

Algorithms PART I: Embarrassingly Parallel

HPC Spring 2017

Prof. Robert van Engelen





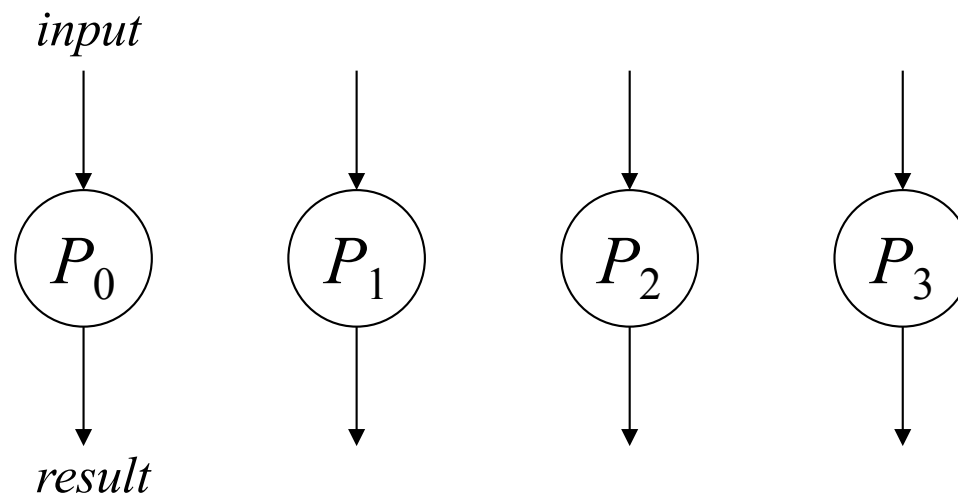
Overview

- Ideal parallelism
- Master-worker paradigm
- Processor farms
- Examples
 - Geometrical transformations of images
 - Mandelbrot set
 - Monte Carlo methods
- Load balancing of independent tasks
- Further reading



Ideal Parallelism

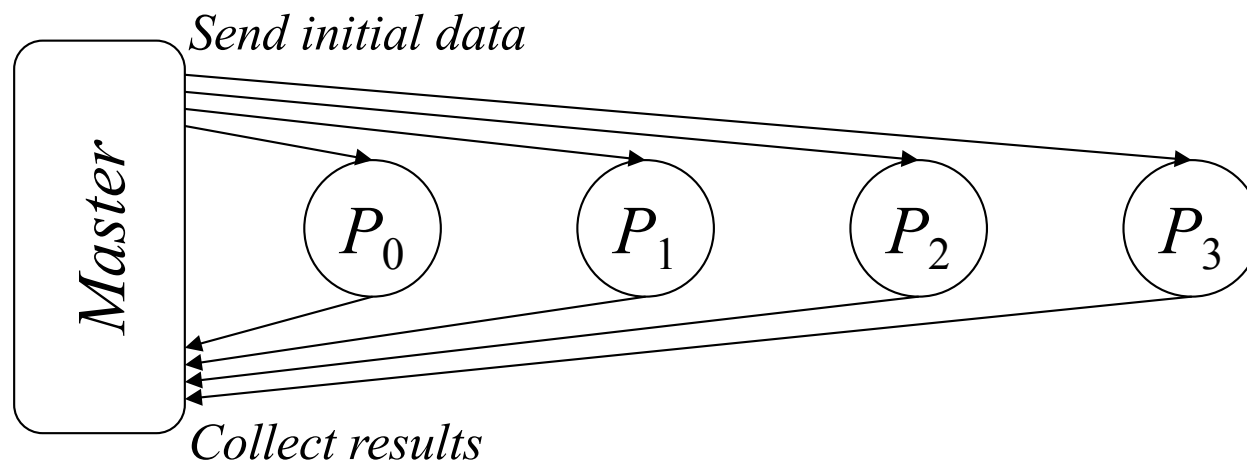
- An *ideal parallel computation* can be immediately divided into completely independent parts
 - “Embarrassingly parallel”
 - “Naturally parallel”
- No special techniques or algorithms required





Ideal Parallelism and the Master-Worker Paradigm

- Ideally there is no communication
 - Maximum speedup
- Practical embarrassingly parallel applications have initial communication and (sometimes) a final communication
 - Master-worker paradigm where master submits jobs to workers
 - No communications between workers





Parallel Tradeoffs

- Embarrassingly parallel with perfect load balancing:

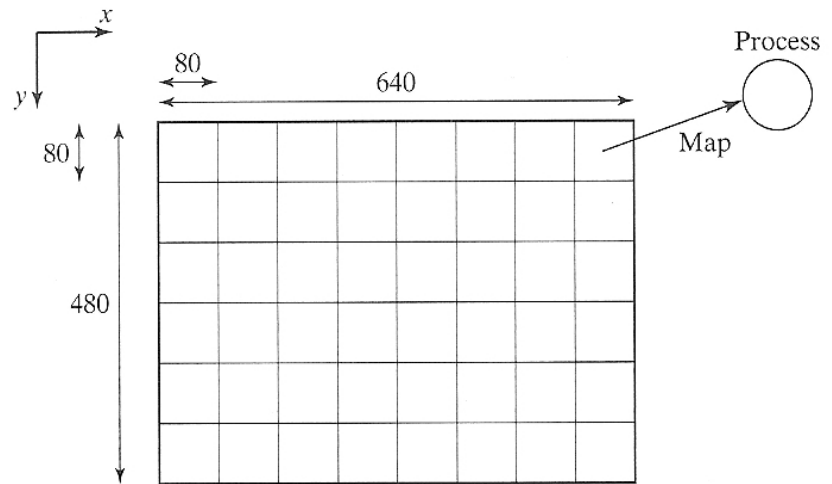
$$t_{comp} = t_s / P$$

assuming P workers and sequential execution time t_s

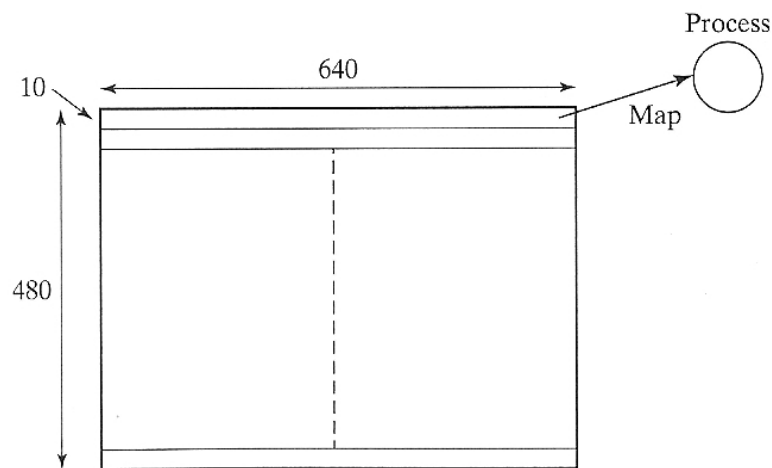
- Master-worker paradigm gives speedup only if workers have to perform a reasonable amount of work
 - Sequential time > total communication time + one workers' time
$$t_s > t_p = t_{comm} + t_{comp}$$
 - Speedup $S_P = t_s / t_p = P t_{comp} / (t_{comm} + t_{comp}) = P / (r^{-1} + 1)$ where $r = t_{comp} / t_{comm}$
 - Thus $S_P \rightarrow P$ when $r \rightarrow \infty$
- However, communication t_{comm} can be expensive
 - Typically $t_s < t_{comm}$ for small tasks, that is, the time to send/recv data to the workers is more expensive than doing all the work
 - Try to overlap computation with communication to hide t_{comm} latency



Example 1: Geometrical Transformations of Images



(a) Square region for each process



(b) Row region for each process

- Partition *pixmap* into regions
 - By block (row & col block)
 - By row
- Pixmap operations
 - Shift
$$x' = x + \Delta x$$
$$y' = y + \Delta y$$
 - Scale
$$x' = S_x x$$
$$y' = S_y y$$
 - Rotation
$$x' = x \cos \theta + y \sin \theta$$
$$y' = -x \sin \theta + y \cos \theta$$
 - Clip
$$x_l \leq x' = x \leq x_h$$
$$y_l \leq y' = y \leq y_h$$



Example 1: Master and Worker Naïve Implementation

```
row = 0;
for (p = 0; p < P; p++)
{ send(row, p);
  row += 480/P;
}
for (i = 0; i < 480; i++)
  for (j = 0; j < 640; j++)
    temp_map[i][j] = 0;
for (i = 0; i < 480 * 640; i++)
{ recv(&oldrow, &oldcol, &newrow, &newcol, anyP);
  if (!(newrow < 0 || newrow >= 480 || newcol < 0 || newcol >= 640))
    temp_map[newrow][newcol] = map[oldrow][oldcol];
}
for (i = 0; i < 480; i++)
  for (j = 0; j < 640; j++)
    map[i][j] = temp_map[i][j];
```

Master

Worker

Each worker computes:

$\forall \text{ row} \leq x < \text{row} + 480/P; 0 \leq y < 640:$

$x' = x + \Delta x$

$y' = y + \Delta y$

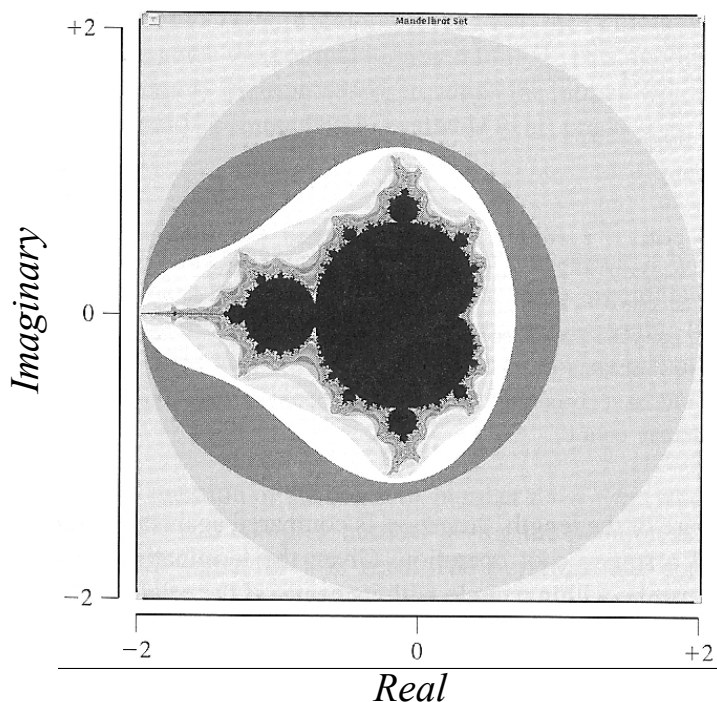
```
recv(&row, master);
for (oldrow = row; oldrow < row + 480/P; oldrow++)
{ for (oldcol = 0; oldcol < 640; oldcol++)
  { newrow = oldrow + delta_x;
    newcol = oldcol + delta_y;
    send(oldrow, oldcol, newrow, newcol, master);
  }
}
```



Example 1: Geometrical Transformation Speedups?

- Assume in the general case the pixmap has n^2 points
- Sequential time of pixmap shift $t_s = 2n^2$
- Communication
$$t_{comm} = P(t_{startup} + t_{data}) + n^2(t_{startup} + 4t_{data}) = O(P + n^2)$$
- Computation
$$t_{comp} = 2n^2 / P = O(n^2/P)$$
- Computation/communication ratio
$$r = O((n^2 / P) / (P + n^2)) = O(n^2 / (P^2 + n^2P))$$
- **This is not good!**
 - The asymptotic computation time should be an order higher than the asymptotic communication time, e.g. $O(n^2)$ versus $O(n)$
 - ... or there must be a very large constant in the computation time
- Performance on shared memory machine can be good
 - No communication time

Example 2: Mandelbrot Set



The number of iterations it takes for z to end up at a point outside the complex circle with radius 2 determines the pixmap color

- A pixmap is generated by iterating the complex-valued recurrence

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

with $z_0=0$ and $c=x+yi$ until $|z|\geq 2$

- The Mandelbrot set is shifted and scaled for display:

$$x = x_{min} + x_{scale} \text{ row}$$

$$y = y_{min} + y_{scale} \text{ col}$$

for each of the pixmap's pixels at row and col location



Example 2: Mandelbrot Set Color Computation

```
int pix_color(float x0, float y0)
{
    float x = x0, y = y0;
    int i = 0;

    while (x*x + y*y < (2*2) && i < maxiter)
    {
        float xtemp = x*x - y*y + x0;
        float ytemp = 2*x*y + y0;

        x = xtemp;
        y = ytemp;

        i++;
    }

    return i;
}
```



Example 2: Mandelbrot Set

Simple Master and Worker

Master

```
row = 0;
for (p = 0; p < P; p++)
{ send(row, p);
  row += 480/P;
}
for (i = 0; i < 480 * 640; i++)
{ recv(&x, &y, &color, anyP);
  display(x, y, color);
}
```

Send/recv (x,y) pixel colors

Worker

```
recv(&row, master);
for (y = row; y < row + 480/P; y++)
{ for (x = 0; x < 640; x++)
  { x0 = xmin + x * xscale;
    y0 = ymin + y * yscale;
    color = pix_color(x0, y0);
    send(x, y, color, master);
  }
}
```



Example 2: Mandelbrot Set

Better Master and Worker

Master

```
row = 0;
for (p = 0; p < P; p++)
{ send(row, p);
  row += 480/P;
}
for (i = 0; i < 480; i++)
{ recv(&y, &color, anyP);
  for (x = 0; x < 640; x++)
    display(x, y, color[x]);
}
```

*Send/recv array of colors[x]
for each row y*

*Assume $n \times n$ pixmap, n iterations on average
per pixel, and P workers:*

Communication time?

Computation time?

Computation/communication ratio?

Speedup?

*Improve the solution with communication
latency hiding. Hint: reuse workers*

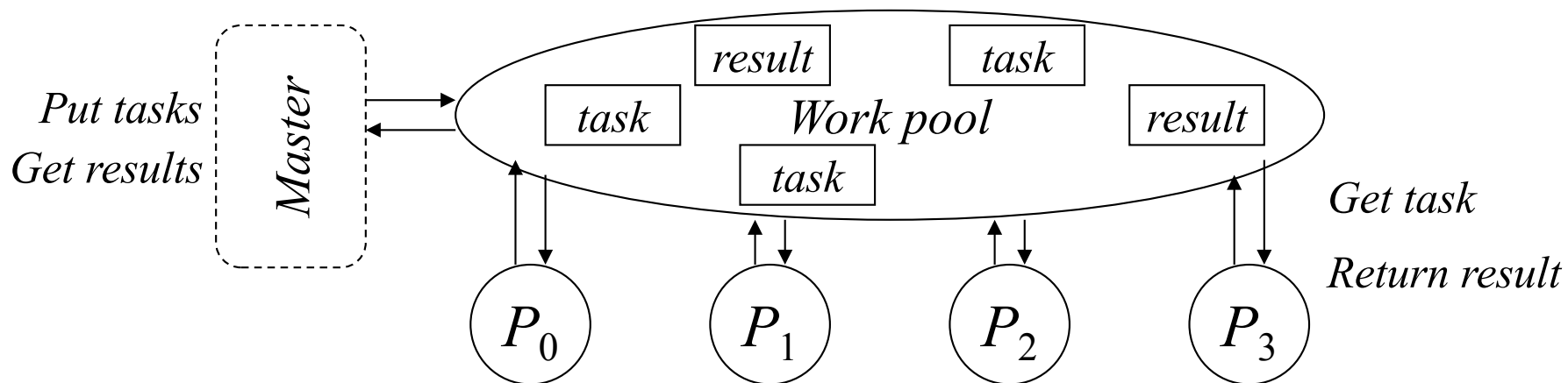
Worker

```
recv(&row, master);
for (y = row; y < row + 480/P; y++)
{ for (x = 0; x < 640; x++)
  { x0 = xmin + x * xscale;
    y0 = ymin + y * yscale;
    color[x] = pix_color(x0, y0);
  }
  send(y, color, master);
}
```



Processor Farms

- *Processor farms* (also called the *work-pool approach*)
- A collection of workers, where each worker repeats:
 - Take new task from pool
 - Compute task
 - Return results into pool
- Achieves *load balancing*
 - Tasks differ in amount of work
 - Workers can differ in execution speed (viz. heterogeneous cluster)





Example 2: Mandelbrot Set with Processor Farm

Master

Assuming synchronous send/recv

```
count = 0;
row = 0;
for (p = 0; p < P; p++)
{ send(row, p);
  count++;
  row++;
}
do
{ recv(&y, &color, anyP);
  count--;
  if (row < 480)
  { send(row, anyP);
    row++;
    count++;
  }
  else
    send(-1, anyP);
  for (x = 0; x < 640; x++)
    display(x, y, color[x]);
} while (count > 0);
```

Keeps track of how many workers are active

Send row to workers

Recv colors for row y

Send next row

Send sentinel

Compute color for (x,y)

Send colors

Recv next row

Worker

```
recv(&y, master);
while (y != -1)
{ for (x = 0; x < 640; x++)
  { x0 = xmin + x * xscale;
    y0 = ymin + y * yscale;
    color[x] = pix_color(x0, y0);
  }
  send(y, color, master);
  recv(&y, master);
}
```



Example 2: Mandelbrot Set with Processor Farm

Master

```
count = 0;
row = 0;
for (p = 0; p < P; p++)
{ send(row, p);
  count++;
  row++;
}
do
{ recv(&rank, &y, &color, anyP);
  count--;
  if (row < 480)
  { send(row, rank);
    row++;
    count++;
  }
  else
    send(-1, rank);
  for (x = 0; x < 640; x++)
    display(x, y, color[x]);
} while (count > 0);
```

Assuming asynchronous send/recv

*Recv a row y and worker rank
and send to worker (using rank)*

Worker

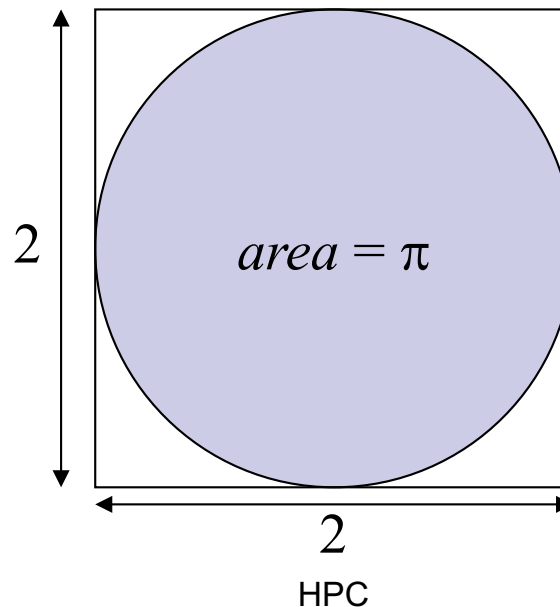
```
recv(&y, master);
while (y != -1)
{ for (x = 0; x < 640; x++)
  { x0 = xmin + x * xscale;
    y0 = ymin + y * yscale;
    color[x] = pix_color(x0, y0);
  }
  send(myrank, y, color, master);
  recv(&y, master);
}
```

*Send colors
and rank*



Example 3: Monte Carlo Methods

- Perform random selections to sample the solution
- Each sample is independent
- Example
 - Compute π by sampling the $[-1..1, -1..1]$ square that contains a circle with radius 1
 - The probability of hitting the circle is $\pi/4$

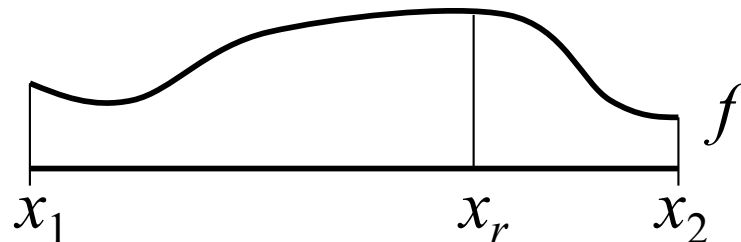




Example 3: Monte Carlo Methods

- General Monte Carlo methods sample inside and outside the solution space
- Many Monte Carlo methods do not sample outside solution space
- Function integration by sampling the function values over the integration domain

$$\int_{x_1}^{x_2} f(x) dx = \lim_{N \rightarrow \infty} \frac{x_2 - x_1}{N} \sum_{r=1}^N f(x_r)$$





Example 3: Monte Carlo Methods and Parallel RNGs

- Approach 1: master sends random number sequences to the workers
 - Uses one random number generator (RNG)
 - Lots of communication
- Approach 2: workers produce independent random number sequences
 - Communication of sample parameters only
 - Cannot use standard pseudo RNG (sequences are the same)
 - Needs parallel RNG
- Parallel RNGs (e.g. SPRNG library)
 - Parallel pseudo RNG
 - Parallel quasi-random RNG



Example 3: Monte Carlo Methods and Parallel RNGs

- *Linear congruential generator (pseudo RNG):*

$$x_{i+1} = (a x_i + c) \bmod m$$

with a choice of a , c , and m

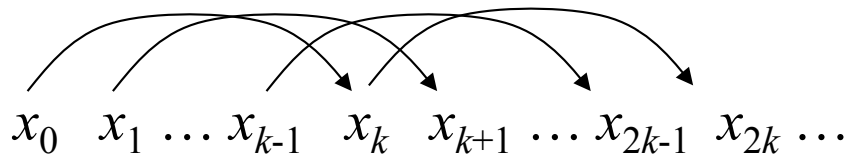
- Good choice of a , c , and m is crucial!
- Cannot easily segment the sequence (for processors)

- A parallel pseudo RNG with a “jump” constant k

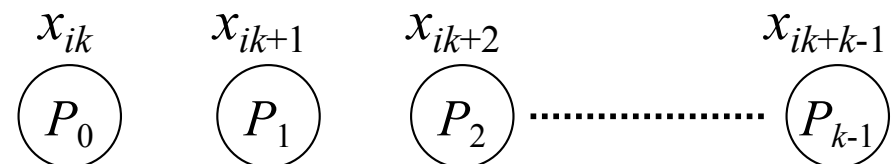
$$x_{i+k} = (A x_i + C) \bmod m$$

where $A = a^k \bmod m$, $C = c(a^{k-1} + a^{k-2} + \dots + a^1 + a^0) \bmod m$

Parallel computation of sequence



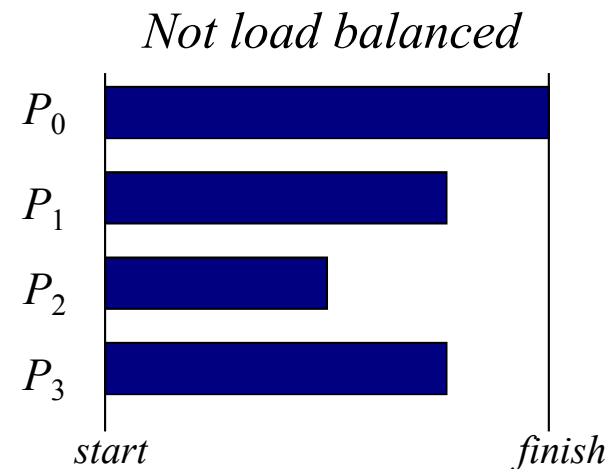
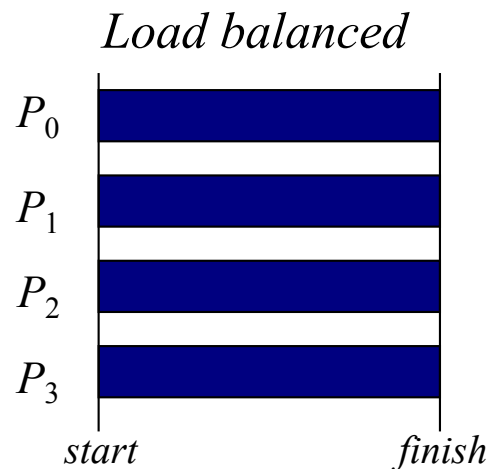
The sequences per processor





Load Balancing

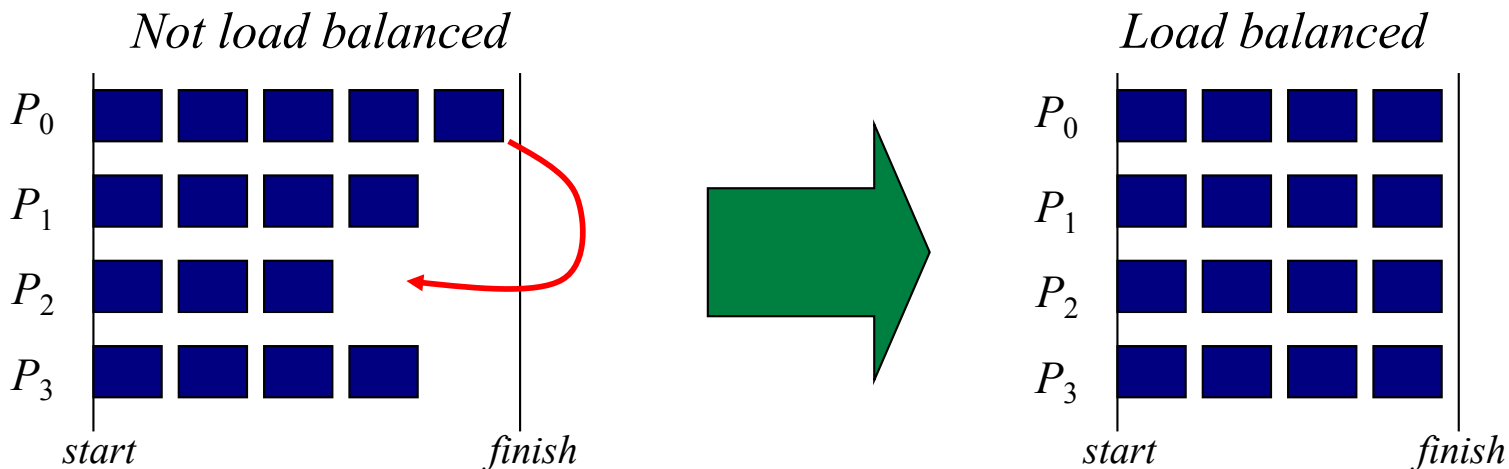
- Load balancing attempts to spread tasks evenly across processors
- Load imbalance is caused by
 - Tasks of different execution cost, e.g. Mandelbrot example
 - Processors operate with different execution speeds or are busy
- When tasks and processors are not load balanced:
 - Some processes finish early and sit idle waiting
 - Global computation is finished when the slowest processor(s) completes its task





Static Load Balancing

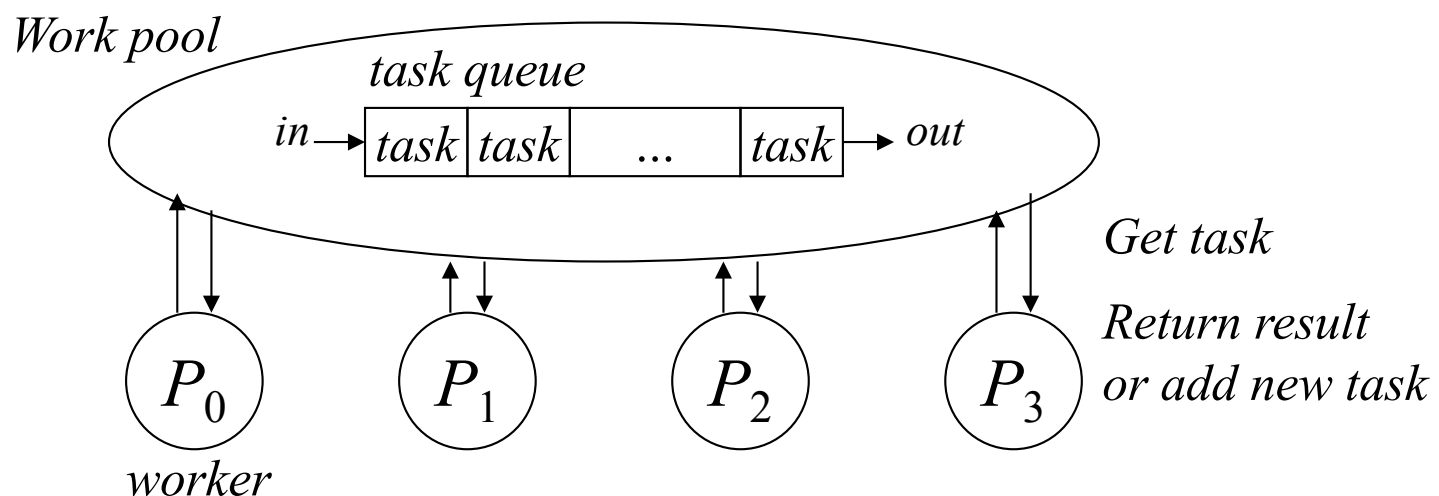
- Load balancing can be viewed as a form of “bin packing”
- *Static scheduling* of tasks amounts to optimal bin packing
 - Round robin algorithm
 - Randomized algorithms
 - Recursive bisection
 - Optimized scheduling with simulated annealing and genetic algorithms
- Problem: difficult to estimate amount of work per task, deal with changes in processor utilization and communication latencies





Centralized Dynamic Load Balancing

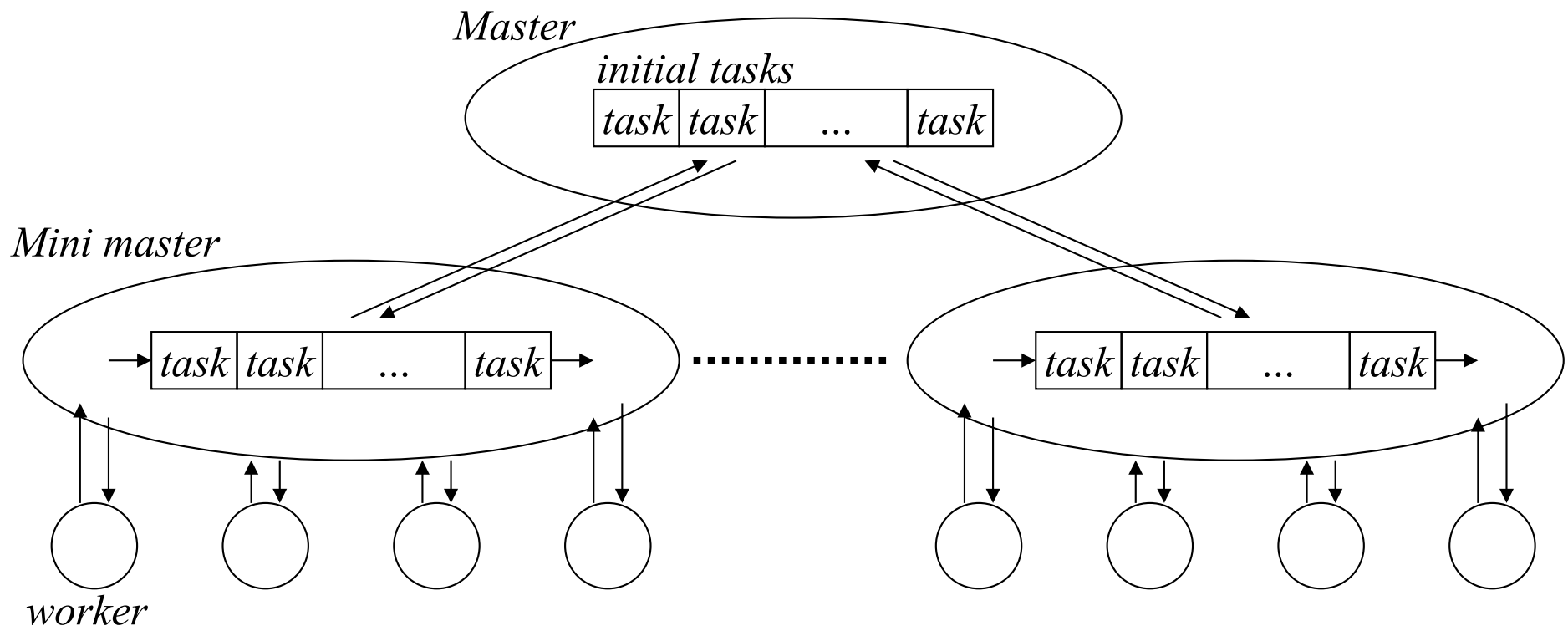
- Centralized: work pool with replicated workers
- Master process or central queue holds incomplete tasks
 - First-in-first-out or priority queue (e.g. priority based on task size)
- Terminates when queue is empty or workers receive termination signal





Decentralized Dynamic Load Balancing

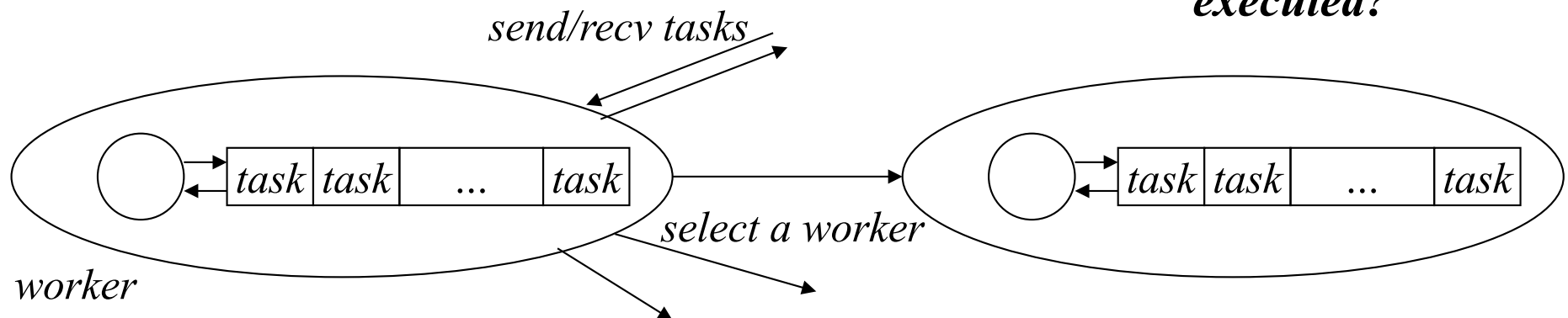
- Disadvantage of centralized approach is the central queue through which tasks move one by one
- Decentralized: distributed work pools



Fully Distributed Work Pool

- *Receiver-initiated poll method*: (an idle) worker process requests a task from another worker process
- *Sender-initiated push method*: (an overloaded) worker process sends a task to another (idle) worker
- Workers maintain local task queues
- Process selection
 - Topology-based: select nearest neighbors
 - Round-robin: try each of the other workers in turn
 - Random polling/pushing: pick an arbitrary worker

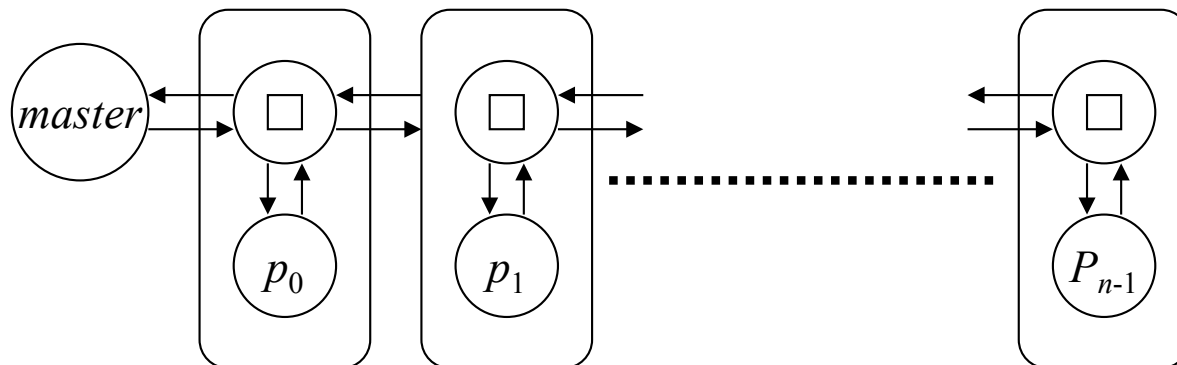
*Absence of starvation:
assume ∞ tasks,
how can we
guarantee each one
is eventually
executed?*





Worker Pipeline

- Workers are organized in an array (or ring) with the master on one end (or middle)
 - Master feeds the pipeline
 - When the buffer of a worker is idle, it sends a request to the left
 - When the buffer of a worker is full, incoming tasks are shifted to the worker on the right (passing task along until an empty slot)





Further Reading

- [PP2] pages 79-99, 201-210