ID1217 Concurrent Programming
Lecture 18

ROYAL INSTITUTE OF TECHNOLOGY

Paradigms for Process Interaction in Distributed Computing. Parallelism in Scientific Computing

Vladimir Vlassov
KTH/ICT/EECS

ROYAL INSTITUTE OF TECHNOLOGY

Paradigms for Process Interactions

- Basic interaction paradigms in distributed programs:
  1. Producer/consumer (filters) – one way
  2. Client/Server – two ways as master/slave
  3. Interacting peers – two ways as equals
- Larger paradigms (models) for process interactions
  1. Manager/Workers (a.k.a. “bag of tasks” or “work farm”)
  2. Heartbeat algorithms
  3. Pipeline algorithms
  4. Probe/echo algorithms; gossip algorithms
  5. Broadcast algorithms
  6. Token-passing algorithms
  - The first three paradigms are commonly used in parallel computing; the others arise in distributed systems.

1. Manager-Workers (Work Farm)

- Based on a **distributed bag-of-tasks** model
  - Manager maintains a bag of independent tasks and collects results
  - Each worker gets a task from the bag, executes it, and possible generates new tasks that it puts to the bag.
  - Manager acts as a server; workers – as clients.
  - How can the manager detect termination? every worker is waiting to get a new task (i.e. idle) and the bag is empty.
- Advantages:
  - **Scalability**: Easy to vary the number of workers, and granularity of tasks.
  - **Good load balancing**: Easy to ensure that each worker does about the same amount of work.

Manager/Worker (cont'd)

- Worker:

```
call getTask(…)   // request-response
compute;
generate a new task;
send putResult(…) // asynchronous send
```

- Manager exports operations "get task" and "put result"

```
in getTask(…) -> …  // returns a task or end
[] putResult(…) -> …      // receives result
ni
```

Example: Adaptive Quadrature Using Manager/Workers

```
module Manager
  op getTask(result double left, right);
  op putResult(double area);
body Manager
  process manager {
    double a, b;          # interval to integrate
    int numIntervals;     # number of intervals to use
    double width = (b-a)/numIntervals;
    double x = a, totalArea = 0.0;
    int tasksDone = 0;
    while (tasksDone < numIntervals) {
      in getTask(left, right) st x < b ->
          left = x; x += width; right = x;
      [] putResult(area) ->
          totalArea += area;
          tasksDone++;
      ni
    }
    print the result totalArea;
  }
end Manager

double f() { ... }          # function to integrate
double quad(...) { ... }    # adaptive quad function

process worker[w = 1 to numWorkers] {
  double left, right, area = 0.0;
  double fleft, fright, lrarea;
  while (true) {
    call getTask(left, right);
    fleft = f(left); fright = f(right);
    lrarea = (fleft + fright) * (right - left) / 2;
    # calculate area recursively as shown in Section 1.5
    area = quad(left, right, fleft, fright, lrarea);
    send putResult(area);
  }
}
```

2. Heartbeat Algorithms

- Divide work (evenly).
- Processes periodically exchange information using a **send (expand)** and then **receive (contract)** interaction.
  - Typically each proc exchanges data with its neighbors
  - The exchanges provide a "fuzzy" barrier among the workers
- Used in many iterative applications with data parallelism
  - E.g. exchange edges in Jacobi iteration

ROYAL INSTITUTE OF TECHNOLOGY

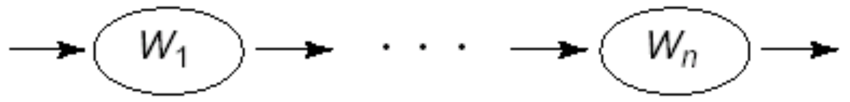
A Typical Heartbeat Algorithm

```
process Worker[i = 1 to numWorkers] {
  declarations of local variables;
  initialize local variables;
  while (not done) {
    send values to neighbors;
    receive values from neighbors;
    update local values;
  }
}
```

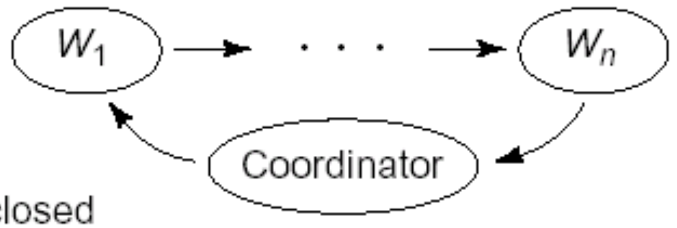
- Examples: Jacobi iteration, region labeling, game of life

3. Pipeline Algorithms

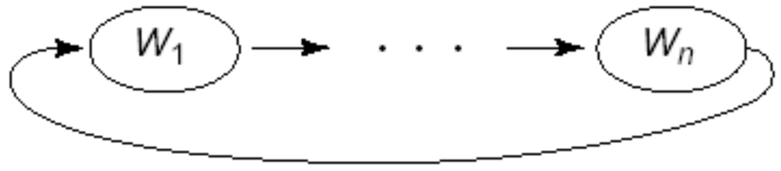
- What: divide work evenly, compute and circulate data among workers
- When: used when workers need all the data, not just edges from neighbors.
- Pipeline structures –circular or open (or closed)



(a) open



(b) closed



(c) circular

ROYAL INSTITUTE OF TECHNOLOGY

Example: Sieve of Eratosthenes

- A closed pipeline that generates prime numbers

```
chan sieve[n](int);
process Sieve[1] { // coordinator
   int p = 2;
   for (i = 3 to n by 2)
          send sieve[2](i);  // send odd numbers to Sieve[2]
   while (p != EOF) {
    println(p);
    receive sieve[1](p);
  }
}
process Sieve[i = 2 to L] {
 int p, next;
 bool sent = false;
 receive sieve[i-1](p);  // receive prime p
 send sieve[1](p);       // send to print out
 while (true) {
   receive sieve[i-1](next);  // receive next candidate
   if (next mod p) != 0 {     // if it might be prime,
     send sieve[i+1](next);   // pass it on
     sent = true;
   }
   if (!sent) { send sieve[1](EOF); break; }
  }
}
```

4. Probe/Echo Algorithms

- Used to disseminate and/or to gather information
  - **Probes** – to disseminate request or information
  - **Echoes** – to collect information or to acknowledge
  - Each probe should be echoed
- For example:
  - Broadcast a message
    - Using a spanning tree – when knows a global topology
    - Using neighbor sets – when knows neighbors
  - The network topology problem: collect all local topologies and build their union , i.e. the entire network topology
  - Web "crawlers“

Example: The Network Topology Problem

- Assume each node knows its local topology – neighbors
- Problem: collect all local topologies and build their union – **a network topology**
- Two phases:
  1. Each node sends a probe to its neighbors.
  2. Later each node sends an echo with local topology back to the node from which it received the first probe.
  - Eventually the initiating node **S** collects all echoes and computes a global topology.
  - **S** may distribute the global topology to other processes.

Probe/echo Algorithm for Gathering the Topology of a Tree.

```
type graph = bool [n,n];
chan probe[n](int sender);
chan echo[n](graph topology)     # parts of the topology
chan finalecho(graph topology)   # final topology

process Node[p = 0 to n-1] {
  bool links[n] = neighbors of node p;
  graph newtop, localtop = ([n*n] false);
  int parent;   # node from whom probe is received
  localtop[p,0:n-1] = links;   # initially my links

  receive probe[p](parent);
  # send probe to other neighbors, who are p's children
  for [q = 0 to n-1 st (links[q] and q != parent)]
    send probe[q](p);

  # receive echoes and union them into localtop
  for [q = 0 to n-1 st (links[q] and q != parent)] {
    receive echo[p](newtop);
    localtop = localtop or newtop;  # logical or
  }
  if (p == S)
    send finalecho(localtop);    # node S is root
  else
    send echo[parent](localtop);
}

process Initiator {
  graph topology;
  send probe[S](S)    # start probe at local node
  receive finalecho(topology);
}
```

ROYAL INSTITUTE OF TECHNOLOGY

5. Broadcast Algorithms

- For dissemination of information
- For making decentralized decisions (consensus)
  - Each process must participate in every decision
- For solving many distributed synchronization problems (e.g. distributed semaphores, distributed mutual exclusion)
- Use logical clocks to order communication events

Logical Clocks

- Many distributed algorithms use logical clocks to order communication events
- **A logical clock** (`lc`) is a private integer counter that a proc increases on every communication event.
- Proc attaches a **timestamp** (`ts`) to each message it sends:

```
ts = lc++; send(m, ts);
```

- Proc checks and corrects its `lc` when it receives a message with `ts`:

```
receive(m, ts); lc = max(lc, ts+1); lc++;
```

ROYAL INSTITUTE OF TECHNOLOGY

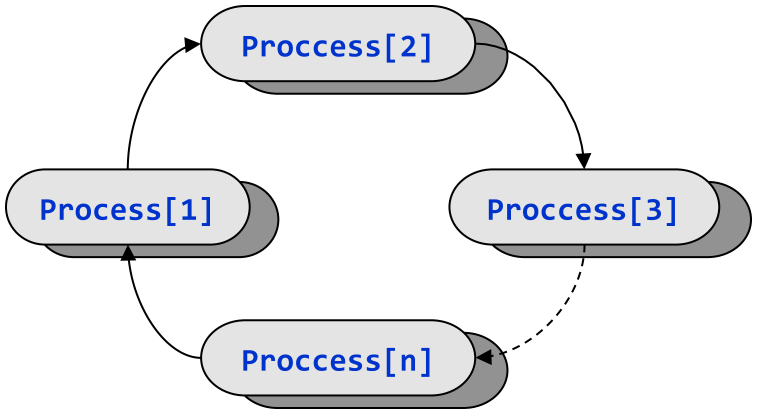
Making Decisions (Achieving Consensus)

- Broadcast a message `m` with `ts` (and process id)
- Wait until have a **later message** from **every other** process
- Now know you won't ever see an earlier message, i.e. the message `m` is **fully acknowledged**.
- Hence if your broadcast is the oldest, you can act upon it:
  - Make decision based on `m` and messages in front of `m` in the message queue ordered by `ts` – **stable prefix** – all messages are fully acknowledged since they have smaller timestamps
  - Use logical clocks for ordering
  - Use something like process ids to break ties

ROYAL INSTITUTE OF TECHNOLOGY

6. Token Passing Algorithms

- Used to convey some permission (e.g. distributed mutual exclusion) or to detect termination
- Example: Mutual exclusion with a token ring
  - A lock is a token that is passed in the ring
  - A proc that needs the locks grabs it when the lock passes the proc.



Mutual Exclusion with a Token Ring

- There are **User** and **Helper** processes in the ring: one **Helper** per **User**.
- A token passed in the ring is a mutex lock.
- A **Helper** gets the token and checks if its **User** needs the token to enter its critical section.
- Invariant:

DMUTEX: `User[i]` is in its CS ⇒ `Helper[i]` has the token ∧ there is exactly one token

```
chan token[1:n](), enter[1:n](), go[1:n](), exit[1:n]();

process Helper[i = 1 to n] {
  while (true) {   # loop invariant DMUTEX
    receive token[i]();              # wait for token
    if (not empty(enter[i])) {       # does user want in?
      receive enter[i]();              # accept enter msg
      send go[i]();                    # give permission
      receive exit[i]();               # wait for exit
    }
    send token[i%n + 1]();           # pass token on
  }
}

process User[i = 1 to n] {
  while (true) {
    send enter[i]();               # entry protocol
    receive go[i]();
    critical section;
    send exit[i]();                # exit protocol
    non-critical section;
  }
}
```

Summary of Paradigms for Process Interaction

- Manager/Worker
  - A distributed bag of tasks or work farm model used in many parallel applications
- Heartbeat Algorithms
  - Exchange on each iteration – Used in many iterative applications with a fixed number of tasks and a fixed number of parallel processes –
- Pipeline Algorithms
  - Divide work evenly, compute and circulate data among workers
- Probe/Echo Algorithms
  - Used to disseminate and/or gather information in a distributed application
- Broadcast Algorithms
  - Used to disseminate information or to make decentralized decisions (consensus)
- Token-Passing Algorithms
  - Used to convey permission or to detect termination in a distr. application

Parallelism in Scientific Computing

Outline

- Demand for parallel computing.
- Parallel scientific computing
  - Speedup, overhead
  - Steps in developing parallel programs
- Techniques in scientific computing
  - Grid computations – modeling of continuous processes and systems
    - Example: The heat problem (the Laplace equation)
  - Particle computations – modeling of discrete systems
    - Example: The N-body problem
  - Matrix computations – algebraic equations, image and signal processing, etc.

Demand for Parallel Computing

- Why parallel computing?
- Application demands for performance
  - Our insatiable need for computing cycles in many areas
    - Scientific computing,
    - General-purpose computing: Engineering, commercial, financial, entertainment, etc.
- Ways to increase performance:
  - Faster clock – limited?
  - **Parallelism** – alternative to faster clock
    - Allows improving both, execution performance and utilization of CPUs

Inevitability of Parallel Computing

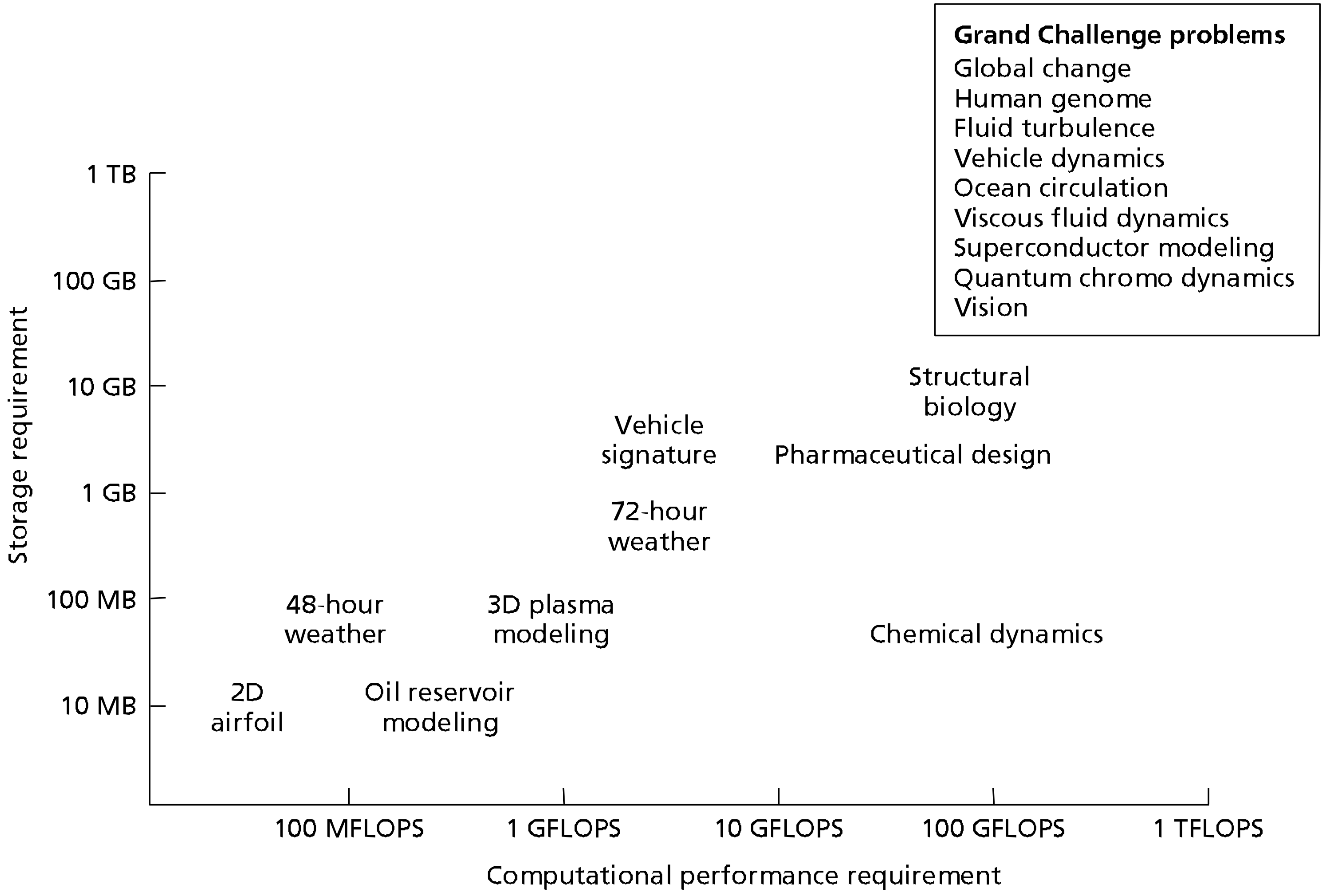
- Application demands – major driving force
- Technology trends
  - Number of transistors on chip growing rapidly
  - Clock rates growing slowly; furthermore it declines
- Computer architecture trends
  - Instruction-level parallelism valuable but limited
  - Coarser-level parallelism, as in MPs and multi-core processors, the most viable approach
- Economics
  - Multiprocessors being pushed by software vendors (e.g. database) as well as hardware vendors (a line of products from PC to high-end servers)
- Current trends:
  - Today's microprocessors are multicore
  - Servers and workstations are multicore-based: HP, Dell, …
  - Today's and tomorrow's microprocessors are multiprocessors.

ROYAL INSTITUTE OF TECHNOLOGY

Scientific Computing

- Scientific computing is a way to examine physical phenomena by **computational modeling**
  - Weather forecast,
  - Modeling of ocean basin,
  - CFD (Computational Fluid Dynamics),
  - Modeling of nuclear reactions,
  - Modeling the evolution of galaxies, etc.
- **High Performance Computing (HPC)**
  - Targets to improving performance by any means – the traditional demand of scientific computing

Scientific Computing Demands. Grand Challenge problems (90s)



Parallel Scientific Computing

Goal: speedup on large problems (or solve an even larger problem)

- **Speedup**: $T_{\text{sequential execution}} / T_{\text{parallel execution on } n \text{ processors}}$.
- Assume N is a problem size; n is a number of processors
- **Amdahl's law** (for a fixed N): If fraction s of sequential execution is inherently serial, then speedup approaches $1/s$ as n approaches infinity

$$\text{speedup}(n) = 1/(s + (1 - s) / n) \leq 1/s$$

- **Gustafson's Law** (for a growing N): If the sequential fraction $s(N)$ diminishes with problem size N, then speedup approaches n as N approaches infinity

$$\text{speedup}(N, n) = n - s(N)\ (n - 1) \leq n$$

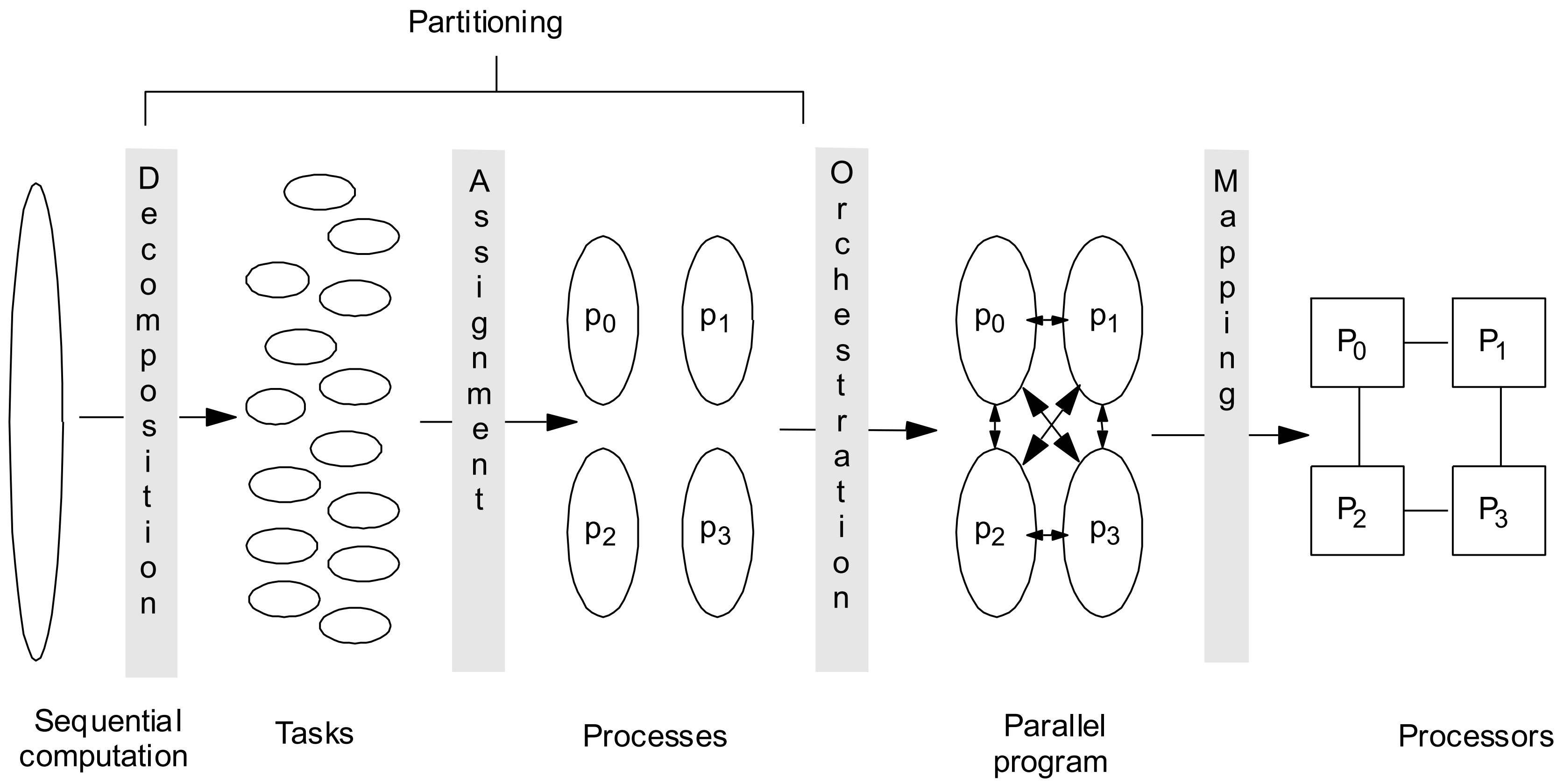
Parallel Scientific Computing (cont'd)

- Parallelization:
  - Typically – **domain decomposition** ("divide and conquer")
  - Simple way to identify parallelism is to look at loops (dependence analysis)
  - If not enough concurrency (e.g. loops are sequential), then look further and examine fundamental dependences, ignoring loop structure
- The challenge for a parallel program is to minimize overheads:
  - Create processes once ("work farm", "thread pool");
  - Achieve good load balancing;
  - Minimize the need for synchronization and use efficient algorithms for critical sections and barriers.

KTH VETENSKAP OCH KONST
ROYAL INSTITUTE OF TECHNOLOGY

Recall: Four Steps in Creating a Parallel Program (see Lecture 1)

(1) Decomposition; (2) Assignment; (3) Orchestration; (3) Mapping



Parallel Simulation of a Phenomenon

- Parallelization based on the idea of **domain decomposition**: divide data into partitions (strips, stripes, blocks); assign a worker to each (or use a "bag of tasks")
- A typical parallel simulation algorithm is a heartbeat algorithm:

– with shared memory:

```
start with a model;
// step through time
for [ t  =  start to finish ] {
   Compute();
   BARRIER();
   Update();
   BARRIER();
}
```

– with message passing:

```
start with a model;
// step through time
for [ t  =  start to finish ] {
      Compute();
      EXCHANGE();
      Update();
      EXCHANGE();
}
```

Fundamental Techniques in Scientific Computing

- **Computations on a grid of points**
  - To solve partial differential equations (PDE) – to approximate the solution on a finite number of points using iterative numerical methods
  - PDEs are used to model continuous systems and processes (e.g. airflow over a wing)
- **Particle computations**
  - To model (discrete) systems of particles/bodies that exert influence on each other (e.g. stars)
- **Matrix computations**
  - Linear equations
  - Arise in many application domains like optimization problems, e.g., modeling the stock market or the economy, image processing, etc.

1. Computations on Grid of Points

- A continuous-time continuous-space system is modeled as a 3D (2D) grid of points in discrete time
  - The finer spatial and temporal steps are the greater accuracy can be achieved.
  - Many different computations per time step.
  - Applications: weather, fluid (air) flow, plasma physics, etc.
- PDE solvers
- Parallelization – the idea of **domain decomposition** (iterative data parallelism)
  - divide area into blocks or strips of points; assign a worker to each partition; iterations over time steps.

Example: Laplace's Equation

- The Laplace PDE with Dirichlet boundary conditions (2D):

$$\partial^2\Phi/ \partial^2 x + \partial^2 \Phi/ \partial^2 y = 0$$

  - A.k.a., stationary heat equation
  - Here Φ is unknown potential such as stress or temperature (heat)
  - Values on boundaries are constant
  - Determine values in the interior area in the steady state (a spacial thermal gradient)
  - In practice: changing boundaries, multiple attributes (e.g., think about weather modeling)
- Solution:
  - Represent the region as an evenly spaced grid of *n* by *n* points and iterate using a finite-difference method until the difference between new values and previous values are within some tolerance ε

ROYAL INSTITUTE OF TECHNOLOGY

Iterative Techniques (From Slow to Fast)

- A Jacobi iteration (two arrays):

  newG[i, j] = (oldG[i, j-1] + oldG[i-1, j] + oldG[i+1, j] + oldG[i, j+1]) / 4

- A Gauss-Seidel (GS) iteration:

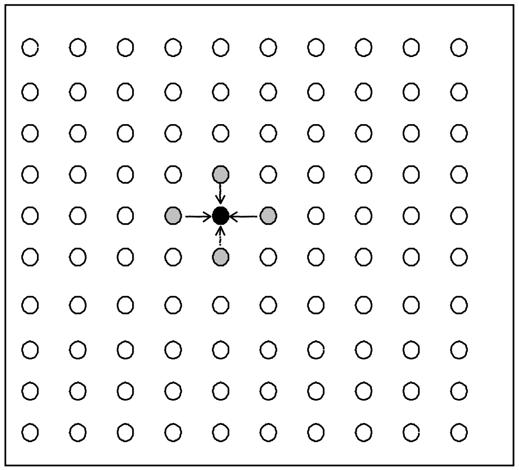
  G[i, j] = 0,25 × (G[i, j-1] + G[i-1, j] + G[i+1, j] + G[i, j+1])

- A successive over-relaxation (SOR) iteration:

  G[i, j] = ω × 0,25 × (G[i, j-1] + G[i-1, j] + G[i+1, j] + G[i, j+1]) + (1 - ω) × G[i, i]

  – here **0 < ω < 2**

- Multigrid
  – changing granularity of the grid



Decomposition of the Dirichlet Problem

- Decomposition in general:
  - Simple way to identify concurrency is to look at loop iterations
    - dependence analysis; if not enough concurrency, then look further
  - If not much concurrency here at this level (most loops are sequential), examine fundamental dependences, ignoring loop structure
- For the Dirichlet problem we use the idea of **domain decomposition (iterative data parallelism)**
  - divide area into blocks or strips of points; assign a worker to each data partition, exchanges values on partitions' boundaries on each iteration (on each time step)

ROYAL INSTITUTE OF TECHNOLOGY

Assignment

- **Static assignments**: ***n/p*** rows are assigned to a process
  - Assignment by strips reduces communication by keeping adjacent rows together
  - Reduces concurrency from ***n*** to ***p***
- **Dynamic assignment** using the bag-of-tasks technique:
  - get a row index, process the row, get a new row, and so on
- We use static assignment here: assignment of strips
  - Use the number of workers and the problem size to compute a strip size;
  - With shared variables: static assignment controlled by values of variables used as loop bounds
    - Use the strip size and a worker number to determine the bounds.

Jacobi Iteration Using Shared Variables

- For $n = 100$, $p = 4$, each worker is assigned a strip of 25 rows: worker[1] – rows 1-25, worker[2] – rows 26-50, etc.
- Each worker performs a number of interactions defined by **`MAXITERS`**.
- To avoid swapping of arrays **`grid`** and **`newgrid`**, the loop is unrolled twice – two consecutive iterations in the loop body.
- Barrier after each iteration .

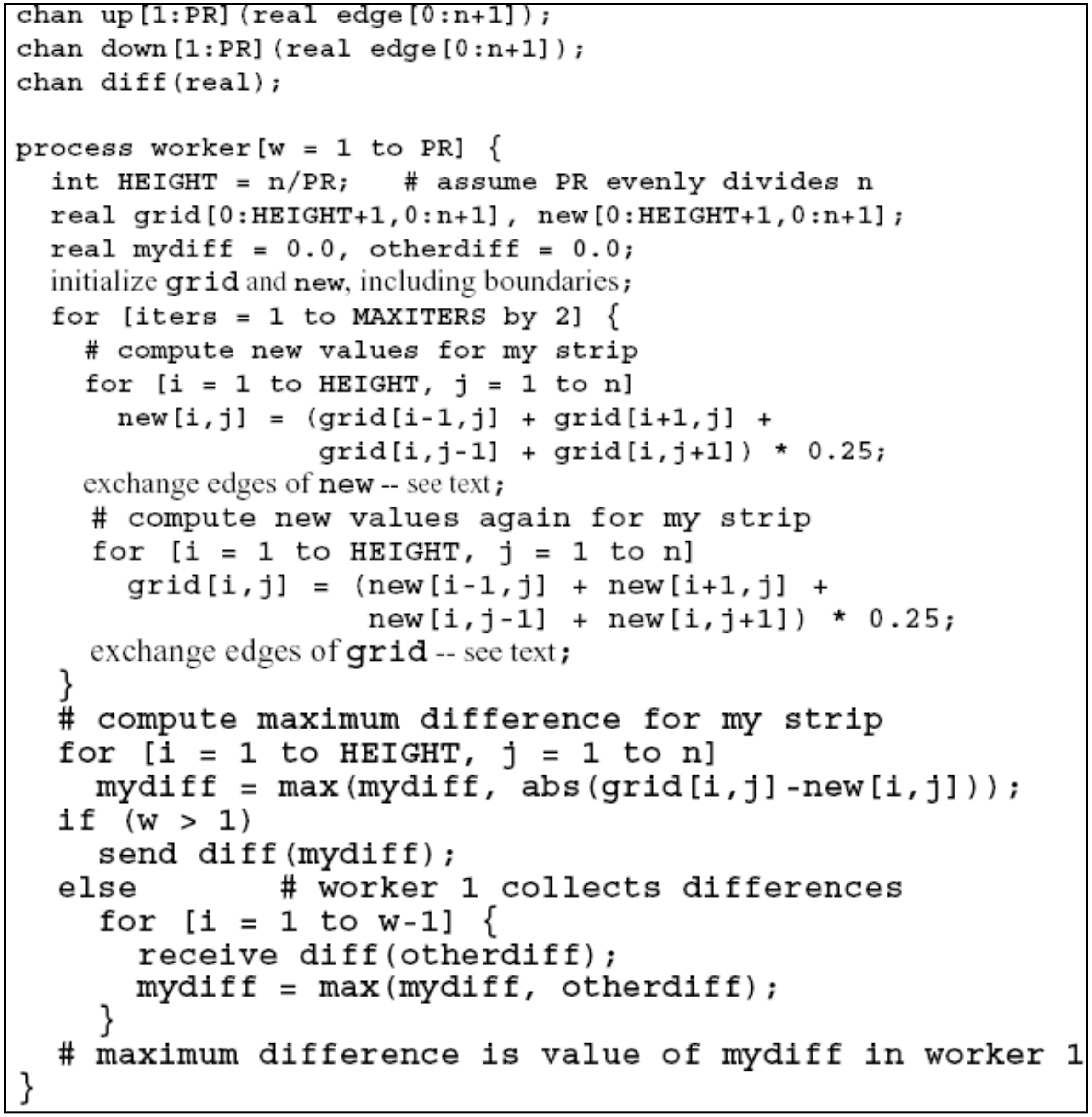
```
real grid[0:n+1,0:n+1], new[0:n+1,0:n+1];
int HEIGHT = n/PR;    # assume PR evenly divides n
real maxdiff[1:PR] = ([PR] 0.0);

procedure barrier(int id) {
  # efficient barrier algorithm from Section 3.4
}

process worker[w = 1 to PR] {
  int firstRow = (w-1)*HEIGHT + 1;
  int lastRow = firstRow + HEIGHT - 1;
  real mydiff = 0.0;
  initialize my strips of grid and new, including boundaries;
  barrier(w);
  for [iters = 1 to MAXITERS by 2] {
    # compute new values for my strip
    for [i = firstRow to lastRow, j = 1 to n]
      new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                  grid[i,j-1] + grid[i,j+1]) * 0.25;
    barrier(w);
    # compute new values again for my strip
    for [i = firstRow to lastRow, j = 1 to n]
      grid[i,j] = (new[i-1,j] + new[i+1,j] +
                   new[i,j-1] + new[i,j+1]) * 0.25;
    barrier(w);
  }
  # compute maximum difference for my strip
  for [i = firstRow to lastRow, j = 1 to n]
    mydiff = max(mydiff, abs(grid[i,j]-new[i,j]));
  maxdiff[w] = mydiff;
  barrier(w);
  # maximum difference is the max of the maxdiff[*]
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Jacobi Iteration Using Message Passing

- Each worker is assigned a partition – a strip of rows.
- Heartbeat algorithm: on each iteration, each worker exchanges edges with its neighbors (if any):

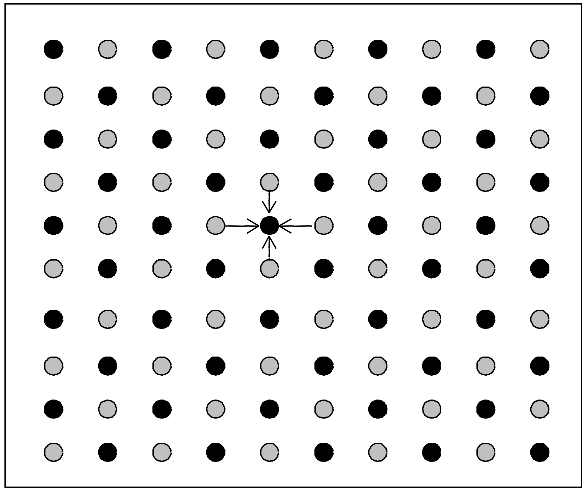
```
if (w > 1)
    send up[w-1](new[1,*]);
if (w < PR)
    send down[w+1](new[HEIGHT,*]);
if (w > PR)
    recv up[w](new[HEIGHT+1,*]);
if (w < 1)
    recv down[w](new[0,*]);
```



```
chan up[1:PR](real edge[0:n+1]);
chan down[1:PR](real edge[0:n+1]);
chan diff(real);

process worker[w = 1 to PR] {
  int HEIGHT = n/PR;    # assume PR evenly divides n
  real grid[0:HEIGHT+1,0:n+1], new[0:HEIGHT+1,0:n+1];
  real mydiff = 0.0, otherdiff = 0.0;
  initialize grid and new, including boundaries;
  for [iters = 1 to MAXITERS by 2] {
    # compute new values for my strip
    for [i = 1 to HEIGHT, j = 1 to n]
      new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                  grid[i,j-1] + grid[i,j+1]) * 0.25;
    exchange edges of new -- see text;
    # compute new values again for my strip
    for [i = 1 to HEIGHT, j = 1 to n]
      grid[i,j] = (new[i-1,j] + new[i+1,j] +
                   new[i,j-1] + new[i,j+1]) * 0.25;
    exchange edges of grid -- see text;
  }
  # compute maximum difference for my strip
  for [i = 1 to HEIGHT, j = 1 to n]
    mydiff = max(mydiff, abs(grid[i,j]-new[i,j]));
  if (w > 1)
    send diff(mydiff);
  else         # worker 1 collects differences
    for [i = 1 to w-1] {
      receive diff(otherdiff);
      mydiff = max(mydiff, otherdiff);
    }
  # maximum difference is value of mydiff in worker 1
}
```

Red/Black Successive Over-Relaxation (SOR) Iteration

- To decompose, exploit application knowledge: reorder grid traversal: red-black ordering (R depends on B and visa versa)
  - Different ordering of updates: may converge quicker or slower
  - **Red sweep and black sweep are each fully parallel:**
    - Global synch between them (conservative but convenient)

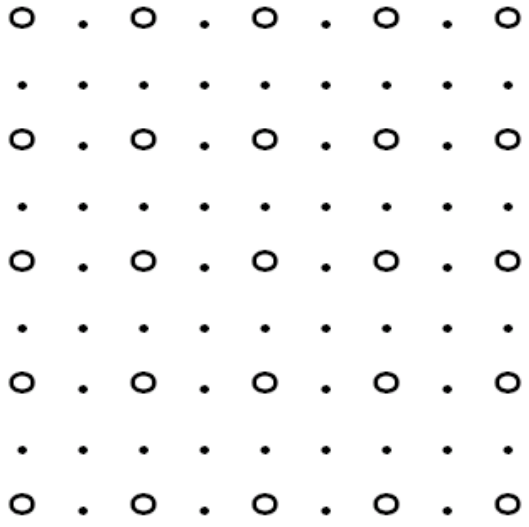


○ Red point
● Black point

See: MPT book Figure 11.6 Red/black Gauss-Seidel using shared variables.

Multigrid Methods

- To solve a large problem rapidly
- Idea: To use grids of different granularities and to switch between them to increase the rate of convergence of the finest grid



.	*Fine grid point*
o	*Coarse and fine grid point*

KTH VETENSKAP OCH KONST

ROYAL INSTITUTE OF TECHNOLOGY

Coarse Grid Correction

- Start with a fine grid, update points for a few iterations using any relaxation method (J, GS, SOR)
- Restrict the result to coarse-grained (twice coarser) by a **restriction operator**

$$\begin{bmatrix} 0 & 1/8 & 0 \\ 1/8 & 1/2 & 1/8 \\ 0 & 1/8 & 0 \end{bmatrix}$$

```
coarse[x,y] = fine[i,j]*0.5 +
  (fine[i-1,j]+fine[i,j-1]+fine[i+1,j]+fine[i,j+1])*0.125
```

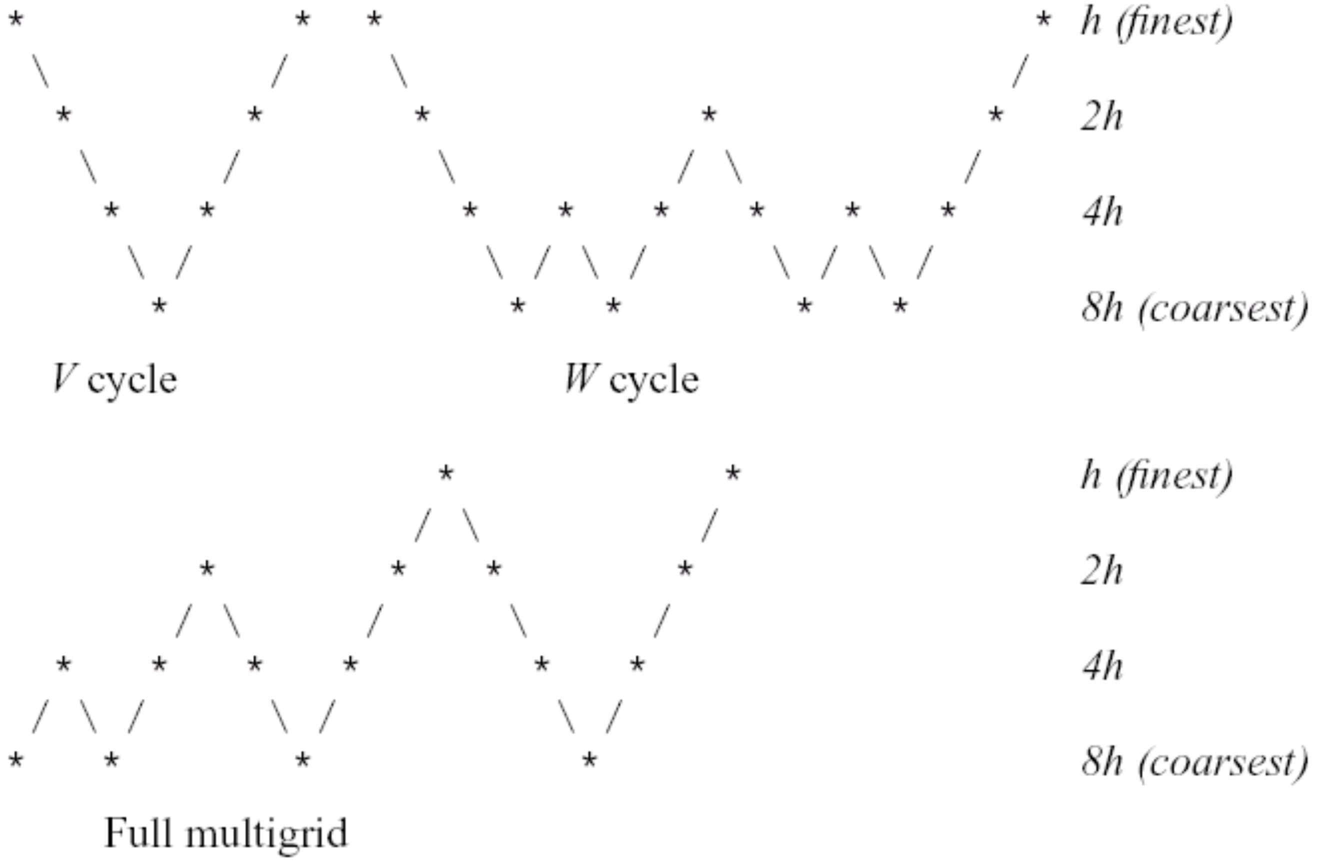
- Compute the coarse grid.
- Interpolate the coarse grid back to the fine grid by using an **interpolation operator**

$$\begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{bmatrix}$$

- Update points on the fine grid for a few iterations.
- Change again to the coarse grid, etc.

ROYAL INSTITUTE OF TECHNOLOGY

Patterns for Multigrid



2. Particle Computations

- To model a discrete system consisting of interacting particles/ bodies
  - ***n*** particles requires $\boldsymbol{O(n^2)}$ calculations on a time step
  - By using approximations, can be improved to $\boldsymbol{O(n \log_2 n)}$
- Typically irregular iterative computations over time steps.
- Examples of applications:
  - Particle interactions due to chemical bonding,
  - Gravity (evolution of galaxies)
  - Electrical charge, fluid flow, etc.

Example: Gravitational N-Body Problem

- *N* bodies. Each body:
  - Position `p (x, y, z)`
  - Velocity `v (x, y, z)`
  - Mass `m` is constant
  - Force `F (x, y, z)`
- Gravity causes the bodies to accelerate and to move.
- The motion of the bodies is simulated by stepping through discrete instants of time:

```
initialize bodies;
for time step {
  calculate forces;  // read p, m; compute F
  move bodies;       // read F, m; compute v and new p
}
```

Calculation Forces. Moving Bodies

- **Force** on a body is the vector sum of the forces from all other bodies
  - Magnitude of the gravitational force between bodies i and j:

    `F[i,j] = (G * m[i] * m[j]) / r[i,j]**2`
  - magnitude: symmetric ("equal and opposite")
  - direction: vector from one body to the other
- Changes in velocity and position (**moving a body**) – leapfrog scheme:
  - Acceleration: `a = F / m`
  - Change in velocity: `dv = a * DT`
  - Change in position:

    `dp = v * DT + (a/2)*DT**2 ≈ (v + dv/2) * DT`
    - Here `DT` – the length of a time step

ROYAL INSTITUTE OF TECHNOLOGY

Parallelization of the N-Body Problem

- Assume, shared memory programming model
- Parallelization – divide bodies among workers, use a barrier after each phase:

```
initialize bodies;
for time step {
    calculate forces;
    BARRIER;
    move bodies;
    BARRIER;
}
```

Assignment

- Assume P worker processes.
- Three ways to assign work (distribute bodies among workers):
  - **Blocks** (assign each worker a continues block of bodies):
    - w[1] is assigned the first n/P bodies,
    - w[2] is assigned the next n/P, etc.
  - **Stripes** (cyclic allocation of bodies to workers):
    - body 1 to w[1], body 2 to w[2], …, body P to w[P],
      body P+1 to w[1], body P+2 to w[2], etc.
  - **Reverse stripes** (reverse cyclic allocation):
    - body 1 to w[1], body 2 to w[2], …, body P to w[P],
      then reverse stripes:
      body P+1 to w[P], body p+2 to w[P-1], etc.

Assignment of Bodies to Workers

- Assume, 2 workers (B – "Black" and W – "White") and 8 bodies
- For each body *i* assigned to a worker, the worker computes forces between the body *i* and bodies *i* + 1, …, *n*

Pattern	1	2	3	4	5	6	7	8	*Workload*
blocks	B	B	B	B	W	W	W	W	B = 22, W = 6
stripes	B	W	B	W	B	W	B	W	B = 16, W = 12
reverse stripes	B	W	W	B	B	W	W	B	B = 14, W = 14

- The pattern of stripes leads to a fairly well-balanced workload that is easy to program.

Synchronization

- Barriers after each stage
- Critical sections in calculate forces phase:
  - Access to the same body by different workers (writers) must be synchronized (mutual exclusion)
  - Approaches:
    - One global lock – very inefficient
    - A lock per a body – too much synchronization
    - Eliminate critical sections at all
- To eliminate critical sections, replicate and split shared data:
  - Change the force vector into a force matrix: a private force vector per worker.
  - The result force vector: sums in columns – can be calculated in parallel in the moving phase

ROYAL INSTITUTE OF TECHNOLOGY

N-Body Parallel Program with Shared Variables

- Procedures:
  - Calculate forces
  - Move bodies
  - Barrier
- Workers:

```
process Worker[w = 1 to PR] {
  # run the simulation with time steps of DT
  for [time = start to finish by DT] {
    calculateForces(w);
    barrier(w);
    moveBodies(w);
    barrier(w);
} }
```

```
type point = rec(double x, y); double G = 6.67e-11;
point p[1:n], v[1:n], f[1:PR,1:n];  # position, velocity,
double m[1:n];                       # force and mass for each body
initialize the positions, velocities, forces, and masses;

procedure barrier(int w) { # efficient barrier from Section 3.4 }

# calculate forces for bodies assigned to worker w
procedure calculateForces(int w) {
  double distance, magnitude; point direction;
  for [i = w to n by PR, j = i+1 to n] {
    distance = sqrt( (p[i].x - p[j].x)**2 +
                     (p[i].y - p[j].y)**2 );
    magnitude = (G*m[i]*m[j]) / distance**2;
    direction = point(p[j].x-p[i].x, p[j].y-p[i].y);
    f[w,i].x = f[w,i].x + magnitude*direction.x/distance;
    f[w,j].x = f[w,j].x - magnitude*direction.x/distance;
    f[w,i].y = f[w,i].y + magnitude*direction.y/distance;
    f[w,j].y = f[w,j].y - magnitude*direction.y/distance;
} }

# move the bodies assigned to worker w
procedure moveBodies(int w) {
  point deltav;   # dv = f/m * DT
  point deltap;   # dp = (v + dv/2) * DT
  point force = (0.0, 0.0);
  for [i = w to n by PR] {
    # sum the forces on body i and reset f[*,i]
    for [k = 1 to PR] {
      force.x += f[k,i].x; f[k,i].x = 0.0;
      force.y += f[k,i].y; f[k,i].y = 0.0;
    }
    deltav = point(force.x/m[i] * DT, force.y/m[i] * DT);
    deltap = point( (v[i].x + deltav.x/2) * DT,
                    (v[i].y + deltav.y/2) * DT);
    v[i].x = v[i].x + deltav.x;
    v[i].y = v[i].y + deltav.y;
    p[i].x = p[i].x + deltap.x;
    p[i].y = p[i].y + deltap.y;
    force.x = force.y = 0.0;
} }
```

Approximate Methods for the N-Body Problem

- Approximation (based on Newtonian mechanics):
  - If two bodies are far apart, the force between them is negligible
  - The force between a distant body and a group can be approximated by the force between the former and a single body that approximates the group (has the total mass of all the bodies in the group and is located in the center of mass of the group).
- Two methods that exploit above approximations:
  - Barnes-Hut algorithm
  - Fast Multipole Method
  - Both allows $O(n \log n)$ force calculation per time step
  - See section 11.2.4 in the MPD text.

Summary of Parallelization of the N-Body Problem

- Decomposition: distribute bodies among workers.
- ***O*(*n*^{2})** force calculations per time step
- Minimizing overheads:
  - create workers once
  - avoid critical sections: compute forces into "private" arrays; add them when ready to move bodies:
    - calculate phase: read positions and masses; compute forces
    - move phase: read forces and masses; compute new v and p
- Load balancing: assignment of stripes to workers
- Faster algorithms, ***O*(*n* log *n*)** use approximations

3. Matrix Computations

- Linear equations
- Arise in many application domains
  - like optimization problems
  - e.g., modeling the stock market or the economy
  - image processing, etc.