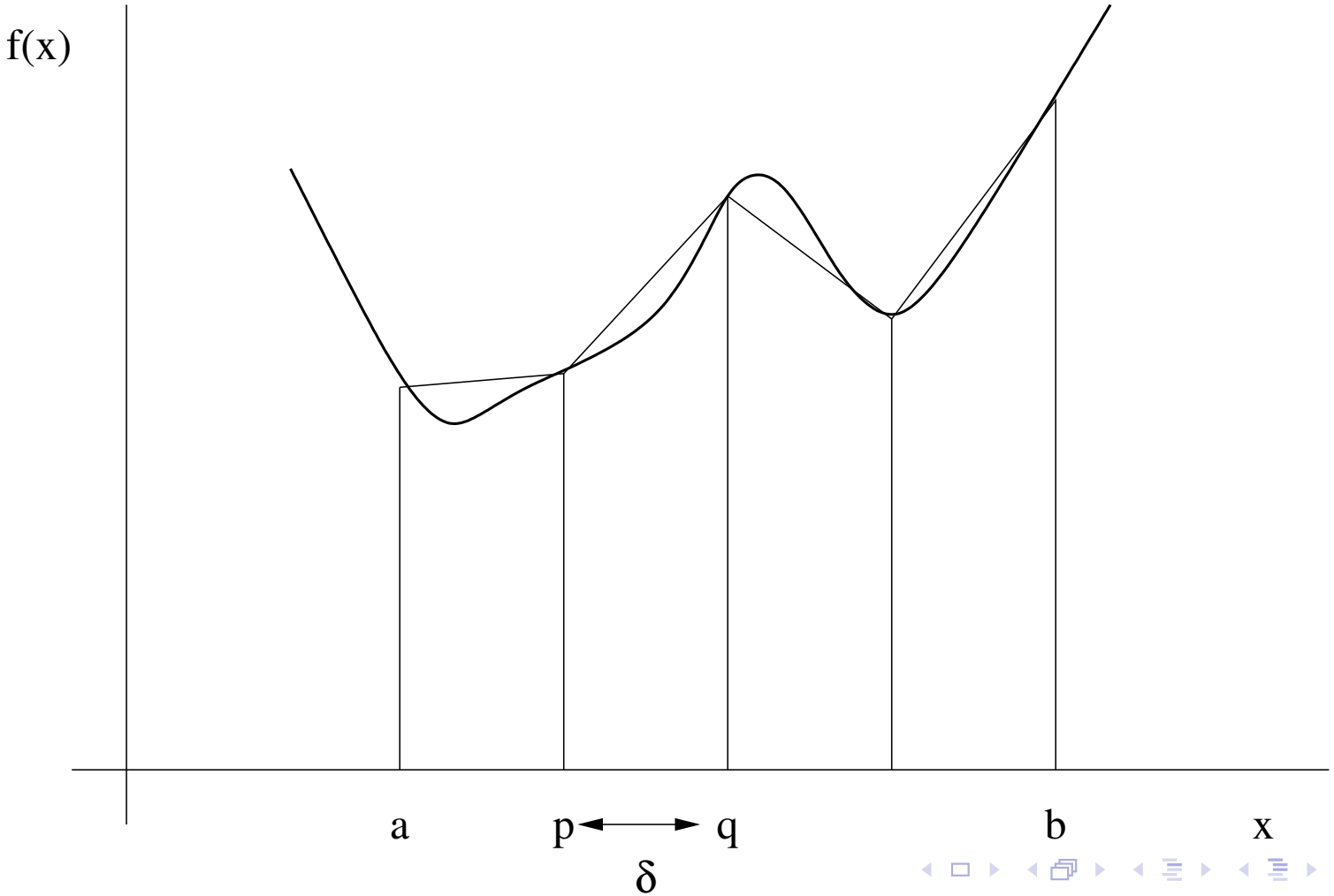
Analysis

Australian National University

- initial partitioning and distribution
  $t_{\text{comm1}} = pt_s + t_w n$
- sort into small buckets
  $t_{\text{comp2}} = n/p$
- send to large buckets: (overlapping communications)
  $t_{\text{comm3}} = (p-1)(t_s + (n/p^2)t_w)$
- sort of large buckets
  $t_{\text{comp4}} = (n/p)\lg(n/p)$
- total
  $t_{\text{p}} = pt_s + nt_w + n/p + (p-1)(t_s + (n/p^2)t_w) + (n/p)\lg(n/p)$
- at best $O(n)$
- what would be the worse case scenario?

Example#3: Integration

- consider the evaluation of an integral using the trapezoidal rule
  $I = \int_a^b f(x)dx$



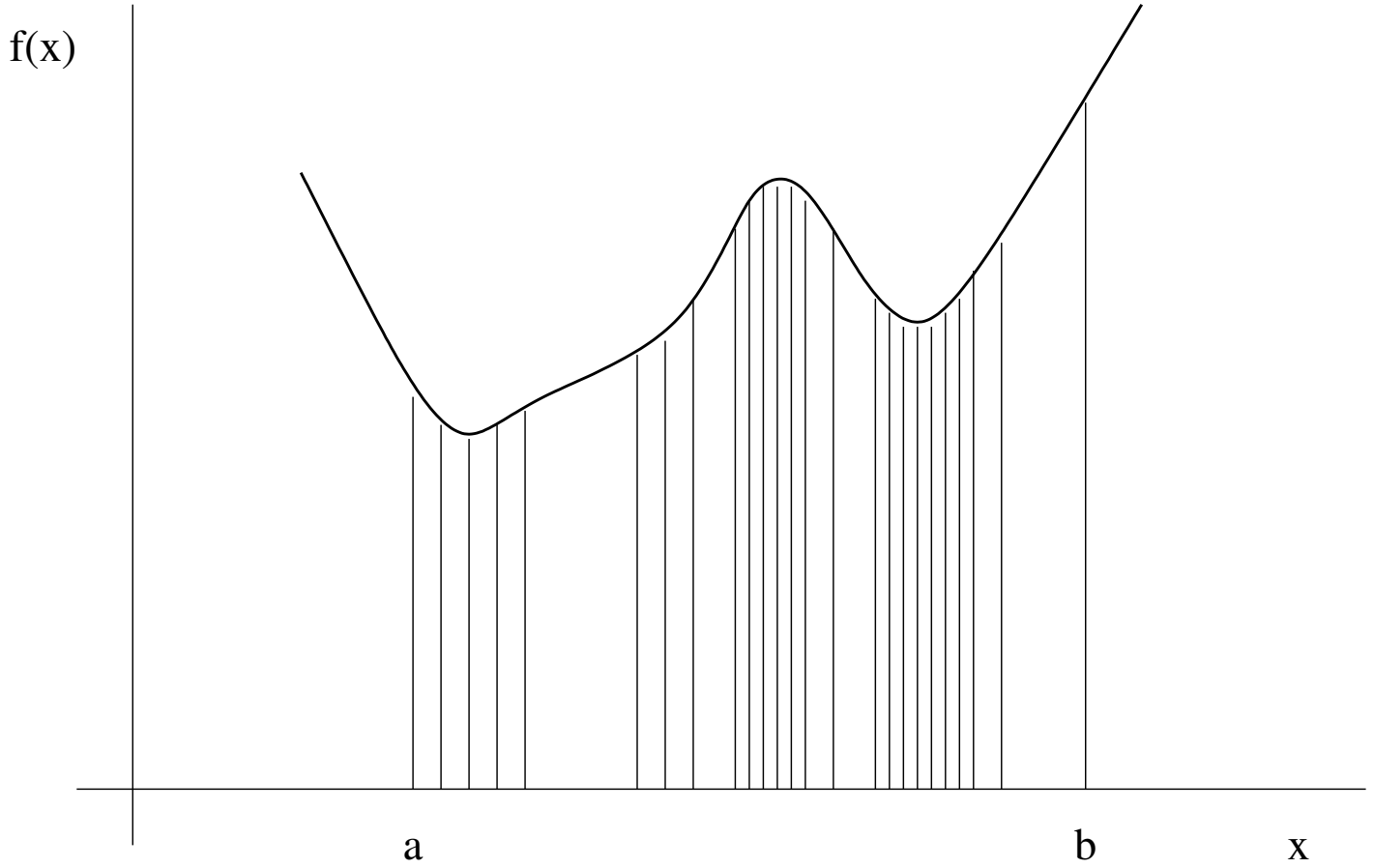
Static Distribution: SPMD Model

Australian National University

```
1  if (process_id == master) {
     printf("Enter number of regions\n");
3    scanf("%d", &n);
   }
5  broadcast(&n, master, p_group)
   region = (b-a)/p;
7  start = a + region*process_id;
   end = start + region;
9  d = (b-a)/n;
   area = 0.0;
11 for (x = start; x < end; x = x + d)
     area = area + 0.5 * (f(x) + f(x+d)) * d;
13 reduce_add(&area, master, p_group);
```

Adaptive Quadrature

- not all areas require the same number of points
- when to terminate division into smaller areas is an issue
- the parallel code will have uneven workload



Example#4: N-Body Problems

Australian National University

- summing long-range pairwise interactions, e.g. gravitation
  $F = \frac{Gm_a m_b}{r^2}$
  where G is the gravitational constant, m_a and m_b are the mass of two bodies, and r is the distance between them
- in Cartesian space:

$$F_x = \frac{Gm_a m_b}{r^2}\left(\frac{x_b - x_a}{r}\right)$$
$$F_y = \frac{Gm_a m_b}{r^2}\left(\frac{y_b - y_a}{r}\right)$$
$$F_z = \frac{Gm_a m_b}{r^2}\left(\frac{z_b - z_a}{r}\right)$$

- what is the total force on the sun due to all other stars in the milky way?
- given the force on each star we can calculate their motions
- molecular dynamics is very similar but the long forces are electrostatic

Simple Sequential Force Code

Australian National University

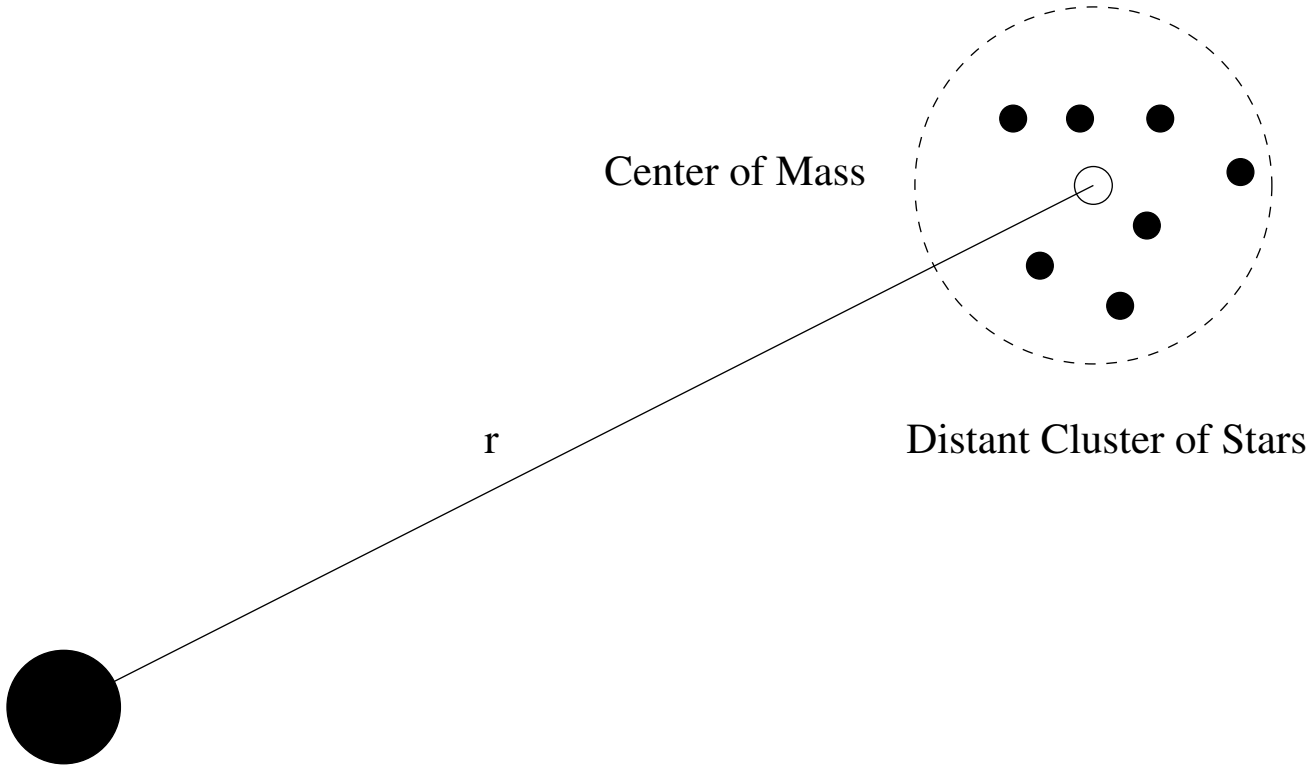
```
1  for (i = 0; i < n; i++)
     for (j = 0; j < n; j++) {
3      if (i != j) {
         rij2 = (x[i]-x[j])*(x[i]-x[j])
5             + (y[i]-y[j])*(y[i]-y[j])
              + (z[i]-z[j])*(z[i]-z[j]);
7        Fx[i] = Fx[i] + G*m[i]*m[j]/rij2 * (x[i]-x[j]) / sqrt(rij2);
         Fy[i] = Fy[i] + G*m[i]*m[j]/rij2 * (y[i]-y[j]) / sqrt(rij2);
9        Fz[i] = Fz[i] + G*m[i]*m[j]/rij2 * (z[i]-z[j]) / sqrt(rij2);
       }
11   }
```

- aside: how could you improve this sequential code?
- $O(n^2)$ - this will get very expensive for large n
- is there a better way?

Clustering

Australian National University

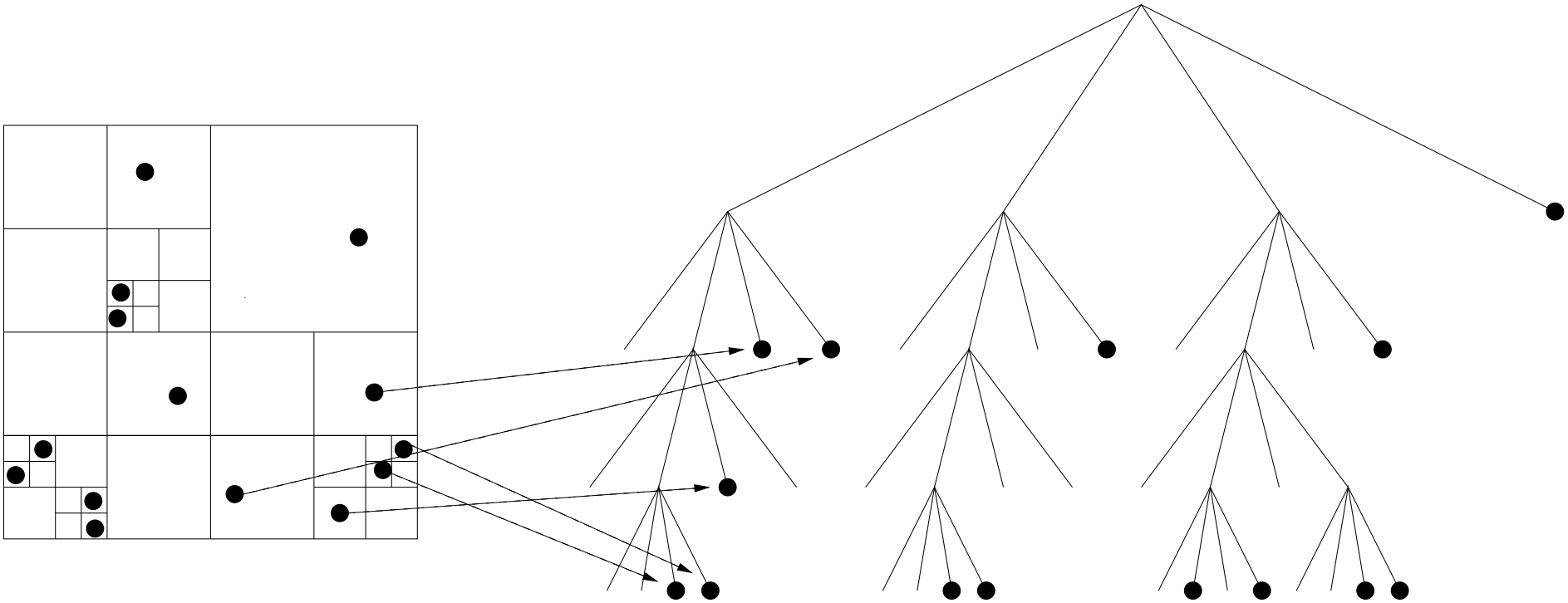
- idea: the interaction with several bodies that are clustered together but are located at large r for another body can be replaced by the interaction with the center of mass of the cluster



Barnes-Hut Algorithm

- start with whole space in one cube
  - divide the cube into 8 sub-cubes
  - delete sub-cubes if they have no particles in them
  - sub-cubes with more than 1 particle are divided into 8 again
  - continue until each cube has only one particle (or none)
- this process creates an **oct-tree**
- total mass and centre of mass of children sub-cubes is stored at each node
- force is evaluated by starting at the root and traversing the tree, BUT stopping at a node if the clustering algorithm can be used
- scaling is $O(n \log n)$
- load balancing likely to be an issue for parallel code

Barnes-Hut Algorithm: 2D Illustration



How to (evenly?) distribute such a structure? How often to re-distribute? (very hard problem!)

Hands-on Exercise: Bucket Sort

Outline

Overview: Synchronous Computations

Australian National University

- degrees of synchronization
- synchronous example 1: Jacobi Iterations
  - serial and parallel code, performance analysis
- synchronous example 2: Heat Distribution
  - serial and parallel code
  - comparison of block and strip partitioning methods
  - safety
  - ghost points

Ref: Chapter 6: Wilkinson and Allen

Degrees of Synchronization

- from fully to loosely synchronous
  - the more synchronous your computation, the more potential overhead
- **SIMD**: synchronized at the instruction level
  - provides ease of programming (one program)
  - well suited for data decomposition
  - applicable to many numerical problems
  - the `forall` statement was introduced to specify **data parallel** operations

```
forall (i = 0; i < n; i++) {
  data parallel work
}
```

Synchronous Example: Jacobi Iterations

- the Jacobi iteration solves a system of linear equations iteratively

$$
\begin{aligned}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \cdots \quad & a_{n-1,n-1}x_{n-1} = b_{n-1} \\
\vdots & \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 + \cdots \quad & a_{2,n-1}x_{n-1} = b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 + \cdots \quad & a_{1,n-1}x_{n-1} = b_1 \\
a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 + \cdots \quad & a_{0,n-1}x_{n-1} = b_0
\end{aligned}
$$

where there are n equations and n unknowns $(x_0, x_1, x_2, \cdots x_{n-1})$

Jacobi Iterations

- consider equation i as:

$$a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 + \cdots a_{i,n-1}x_{n-1} \quad = \quad b_i$$

  which we can re-cast as: $x_i = (1/a_{i,i})[b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 + \cdots a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} + \cdots a_{i,n-1}x_{n-1})]$

  i.e.

$$x_i = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j}x_j\right]$$

- strategy: guess x, then iterate and hope it converges!
  - converges if the matrix is **diagonally dominant**: $\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|$
- terminate when convergence is achieved:

$$|x^t - x^{t-1}| < \text{error tolerance}$$

Sequential Jacobi Code

- ignoring convergence testing:

```
1   for (i = 0; i < n; i++)
      x[i] = b[i];
3   for (iter = 0; iter < max_iter; iter++) {
      for (i = 0; i < n; i++) {
5       sum = -a[i][i]*x[i];
        for (j = 0; j < n; j++){
7           sum = sum + a[i][j]*x[j]
        }
9       new_x[i] = (b[i] - sum) / a[i][i];
      }
11    for (i = 0; i < n; i++)
        x[i] = new_x[i];
13  }
```

Parallel Jacobi Code

- ignoring convergence testing and assuming parallelisation over n processes:

```
1  x[i] = b[i];
   for (iter = 0; iter < max_iter; iter++) {
3    sum = -a[i][i] * x[i];
     for (j = 0; j < n; j++){
5        sum = sum + a[i][j]*x[j]
     }
7    new_x[i] = (b[i] - sum) / a[i][i];
     broadcast_gather(&new_x[i], new_x);
9    global_barrier();
     for (i = 0; i < n; i++)
11     x[i] = new_x[i];
   }
```

- `broadcast_gather()` sends the local `new_x[i]` to all processes and collects their new values

  Question: do we really need the barrier as well as this?

Partitioning

Australian National University

- normally the number of processes is much less than the number of data items
  - **block partitioning**: allocate groups of consecutive unknowns to processes
  - **cyclic partitioning**: allocate in a round-robin fashion
- analysis: τ iterations, n/p unknowns per process
  - computation – decreases with p

$$t_{\text{comp}} = \tau(2n+4)(n/p)t_f$$

  - communication – increases with p

$$t_{\text{comm}} = p(t_s + (n/p)t_w)\tau = (pt_s + nt_w)\tau$$

  - total - has an overall minimum

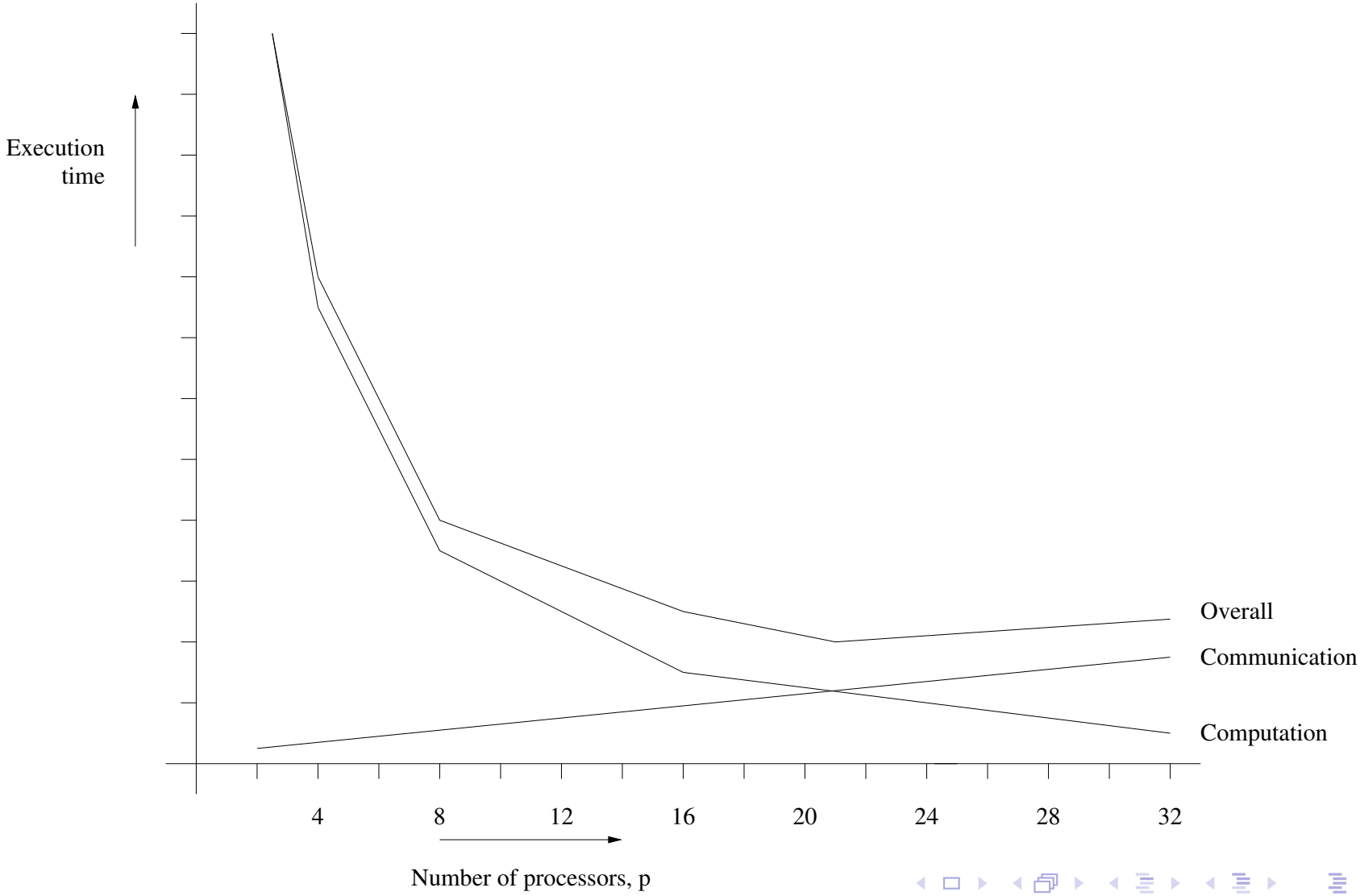
$$t_{\text{tot}} = ((2n+4)(n/p)t_f + pt_s + nt_w)\tau$$

  - question: can we do an all-gather faster than $pt_s + nt_w$?

Parallel Jacobi Iteration Time

Australian National University

Parameters: $t_s = 10^5 t_f, t_w = 50 t_f, n = 1000$



Locally Synchronous Example: Heat Distribution Problem

Consider a metal sheet with a fixed temperature along the sides but unknown temperatures in the middle – find the temperature in the middle.

- finite difference approximation to the Laplace equation:

$$\frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2} = 0$$

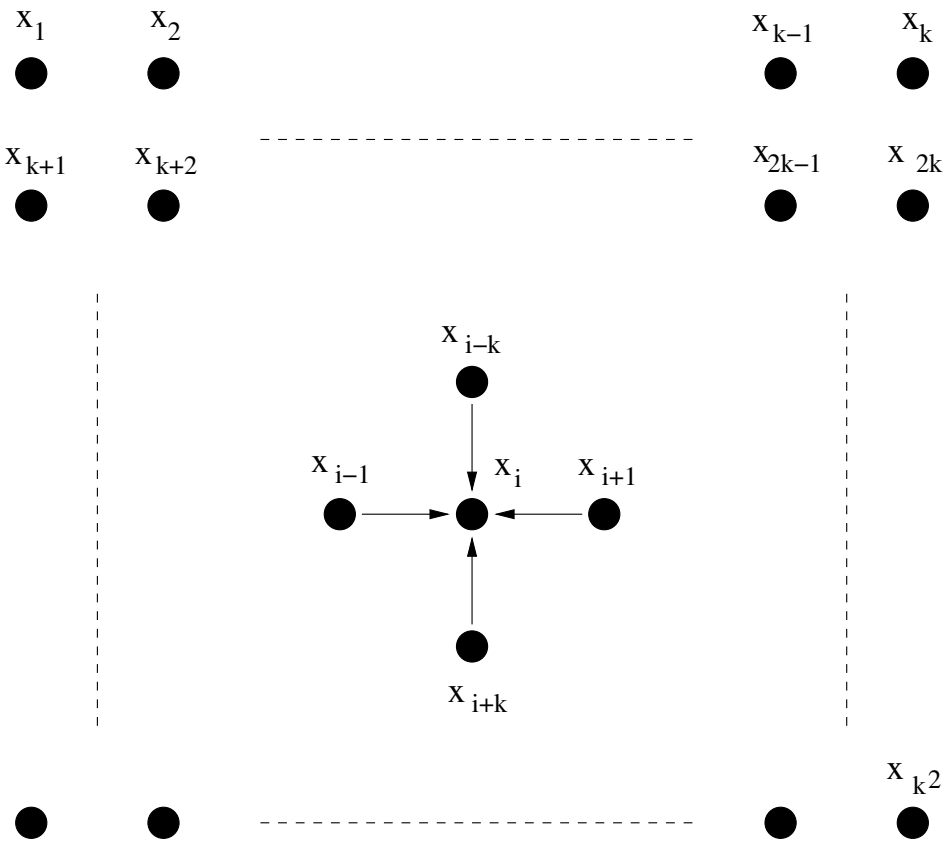
$$\frac{T(x+\delta x,y) - 2T(x,y) + T(x-\delta x,y)}{\delta x^2} + \frac{T(x,y+\delta y) - 2T(x,y) + T(x,y-}{\delta y^2}$$

- assuming an even grid (i.e. $\delta x = \delta y$) of $n \times n$ points (denoted as $h_{i,j}$), the temperature at any point is an average of surrounding points:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

- problem is very similar to the Game of Life, i.e. what happens in a cell depends upon its neighbours

Array Ordering



- we will solve iteratively: $x_i = \frac{x_{i-1}+x_{i+1}+x_{i-k}+x_{i+k}}{4}$
- but this problem may also be written as a system of linear equations:

$$x_{i-k} + x_{i-1} - 4x_i + x_{i+1} + x_{i+k} = 0$$

Heat Equation: Sequential Code

- assume a fixed number of iterations and a square mesh
- beware of what happens at the edges!

```
for (iter = 0; iter < max_iter; iter++) {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
      g[i][j] = 0.25*(h[i-1][j] + h[i+1][j] +
                      h[i][j-1] + h[i][j+1]);
    for (i = 1; i < n; i++)
      for (j = 1; j < n; j++)
        h[i][j] = g[i][j];
}
```

Heat Equation: Parallel Code

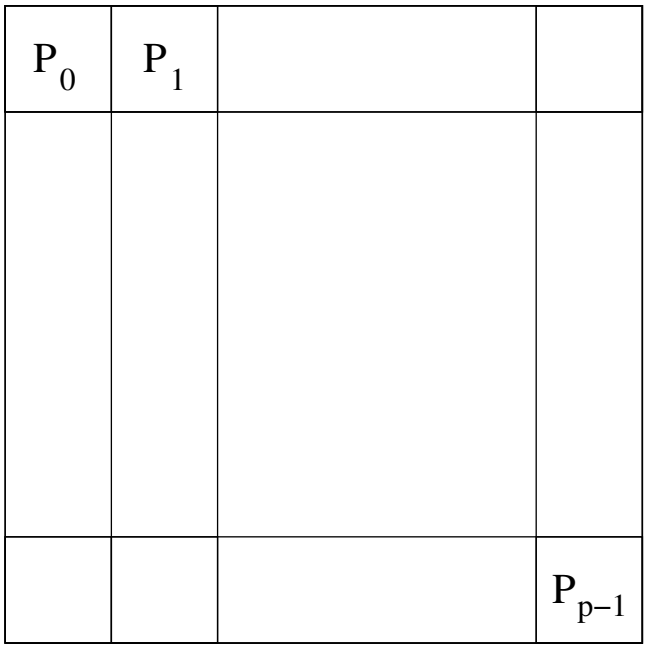
- one point per process
- assuming **locally-blocking sends**:

```
1  for (iter = 0; iter < max_iter; iter++) {
     g = 0.25*(w + x + y + z);
3    send(&g, P(i-1,j));
     send(&g, P(i+1,j));
5    send(&g, P(i,j-1));
     send(&g, P(i,j+1));
7    recv(&w, P(i-1,j));
     recv(&x, P(i+1,j));
9    recv(&y, P(i,j-1));
     recv(&z, P(i,j+1));
11 }
```

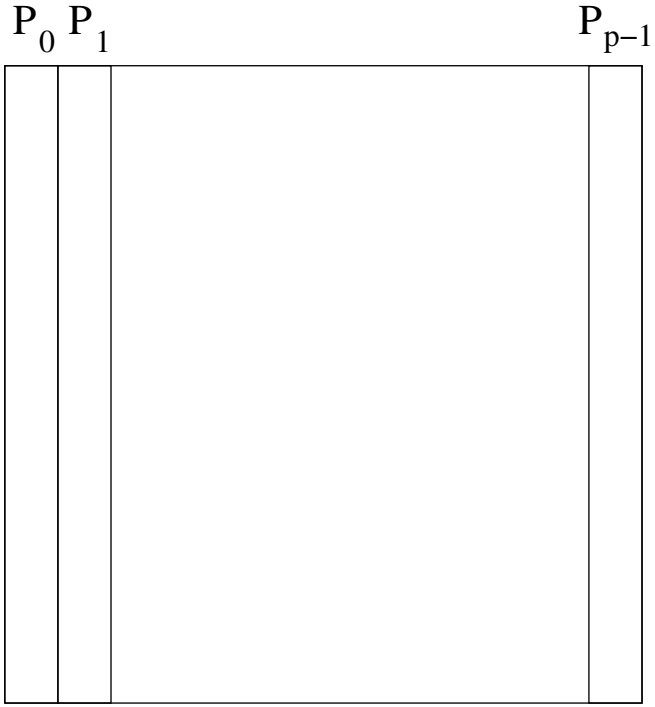
- sends and receives provide a local barrier
  - each process synchronizes with 4 others surrounding processes

Heat Equation: Partitioning

- normally more than one point per process
- option of either **block** or **strip partitioning**



Block Partitioning



Strip Partitioning

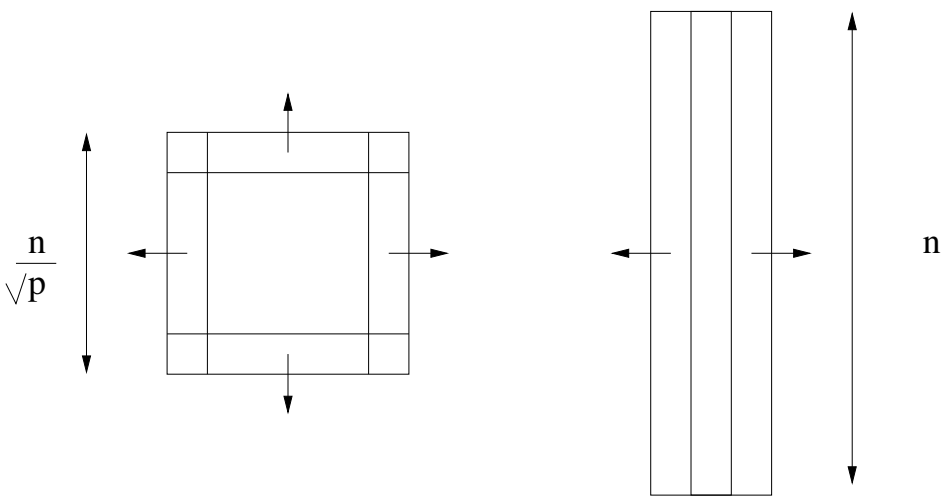
Block/Strip Communication Comparison

- **block partitioning**: four edges exchanged (n^2 data points, p processes)

$$t_{\text{comm}} = 8(t_s + \frac{n}{\sqrt{p}} t_w)$$

- **strip partitioning**: two edges exchanged

$$t_{\text{comm}} = 4(t_s + nt_w)$$



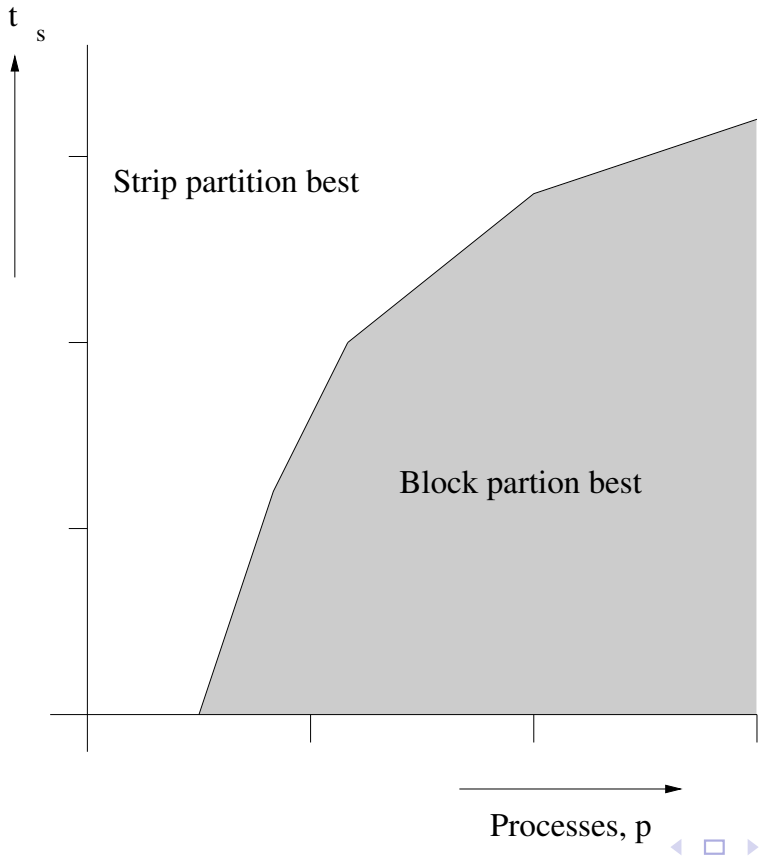
Block Communications Strip Communications

Block/Strip Optimum

- **block** communication is larger than **strip** if:

  $8\left(t_s + \frac{n}{\sqrt{p}}t_w\right) > 4(t_w + nt_w)$

  i.e. if $t_s > n\left(1 - \frac{2}{\sqrt{p}}\right)t_w$



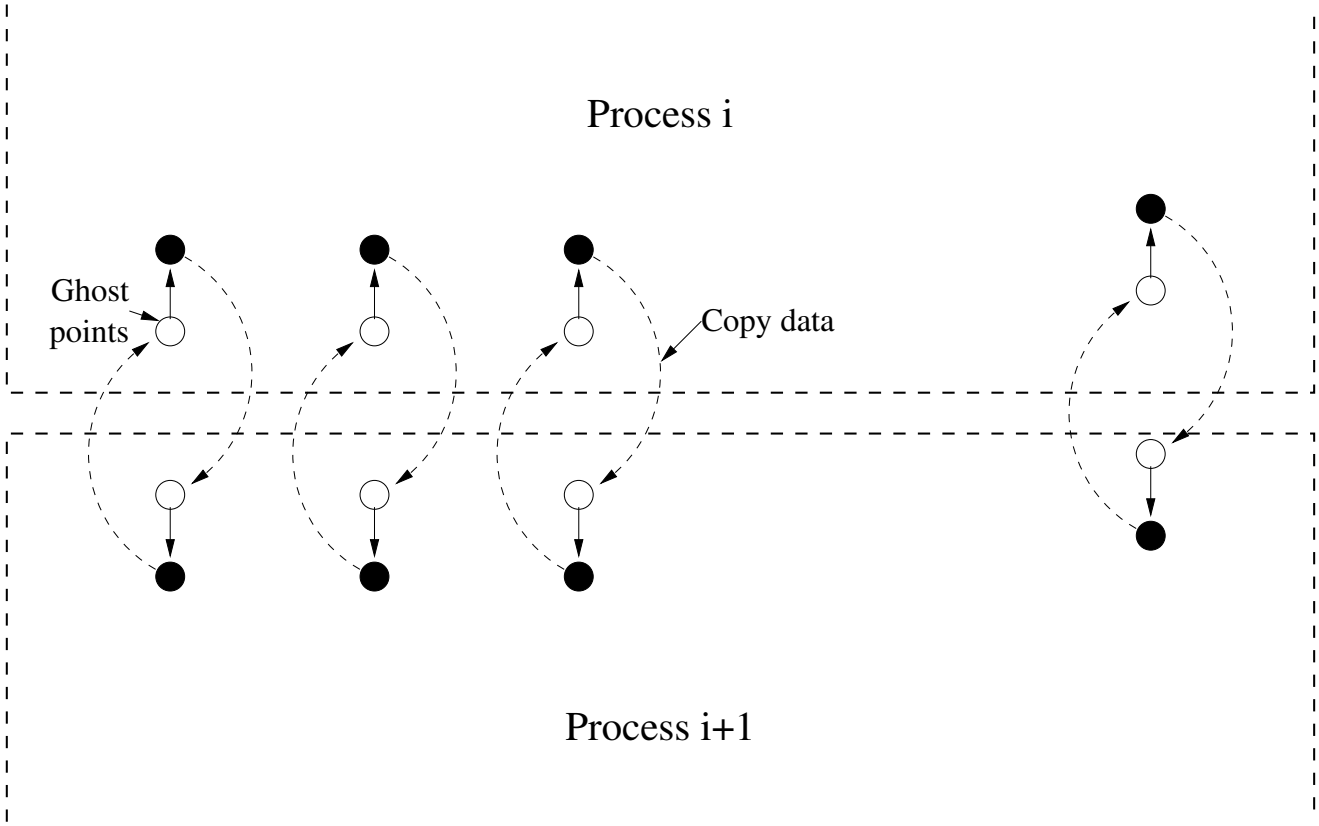
Safety and Deadlock

- with all processes sending and then receiving data, the code is unsafe: it relies on **local buffering** in the `send()` function
  - potential for deadlock (as in Prac 1, Ex 3)!
- alternative #1: re-order sends and receives
  e.g. for **strip partitioning**:

```
if ((myid % 2) == 0){
  send(&g[1][1], n, P(i-1));
  recv(&h[1][0], n, P(i-1));
  send(&g[1][n], n, P(i+1));
  recv(&h[1][n+1], n, P(i+1));
} else {
  recv(&h[1][0], n, p(i-1));
  send(&g[1][1], n, p(i-1));
  recv(&h[1][n+1], n, p(i+1));
  send(&g[1][n], n, p(i+1));
}
```

Alt# 2: Asynchronous Comm. using Ghostpoints

- assign extra receive buffers for edges where data is exchanged
  - typically these are implemented as extra rows and columns in each process' local array (known as a **halo**)
- can use asynchronous calls (e.g. `MPI_Isend()`)

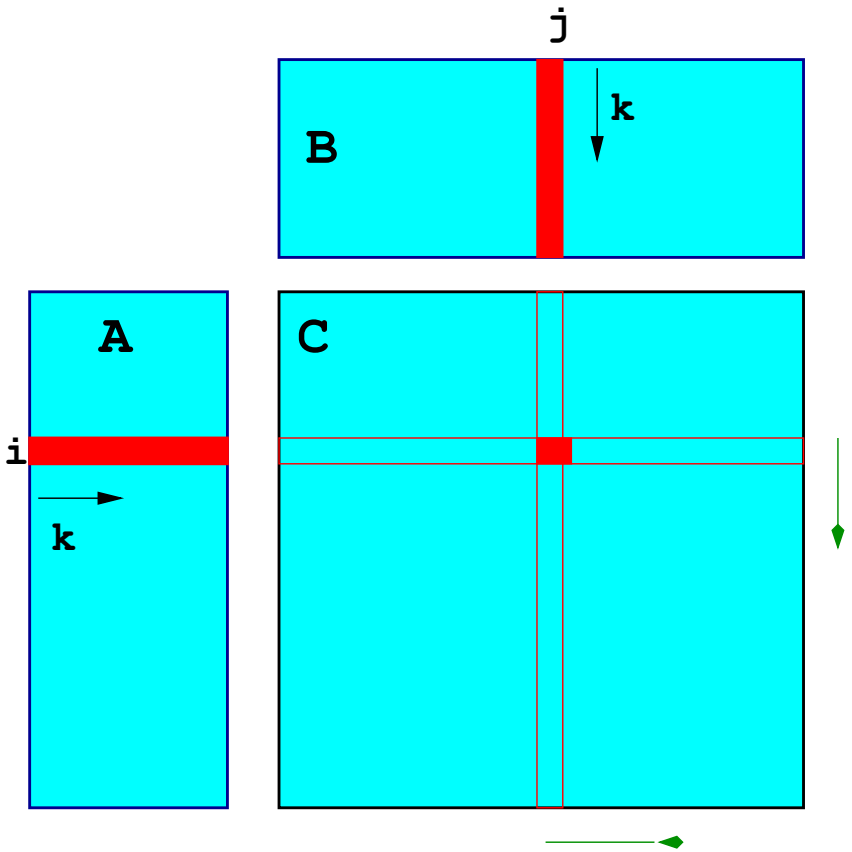


Hands-on Exercise: Synchronous Computations

Australian National University

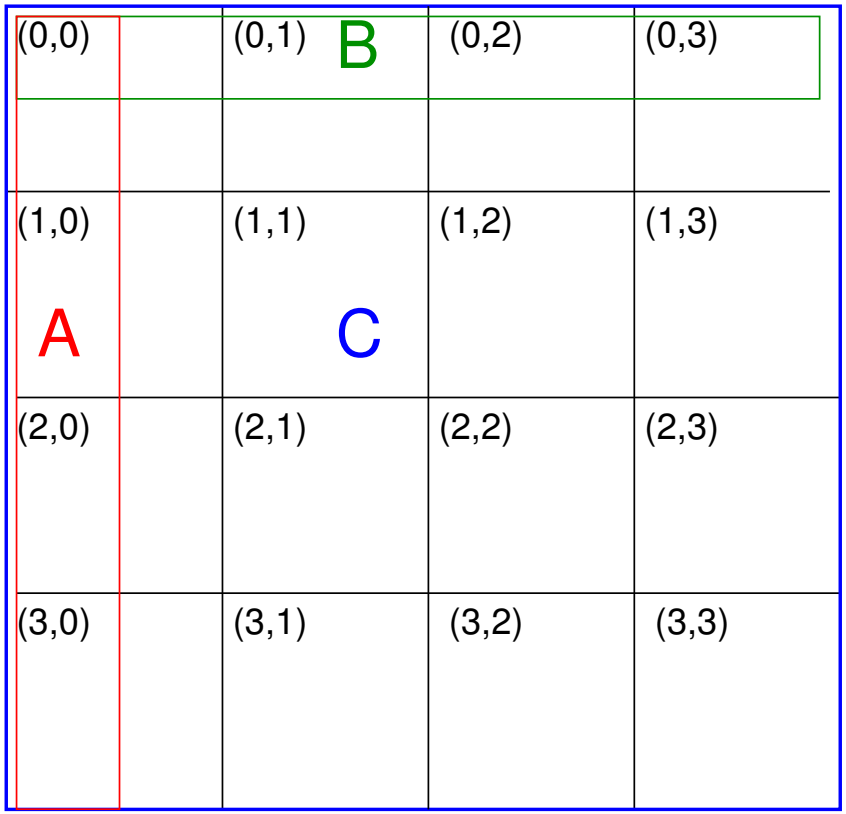
Outline

Matrix Multiplication



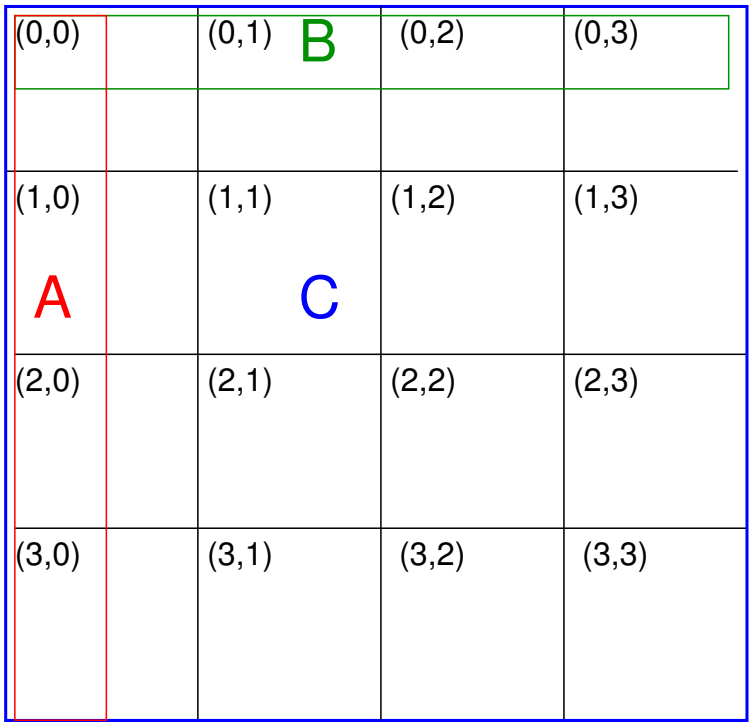
- matrix multiplication is the dominant computation in linear algebra and neural net training
  - tensor operations are broken down to matrix operations on TPUs and GPUs!
- C += AB, where A, B, C are $M \times K, K \times N, M \times N$ matrices, respectively
- $c_{i,j}$ += $\sum_{k=0}^{K-1} a_{i,k} b_{k,j}$, for $0 \leq i < M, 0 \leq j < N$
- we will primarily consider the case $M = N, K < N$

Matrix Process Topologies and Distributions



- use a logical two-dimensional $p = p_y \times p_x$ process grid
  - e.g. a process with ID rank r has a 2D rank (r_y, r_x), where $r = r_y p_x + r_x$, $0 \leq r_x < p_x, 0 \leq r_y < p_y$
  - for performance, $\frac{p_y}{p_x} \approx \frac{M}{N}$ is generally optimal (best local multiply speed, lowest communication volume)
- here, a **block distribution** over the whole 4×4 process grid is used for C
- notice that A (B) must be aligned to be on the same process rows (columns) as C

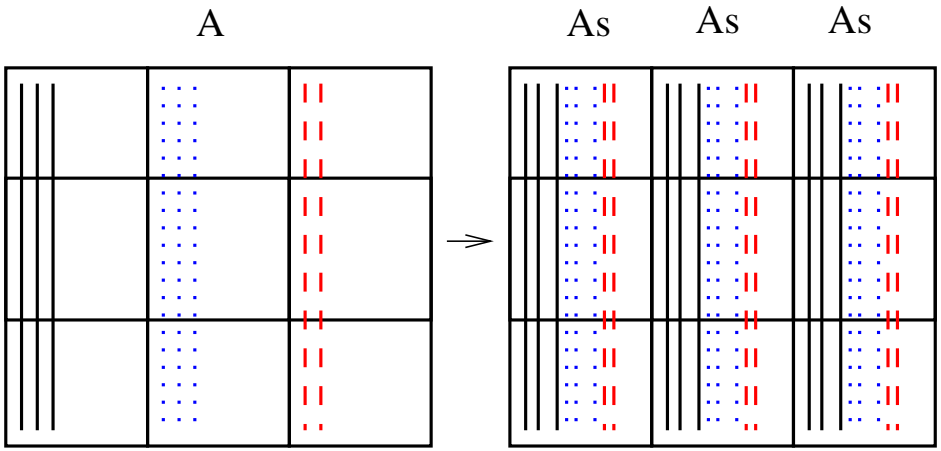
Rank-K Matrix Multiply: Simple Case



- consider the simplest case where K is small enough and $A(B)$ are distributed block-wise over a single process column (row)
- assume local matrix sizes are $m = \frac{M}{p_y}, n = \frac{N}{p_x}$
- denoting `A`, `B`, `C` as local portions of the matrices, the parallel multiply can be done by:
  1. row-broadcast from col $r_x = 0$ of `A` (size $m \times K$ result in `As`)
  2. column-broadcast for row $r_y = 0$ of `B` (size $K \times n$, result in `Bs`)
  3. perform local matrix multiply of `As`, `Bs` and `C`

Rank-K Matrix Multiply: General Case

All processes may have some columns of the distributed matrix A



- e.g. matrix A with $K = 8$, distributed across a 3×3 process grid
- each process column must broadcast its portion, storing result in `As` (a 'spread' of the K-dim. of A)

Algorithm now becomes:

1. row-wise all-gather of `A` (result in `As`, size $m \times K$)
2. column-wise all-gather of `B` (result in `Bs`, size $K \times n$)
3. perform local matrix multiply of `As`, `Bs` and `C`

If $K \not| p_x, p_y$, we can 'pad out' matrices (or use `MPI_Allgatherv()`).
If K is large, we can reduce the size of `As` and `Bs`, by breaking this down into stages.

2D Process Topology Support in MPI

This involves creating a periodic 2D cartesian topology, followed by row and column communicators:

```
int np, rank, px, py;
MPI_Comm comm2D, commRow, commCol; int dims[2], rank2D, r2D[2];
int periods[] = {1,1}, rowSpec[] = {1,0}, colSpec[] = {0,1};
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_rank(MPI_COMM_WORLD, &np);
px = ...; py = np / px; assert (px * py == np);
dims[0] = px, dims[1] = py;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm2D);
MPI_Comm_rank(comm2D, &rank2D); //likely that rank2d==rank
MPI_Cart_coords(comm2D, rank2D, 2, r2D);
// create this process' 1D row / column communicators
MPI_Cart_sub(comm2D, rowSpec, &commRow); //of size px
MPI_Cart_sub(comm2D, colSpec, &commCol); //of size py
MPI_Comm_rank(commRow, &rx); MPI_Comm_rank(commCol, &ry);
assert (rx == r2D[0]  &&  ry == r2D[1]);
```

MPI Rank-K Update Algorithm

- an $m \times n$ local matrix C can be represented as the pair (`C`, `ldC`), where `double *C` points to the 0th element, the leading dimension `int ldC` $\geq m$ and $c_{i,j}$ is stored `C[i + j*ldC]`
- defining a datatype for A will avoid an explicit pack operation:

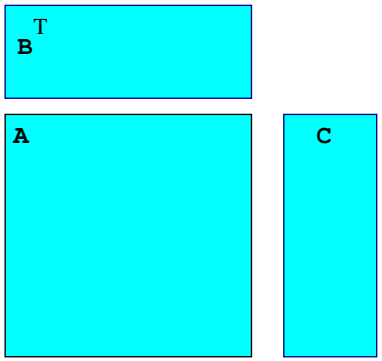
```
MPI_Datatype aCol;
MPI_Type_vector(1, m, ldA, MPI_DOUBLE, &aCol);
MPI_Type_commit(&aCol);
```

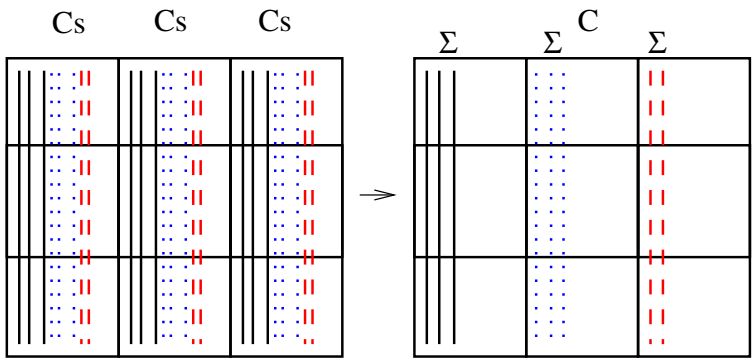
- in order to use `Bs` directly for a local matrix multiply (`dgemm()`) we must transpose and pack B:

```
double Bt[n*kB], As[m*K], Bs[n*K]; int i, j;
for (i=0; i < kB; i++)
  for (j=0; j < n; j++)
    Bt[j + i*n] = B[i + j*ldB];
MPI_Allgather(A, kA, aCol, As, m*kA, MPI_DOUBLE, commRow);
MPI_Allgather(Bt, n*kB, MPI_DOUBLE, Bs, n*kB, MPI_DOUBLE,
    commCol);
dgemm_("NoTrans", "Trans", m, n, K, As, m, Bs, n, C, ldC);
```

(non-unit stride on the K-dim. is generally non-optimal for the multiply)

Matrix Multiplication: AB^T Case





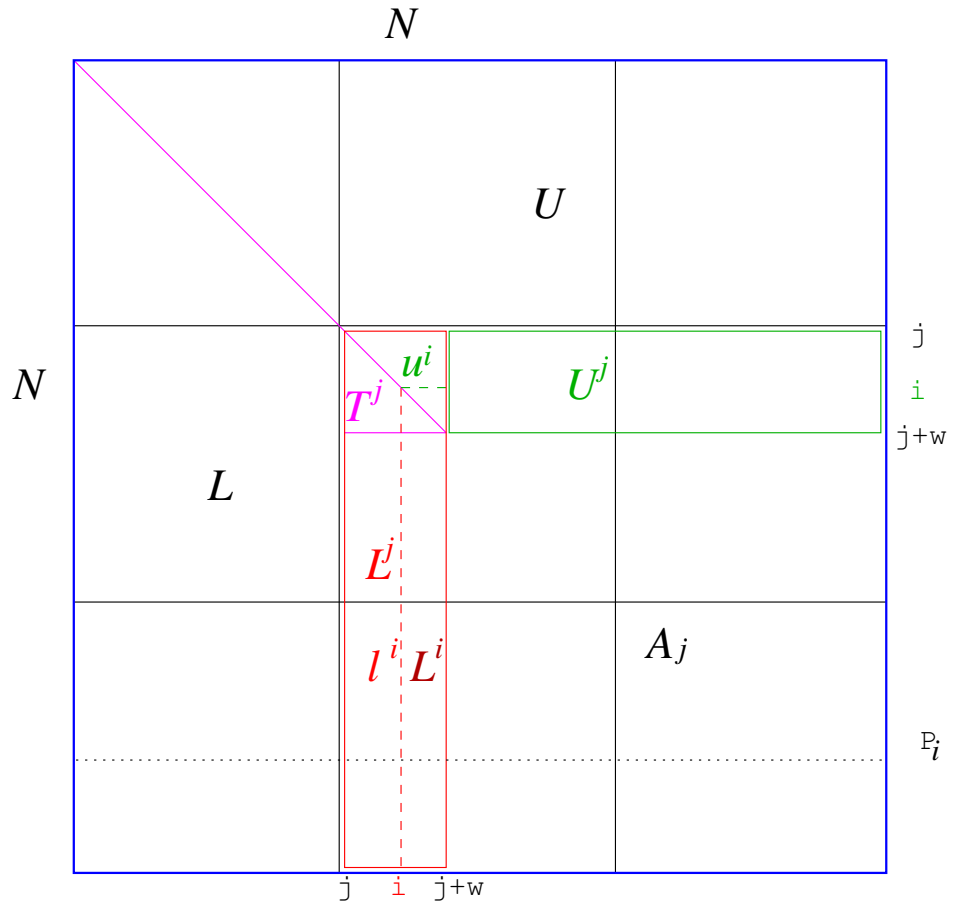
- consider C += AB^T, where $N < K = M$
- this variant of the algorithm is:
  1. column-wise all-gather of `B` (result in `Bs`, size $N \times kB$)
  2. create a workspace `Cs` of size $m \times N$
  3. perform local matrix multiply of `A`, `Bs` and `Cs`
  4. row-wise reduce-scatter of `Cs` (add result to `C`, size $m \times nC$)

- there is an analogous variant for C += $A^T B$, efficient for $M < K = N$
- general matrix multiply algorithm: choose variant involving least data movement, performing a global transposition if needed

Parallel Blocked-Partitioned Matrix Algorithms

Australian National University

- blocked LU factorization (LINPACK) rated in the Top 10 Algorithms of the 20th Century (CS&E, Jan 2000)
  - symmetric eigenvalue algorithm arguably even more important
  - idea: express vector operations of original algorithm into blocks
    - majority of operations are now matrix-matrix
    - transforms algorithm from data-access to computation bound
  - normal block distribution inadequate for || algorithms!

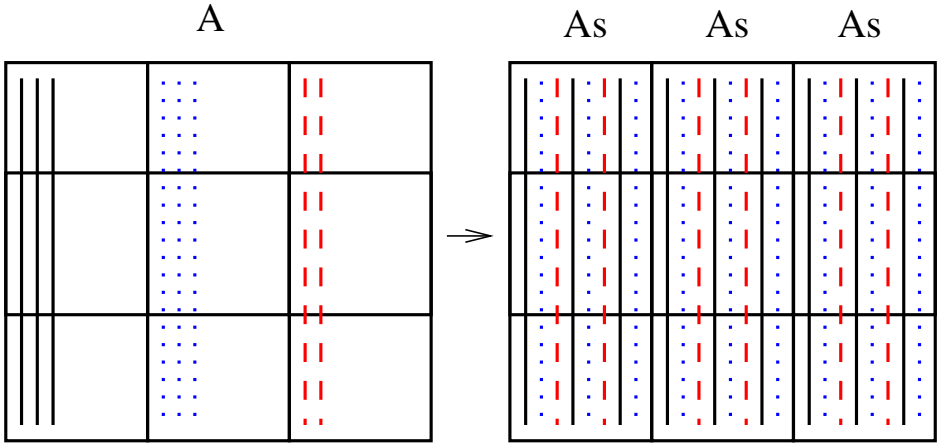


Matrix Distributions: Block-Cyclic

- is 'standard' for || LA (load balance on triangular & sub- matrices)
- divide the global matrix A into $b_y \times b_x$ blocks on a $P \times Q$ array; if block $(0,0)$ is on process (r_y, r_x), block (i,j) is on process $((i + r_y)\%p_y, (j + r_x)\%p_x)$
- e.g. $r_y = r_x = 0, b_y = 3, b_x = 2$ on a 2×3 array, a 10×10 matrix A:

a_{00}	a_{01}	a_{06}	a_{07}	a_{02}	a_{03}	a_{08}	a_{09}	a_{04}	a_{05}
a_{10}	a_{11}	a_{16}	a_{17}	a_{12}	a_{13}	a_{18}	a_{19}	a_{14}	a_{15}
a_{20}	a_{21}	a_{26}	a_{27}	a_{22}	a_{23}	a_{28}	a_{29}	a_{24}	a_{25}
a_{60}	a_{61}	a_{66}	a_{67}	a_{62}	a_{63}	a_{68}	a_{69}	a_{64}	a_{65}
a_{70}	a_{71}	a_{76}	a_{77}	a_{72}	a_{73}	a_{78}	a_{79}	a_{74}	a_{75}
a_{80}	a_{81}	a_{86}	a_{87}	a_{82}	a_{83}	a_{88}	a_{89}	a_{84}	a_{85}
a_{30}	a_{31}	a_{36}	a_{37}	a_{32}	a_{33}	a_{38}	a_{39}	a_{34}	a_{35}
a_{40}	a_{41}	a_{46}	a_{47}	a_{42}	a_{43}	a_{48}	a_{49}	a_{44}	a_{45}
a_{50}	a_{51}	a_{56}	a_{57}	a_{52}	a_{53}	a_{58}	a_{59}	a_{54}	a_{55}
a_{90}	a_{91}	a_{96}	a_{97}	a_{92}	a_{93}	a_{98}	a_{99}	a_{94}	a_{95}

Matrix Multiply on the Block-Cyclic Distribution



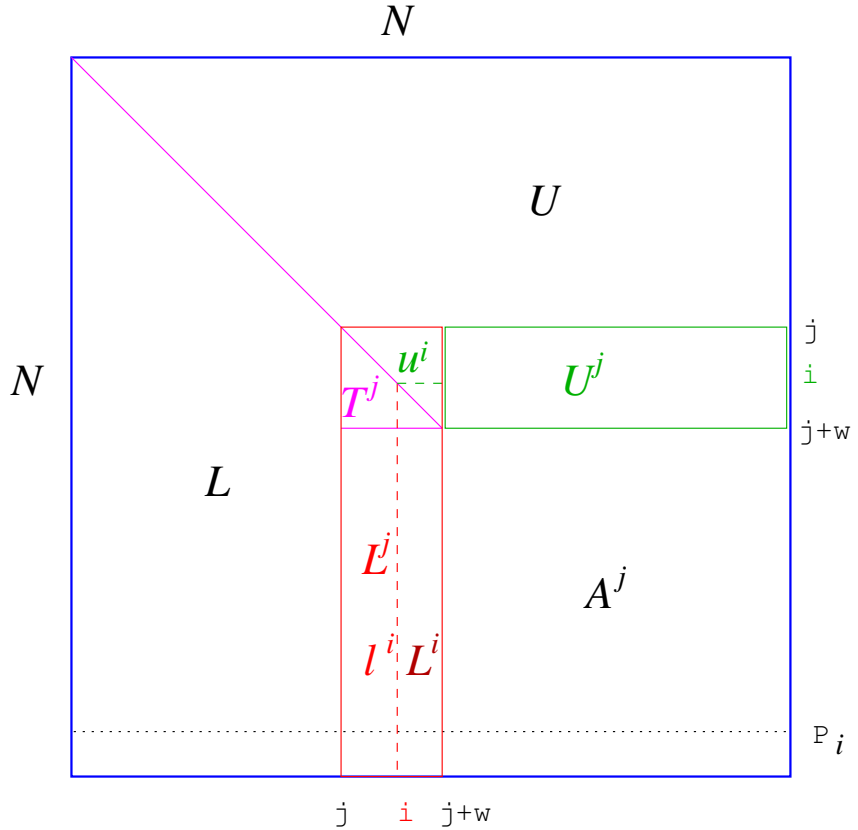
- e.g. matrix A with $K = 8$, distributed across a 3×3 process grid
- each process column must broadcast its portion, storing result in `As` (a 'spread' of the K-dim. of A)

The 'spread' of the K-dim. of A should respect the global order of indices

- `MPI_Allgather()` will not give this order
- unless $p_x = p_y$ and $b_x = b_y$, must re-order columns in `As`, `Bs` before calling `dgemm()`

Blocked LU Factorization Algorithm

- **right-looking** variant using **partial pivoting** on an $N \times N$ matrix A




```
for (j=0; j < N; j+=w)
  for (i=j; i < j+w; i++)
    find P[i] s.t. |A_{P[i], i}| ≥ |A_{i:N-1, i}|
    A_{i, j:j'-1} ↔ A_{P[i], j:j'-1}   (j' = j+w)
    l^i ← l^i / A_{i,i}
    L^i ← L^i − l^i u^i
  for (i = j; i < j+w; i++)
    A_{i,:} ↔ A_{P[i],:}  (outside L^j)
  U^j ← (T^j)^{-1} U^j
  A^j ← A^j − L^j U^j
```

- $l^i = A_{i+1:N-1,\, i}$; $u^i = A_{i,\, i+1:j'-1}$; $L^i = A_{i+1:N-1,\, i+1:j'-1}$
- $(T^j) = A_{j:j'-1,\, j:j'-1}$ (lower triangular matrix, with unit diagonal)
- $L^j = A_{j':N-1,\, j:j'-1}$; $U^j = A_{j:j'-1,\, j':N-1}$; $A^j = A_{j':N-1,\, j':N-1}$

Blocked LU Factorization: Communication

Australian National University

- let r_x^i (r_y^i) denote the process row (column) rank holding $A_{i,i}$
- assume $w = b_x = b_y$ (one process row (column) holds U^j (L^j) panel)

```
for (j=0; j < N; j+=w)
```

  for (i=j; i < j+w; i++) (note $r_x^i = r_x^j, r_y^i = r_y^j$)

    find $P[i]$ s.t. $|A_{P[i],\,i}| \geq |A_{i:N-1,\,i}|$ (all-reduce on col. r_x^i)

    $A_{i,\,j:j'-1} \leftrightarrow A_{P[i],\,j:j'-1}$ (swap on processes (r_y^i, r_x^i) and $(r_y^{P[i]}, r_x^i)$)

    $l^i \leftarrow l^i / A_{i,i}$ (broadcast $A_{i,i}$ on col. r_x^i from row r_y^i)

    $L^i \leftarrow L^i - l^i u^i$ (broadcast l^i on col. r_x^i from row r_y^i)

  for (i = j; i < j+w; i++)

    $A_{i,:} \leftrightarrow A_{P[i],:}$ (swap on processes (r_y^i, r_x^i) and $(r_y^{P[i]}, r_x^i)$)

  $U^j \leftarrow (T^j)^{-1} U^j$ (broadcast T^j on col. r_x^j from row r_y^j)

  $A^j \leftarrow A^j - L^j U^j$ (row (col.) broadcast L^j (U^j) from col. r_x^j (row r_y^j))

- exercise: implement this using MPI and BLAS!
  - need to calculate local lengths for vector of length say $N - j$ from say process row r_y^j

Parallel Factorization Analysis and Methods

Australian National University

- performance is determined by **load balance** and **communication overhead** issues
- for an $N \times N$ matrix on a $p_x \times p_x$ process grid, || execution time is:
  $t(N) = c_1 N t_s + c_2 N^2/p_x t_w + c_3 N^3/p_x^2 t_f$
  As $c_1 = O(\lg_2 p_x) > c_2 > c_3$ and typically $\frac{t_s}{f_f} \approx 10^3$,
  the t_s term can be significant for small-moderate N/p_x.
  Note: t_s mainly due to software: several layers of function calls, error checking, message header formation, buffer allocation & search
- **storage blocking**: $(\omega = b_x = b_y)$
  - simplest to implement, minimizes number of messages
  - suffers from $O(b_x + b_y)$ load imbalance on panel formation:
    i.e. one processor column (row) holds L^i (U^i); also in $A^i \leftarrow A^i - L^i U^i$
- **algorithmic blocking**: can use `dgemm`-optimal ω, $b_x = b_y \approx 1$
  - greatly reduces these imbalances, ||izes row swaps
  - introduces $4N$ extra messages; local panel width is small ($\approx \omega/p_x$)
- **lookahead** (High Performance Linpack)
  - eliminates load imbalance in forming L^i by computing it in advance
  - hard to implement; only applicable to some computations

Summary

Topics covered today involve the message passing paradigm:

- issues in parallelizing 'embarrassingly parallel' problems
- parallelizing by domain decomposition
- synchronous computations (mainly using domain decomposition)
- case study: parallel matrix multiply and factorization

Tomorrow - the Partitioned Global Address Space paradigm

Hands-on Exercise: Matrix Multiply