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

![Diagram showing the master-worker paradigm with processes P0, P1, P2, and P3. The diagram illustrates the flow of sending initial data from the master to the processes and the collection of results.]
Parallel Tradeoffs

- Embarrassingly parallel with perfect load balancing:
  \[ t_{\text{comp}} = \frac{t_s}{P} \]
  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
    \[ t_s > t_p = t_{\text{comm}} + t_{\text{comp}} \]
  - Speedup \( S_P = \frac{t_s}{t_p} = P \frac{t_{\text{comp}}}{t_{\text{comm}} + t_{\text{comp}}} = P / (r^{-1} + 1) \)
    where \( r = \frac{t_{\text{comp}}}{t_{\text{comm}}} \)
  - Thus \( S_P \to P \) when \( r \to \infty \)

- However, communication \( t_{\text{comm}} \) can be expensive
  - Typically \( t_s < t_{\text{comm}} \) for small tasks, that is, the time to send/recv data to the workers is more expensive than doing all the work
Example 1: Geometrical Transformations of Images

- 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; 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:
∀ row ≤ x < row + 480/P; 0 ≤ y < 640:
  x’ = x + Δx
  y’ = y + Δy
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
  
  $t_{\text{comm}} = P(t_{\text{startup}} + t_{\text{data}}) + n^2(t_{\text{startup}} + 4t_{\text{data}}) = O(P + n^2)$
- Computation
  
  $t_{\text{comp}} = 2n^2 / P = O(n^2/P)$
- 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

- 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:
  \[
  x = x_{\text{min}} + x_{\text{scale}} \text{ row} \\
  y = y_{\text{min}} + y_{\text{scale}} \text{ col}
  \]
  for each of the pixmap’s pixels at \text{row} and \text{col} location

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.
Example 2: Mandelbrot Set
Color Computation

```c
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**

```c
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);
}
```

**Worker**

```c
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);
  }
}
```

Send/recv (x,y) pixel colors
Example 2: Mandelbrot Set
Better Master and Worker

Master

```c
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]);
}
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);
}
```

Worker

Assume \( n \times n \) pixmap, \( n \) iterations on average per pixel, and \( P \) workers:
- Communication time?
- Computation time?
- Computation/communication ratio?
- Speedup?

Send/recv array of colors\([x]\) for each row \( y \)
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

```c
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);
```

Worker

```c
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);
}```

Assuming synchronous send/recv

- Keeps track of how many workers are active
- Send initial row to all workers
- Recv colors for row y
- Send next row
- Send sentinel
- Compute color for (x,y)
- Send colors
- Recv next row
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

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);
Example 3: Monte Carlo Methods

- Perform random selections to sample the solution
- Each sample is independent
- Example
  - Compute $\pi$ 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

\[
\int_{x_1}^{x_2} f(x) \, dx = \lim_{N \to \infty} \frac{x_2 - x_1}{N} \sum_{r=1}^{N} f(x_r)
\]
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):
  \[ x_{i+1} = (a \cdot x_i + c) \mod m \]
  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 \(k\)
  \[ x_{i+k} = (A \cdot x_i + C) \mod m \]
  where \(A = a^k \mod m\), \(C = c(a^{k-1} + a^{k-2} + \ldots + a^1 + a^0) \mod m\)

**Parallel computation of sequence**

\[
\begin{align*}
  x_0 & x_1 \ldots x_{k-1} & x_k & x_{k+1} \ldots x_{2k-1} & x_{2k} \ldots \\
  P_0 & P_1 & P_2 & \ldots & P_{k-1}
\end{align*}
\]

**The sequences per processor**

\[
\begin{align*}
  x_{ik} & x_{ik+1} & x_{ik+2} & \ldots & x_{ik+k-1}
\end{align*}
\]
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

**Diagram:**

- **Load balanced**
  - Processes start and finish at the same time.
  - All processors are active throughout.

- **Not load balanced**
  - Processes start at the same time, but finish at different times.
  - Processors have varying execution times, leading to imbalance.

![Diagram showing load balanced and not load balanced scenarios]
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

Assume $\infty$ 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