Scientific Computing

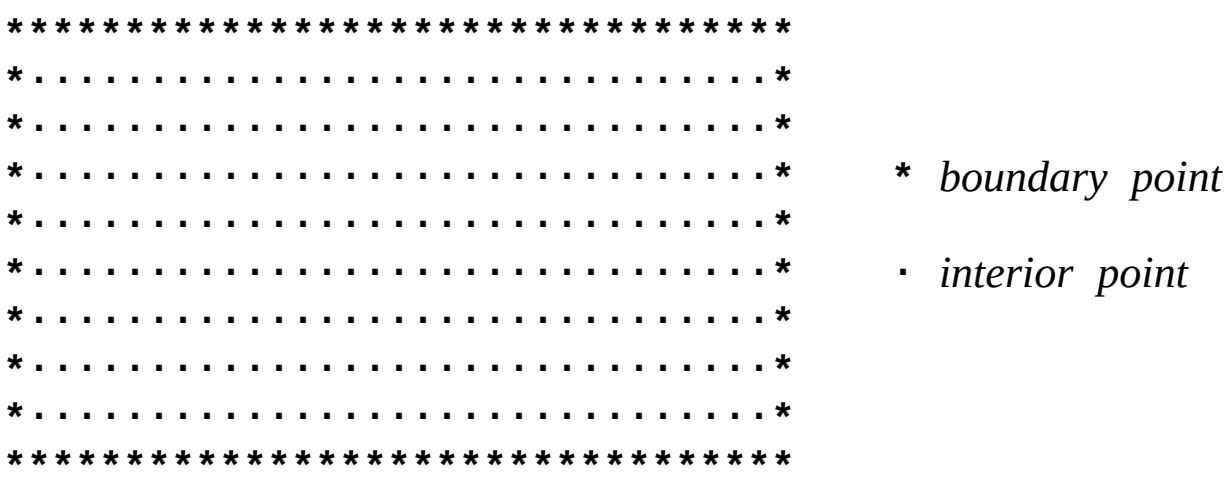

Figure 11.1 Approximating Laplace's equation using a grid.

```
real grid[0:n+1,0:n+1], new[0:n+1,0:n+1];
real maxdiff = 0.0;
initialize the grids, including the boundaries;
for [iters = 1 to MAXITERS by 2] {
  # compute new values for all interior points
  for [i = 1 to n, 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;
  # compute new values again for interior points
  for [i = 1 to n, 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;
}
# compute the maximum difference
for [i = 1 to n, j = 1 to n]
  maxdiff = max(maxdiff, abs(grid[i,j]-new[i,j]));
print the final grid and maximum difference;
```

Figure 11.2 Optimized sequential program for Jacobi 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[*]
}
```

Figure 11.3 Jacobi iteration using shared variables.

```
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
}
```

Figure 11.4 Jacobi iteration using message passing.

```
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;
    # send edges of new to neighbors
    if (w > 1)
      send up[w-1](new[1,*]);
    if (w < PR)
      send down[w+1](new[HEIGHT,*]);
    # compute new values for interior of my strip
    for [i = 2 to HEIGHT-1, 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;
    # receive edges of new from my neighbors
    if (w < PR)
      receive up[w](new[HEIGHT+1,*]);
    if (w > 1)
      receive down[w](new[0,*]);
    # compute new values for edges of my strip
    for [j = 1 to n]
      grid[1,j] = (new[0,j] + new[2,j] +
                   new[1,j-1] + new[1,j+1]) * 0.25;
    for [j = 1 to n]
      grid[HEIGHT,j] = (new[HEIGHT-1,j] +
                        new[HEIGHT+1,j] + new[HEIGHT,j-1] +
                        new[HEIGHT,j+1]) * 0.25;
  }
  compute maximum difference as in Figure 11.4;
}
```

Figure 11.5 Optimized Jacobi iteration using message passing.

```
real grid[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;
  int jStart;
  real mydiff = 0.0;
  initialize my strip of grid, including boundaries;
  barrier(w);
  for [iters = 1 to MAXITERS] {
    # compute new values for red points in my strip
    for [i = firstRow to lastRow] {
      if (i%2 == 1) jStart = 1;      # odd row
      else jStart = 2;               # even row
      for [j = jStart to n by 2]
        grid[i,j] = (grid[i-1,j] + grid[i,j-1] +
                     grid[i+1,j] + grid[i,j+1]) * 0.25;
    }
    barrier(w);
    # compute new values for black points in my strip
    for [i = firstRow to lastRow] {
      if (i%2 == 1) jStart = 2;      # odd row
      else jStart = 1;               # even row
      for [j = jStart to n by 2]
        grid[i,j] = (grid[i-1,j] + grid[i,j-1] +
                     grid[i+1,j] + grid[i,j+1]) * 0.25;
    }
    barrier(w);
  }
  # compute maximum difference for my strip
  perform one more set of updates, keeping track of the maximum
        difference between old and new values of grid[i,j];
  maxdiff[w] = mydiff;
  barrier(w);
  # maximum difference is the max of the maxdiff[*]
}
```

Figure 11.6 Red/black Gauss-Seidel using shared variables.

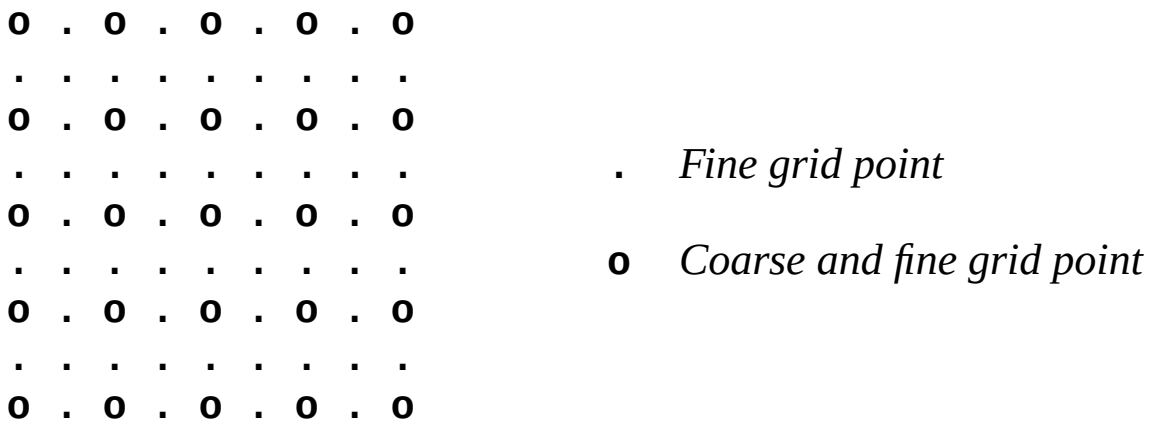

Figure 11.7 Fine and coarse grids, including boundary points.

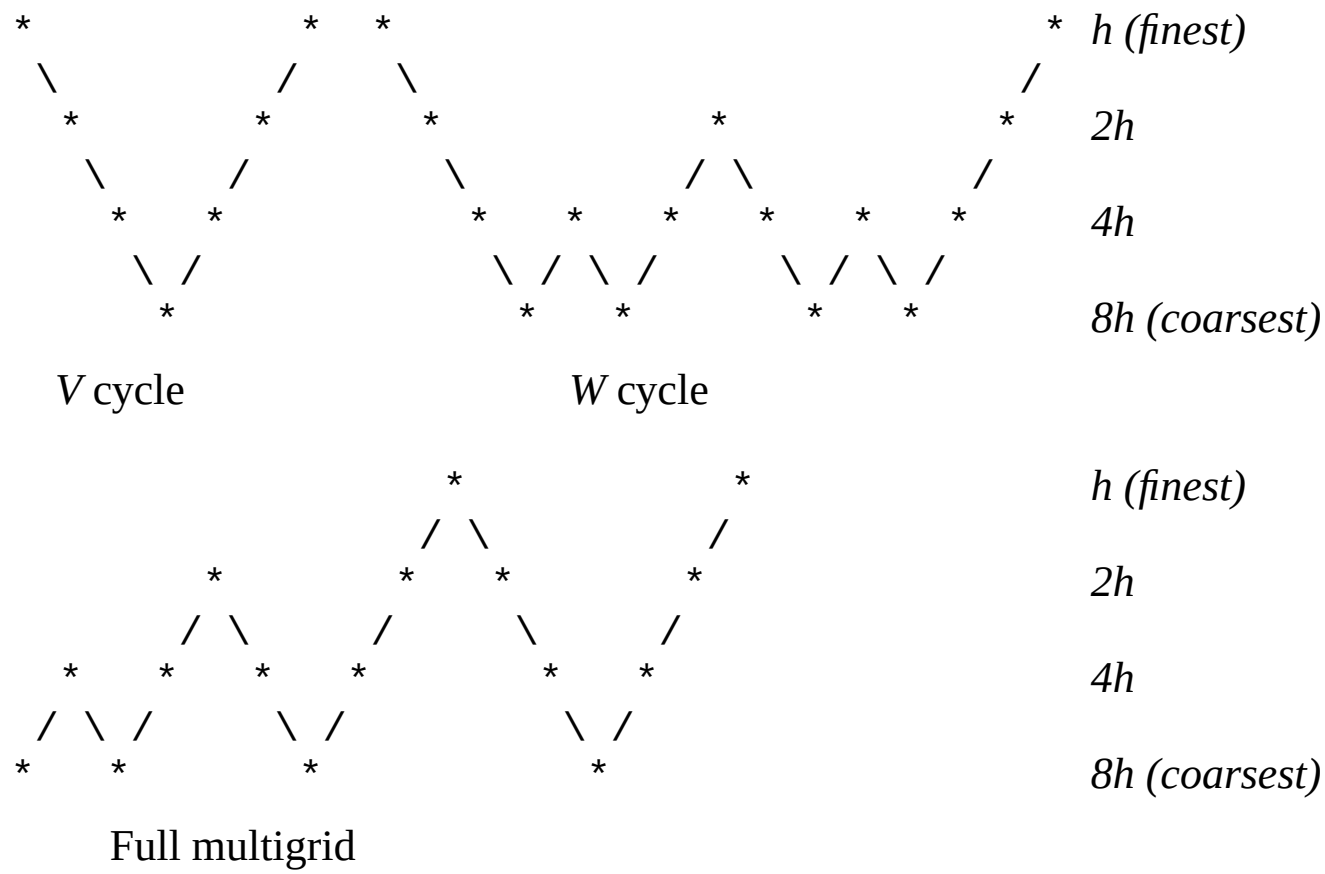

Figure 11.8 Patterns for multigrid methods.

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

```
# calculate total force for every pair of bodies
procedure calculateForces() {
  double distance, magnitude; point direction;
  for [i = 1 to n-1, 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[i].x = f[i].x + magnitude*direction.x/distance;
    f[j].x = f[j].x - magnitude*direction.x/distance;
    f[i].y = f[i].y + magnitude*direction.y/distance;
    f[j].y = f[j].y - magnitude*direction.y/distance;
  }
}

# calculate new velocity and position for each body
procedure moveBodies() {
  point deltav;   # dv = f/m * DT
  point deltap;   # dp = (v + dv/2) * DT
  for [i = 1 to n] {
    deltav = point(f[i].x/m[i] * DT, f[i].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;
    f[i].x = f[i].y = 0.0;    # reset force vector
  }
}

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

Figure 11.9 Sequential program for the *n*-body problem.

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

Figure 11.10 Patterns for assigning bodies to workers.

```
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;
} }

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);
} }
```

Figure 11.11 Shared variable program for the *n*-body problem.

```
chan getTask(int worker), task[1:PR](int block1, block2);
chan bodies[1:PR](int worker; point pos[*], vel[*]);
chan forces[1:PR](point force[*]);

process Manager {
  declare and initialize local variables;
  for [time = start to finish by DT] {
    initialize the bag of tasks;
    for [i = 1 to numTasks+PR ] {
      receive getTask(worker);
      select next task; use (0, 0) to signal bag is empty;
      send task[worker](block1, block2);
    }
  }
}

process Worker[w = 1 to PR] {
  point p[1:n], v[1:n], f[1:n];  # position, velocity
  double m[1:n];        # force and mass for each body
  declare other local variables; initialize all local variables;
  for [time = start to finish by DT] {
    while (true) {
      send getTask(w); receive task[w](block1, block2);
      if (block1 == 0) break;  # bag is empty
      calculate forces between bodies in block1 and block2;
    }
    for [i = 1 to PR st i != w]     # exchange forces
      send forces[i](f[*]);
    for [i = 1 to PR st i != w] {
      receive forces[w](tf[*]);
      add values in tf to those in f;
    }
    update p and v for my block of bodies;
    for [i = 1 to PR st i != w]     # exchanges bodies
      send bodies[i](w, p[*], v[*]);
    for [i = 1 to PR st i != w] {
      receive bodies[w](worker, tp[*], tv[*]);
      move bodies of worker from tp and tv to p and v;
    }
    reinitialize f to zeros;
  }
}
```

Figure 11.12 Manager/workers program for the *n*-body problem

```
chan bodies[1:PR](int worker; point pos[*], vel[*]);
chan forces[1:PR](point force[*]);

process Worker[w = 1 to PR] {
  int blockSize = size of my block of bodies;
  int tempSize = maximum number of other bodies in messages;
  point p[1:blockSize], v[1:blockSize], f[1:blockSize];
  point tp[1:tempSize], tv[1:tempSize], tf[1:tempSize];
  double m[1:n];
  declarations of other local variables;
  initialize all local variables;
  for [time = start to finish by DT] {
    # send my bodies to lower numbered workers
    for [i = 1 to w-1]
      send bodies[i](w, p[*], v[*]);
    calculate f for my block of bodies;
    # receive bodies from and send forces back to
    # higher numbered workers
    for [i = w+1 to PR] {  # get bodies from others
      receive bodies[w](other, tp[*], tv[*]);
      calculate forces between my block and other block;
      send forces[other](tf[*]);
    }
    # get forces from lower numbered workers
    for [i = 1 to w-1] {
      receive forces[w](tf[*]);
      add forces in tf to those in f;
    }
    update p and v for my bodies;
    re-initialize f to zeros;
  }
}
```

Figure 11.13 Heartbeat program for the *n*-body problem.

```
chan bodies[1:PR](int owner; point p[*], v[*], f[*]);

process Worker[w = 1 to PR] {
  int owner, setSize = n/PR, next = w%PR + 1;
  point p[1:setSize], v[1:setSize], f[1:setSize];
  point tp[1:setSize], tv[1:setSize], tf[1:setSize];
  double m[1:n];
  declarations of other local variables;
  initialize my block of bodies and other variables;
  for [time = start to finish by DT] {
    send bodies[next](w, p[*], v[*], f[*]);
    compute the forces among my block of bodies;
    for [i = 1 to PR-1] {
      receive bodies[w](owner, tp[*], tv[*], tf[*]);
      calculate the forces between my bodies and the new ones;
      send bodies[next](owner, tp[*], tv[*], tf[*]);
    }
    # get back my bodies (owner will equal w)
    receive bodies[w](owner, tp[*], tv[*], tf[*]);
    add forces in tf to those in f;
    update p and v for my set of bodies;
    re-initialize forces on my bodies to zeros;
  }
}
```

Figure 11.14 Pipeline program for the *n*-body problem.

Program/Phase	*Number of Messages*	*Bodies per Message*
Manager/workers		
Get task	`2 * (numTasks+PR)`	`2`
Exchange forces	`PR * (PR-1)`	`n`
Exchange bodies	`PR * (PR-1)`	`2 * n`
Heartbeat		
Exchange bodies	`PR * (PR-1) / 2`	`2 * n`
Exchange forces	`PR * (PR-1) / 2`	`n`
Pipeline		
Circulate bodies	`PR * PR`	`3 * n * PR`

Figure Table 11.1 Comparison of message-passing programs for *n*-body problem.

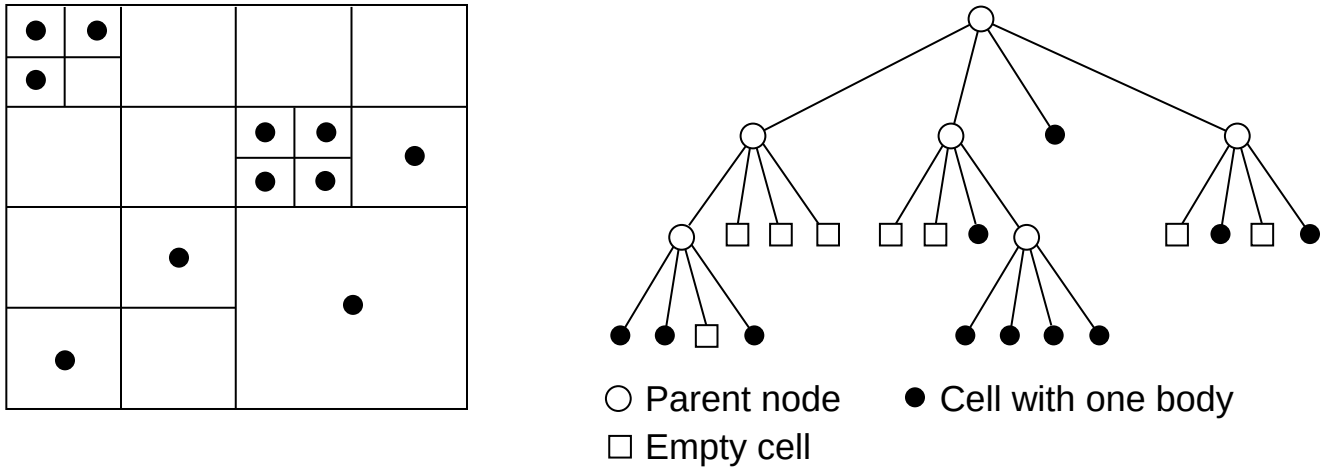

Figure 11.15 Cells and quadtree for a two-dimensional space.

```
double A[1:n,1:n], LU[1:n,1:n];  # assume A initialized
int ps[1:n];                     # pivot row indices
double pivot; int pivotRow;      # pivot value and row
double mult; int t;              # temporaries

# initialize ps and LU
for [i = 1 to n] {
  ps[i] = i;
  for [j = 1 to n]
    LU[i,j] = A[i,j];
}

# perform Gaussian elimination with partial pivoting
for [k = 1 to n-1] {     # iterate down main diagonal
  pivot = abs(LU[ps[k],k]); pivotRow = k;
  for [i = k+1 to n] {   # select pivot in column k
    if (abs(LU[ps[i],k]) > pivot) {
      pivot = abs(LU[ps[i],k]); pivotRow = i;
    }
  }
  if (pivotRow != k) {   # swap rows by swapping indices
    t = ps[k]; ps[k] = ps[pivotRow]; ps[pivotRow] = t;
  }
  pivot = LU[ps[k],k];      # get actual value of pivot
  for [i = k+1 to n] {      # for all rows in submatrix
    mult = LU[ps[i],k]/pivot;  # calculate multiplier
    LU[ps[i],k] = mult;        #   and save it
    for [j = k+1 to n]      # eliminate across columns
      LU[ps[i],j] = LU[ps[i],j] - mult*LU[ps[k],j];
  }
}
```

Figure 11.16 Sequential program for LU decomposition of a matrix.

```
double LU[1:n,1:n]; int ps[1:n];  # see Figure 11.16
double sum, x[1:n], b[1:n];

# forward substitution to solve L y = b, storing y in x
for [i = 1 to n] {
  sum = 0.0;
  for [j = 1 to i-1]
    sum = sum + LU[ps[i],j] * x[j];
  x[i] = b[ps[i]] - sum;
}

# backward substitution to solve U x = y for x
for [i = n to 1 by -1] {
  sum = 0.0;
  for [j = i+1 to n]
    sum = sum + LU[ps[i],j] * x[j];
  x[i] = (x[i] - sum) / LU[ps[i],i];
}
```

Figure 11.17 Solving **A x = b** given an LU decomposition of **A**.

```
double A[1:n,1:n], LU[1:n,1:n];  # assume A initialized
int ps[1:n];                     # pivot row indices

procedure barrier(int id) { ... }  # see Chapter 3

process Worker(w = 1 to PR) {
  double pivot, mult;
  declarations of other local variables, such as a copy of ps;
  for [i = w to n by PR]
    initialize ps and my stripes of LU;
  barrier(w);

  # perform Gaussian elimination with partial pivoting
  for [k = 1 to n-1] {     # iterate down main diagonal
    find maximum pivot element — see text;
    if necessary, swap pivot row and row k, then call barrier(w);
    pivot = LU[ps[k],k];     # get actual value of pivot
    for [i = k+1 to n st (i%PR == 0)] {  # for my stripe
      mult = LU[ps[i],k]/pivot;  # calculate multiplier
      LU[ps[i],k] = mult;        #   and save it
      for [j = k+1 to n]     # eliminate across columns
        LU[ps[i],j] = LU[ps[i],j] - mult*LU[ps[k],j];
    }
    barrier(w);
  }
}
```

Figure 11.18 Outline of shared variable program for LU decomposition.

```
declarations of channels;

process Worker(w = 1 to PR) {
  double LU[1:n/PR,1:n/PR];    # my rows of LU
  int ps[1:n/PR];              # pivot row indices
  double pivot, mult, pivotRow[n];
  int myRow;
  declarations of other local variables;
  initialize ps and my rows of LU;
  # perform Gaussian elimination with partial pivoting
  for [k = 1 to n-1] {     # iterate down main diagonal
    find maximum pivot element in column k of my rows;
    exchange pivot with other workers;
    select global maximum and update ps;
    if (owner of pivot row)
      broadcast pivotRow to other workers;
    else
      receive pivotRow;
    # eliminate my rows of LU using pivot and pivotRow
    for [i = k+1 to n st (i%PR == 0)] { # for my stripe
      myRow = i/PR;                 # convert row index
      mult = LU[ps[myRow],k]/pivot;  # compute multiplier
      LU[ps[myRow],k] = mult;        #   and save it
      for [j = k+1 to n]      # eliminate across columns
        LU[ps[myRow],j] = LU[ps[myRow],j] -
                          mult * pivotRow[j];
    }
  }
}
```

Figure 11.19 Outline of message-passing program for LU decomposition.