# Programming Models

Parallel Computing Patterns

## Parallel Computing Patterns

- Design guidelines to implement a parallelized version from a sequential code
- Based on 4 design spaces concerning both algorithm expression and software construction:

<table>
<thead>
<tr>
<th>Algorithm Expression</th>
<th>Software Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Finding Concurrency</strong></td>
<td><strong>3. Supporting Structures</strong></td>
</tr>
<tr>
<td>– Expose concurrent tasks</td>
<td>• Code and data structuring patterns</td>
</tr>
<tr>
<td><strong>2. Algorithm Structure</strong></td>
<td><strong>4. Implementation Mechanisms</strong></td>
</tr>
<tr>
<td>– Map tasks to processes to</td>
<td>• Low level mechanisms used to write parallel programs</td>
</tr>
<tr>
<td>– exploit parallel architecture</td>
<td></td>
</tr>
</tbody>
</table>


Motivation

- Example: MPEG decoder
- Program complexity ask for design guidelines for parallelization

Example: MPEG Decoder

- Task decomposition
  - Independent coarse-grained computation
  - Inherent to algorithm
- Sequence of statements (instructions) that operate together as a group
  - Corresponds to some logical part of program
  - Usually follows from the way programmer thinks about a problem
Example: MPEG Decoder

• Task decomposition
  – Parallelism in the application

• Data decomposition
  – Same computation is applied to small data chunks derived from large data set
Patterns & Decompositions

• Patterns are more specific than decomposition strategies as we discussed earlier in the course

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task-level parallelism</td>
<td>Task</td>
</tr>
<tr>
<td>Divide and Conquer</td>
<td>Task/Data</td>
</tr>
<tr>
<td>Geometric Decomposition</td>
<td>Data</td>
</tr>
<tr>
<td>Pipeline</td>
<td>Data Flow</td>
</tr>
<tr>
<td>Wavefront</td>
<td>Data Flow</td>
</tr>
</tbody>
</table>

Design Spaces in Constructing a Parallel Program

• Structure the problem to expose exploitable concurrency
• Structure the algorithm to take advantage of concurrency
• Intermediate stage between Algorithm Structure and Implementation
  – program structuring
  – definition of shared data structures
• Mapping of the higher level patterns onto a programming environment
Finding Concurrency Design Space

• Result
  – A task decomposition that identifies tasks that can execute concurrently
  – A data decomposition that identifies data local to each task and data shared among tasks
  – A way of grouping tasks and ordering them according to data dependencies and temporal constraints
• This will be used as an input for the Algorithm Structure design space

Algorithm Structure Design Space

• Given a collection of concurrent tasks, what’s the next step?
• Map tasks to units of execution (e.g., threads)
• Important considerations
  – Magnitude of number of execution units platform will support
  – Cost of sharing information among execution units
  – Avoid tendency to over constrain the implementation
    • Work well on the intended platform
    • Flexible enough to easily adapt to different architectures
Major Organizing Principle

• How to determine the algorithm structure that represents the mapping of tasks to units of execution?
• Concurrency usually implies major organizing principle
  – Organize by tasks
  – Organize by data decomposition
  – Organize by flow of data

Organizing Principles
Organize by Tasks?

Recursive? yes → Divide and Conquer

Recursive? no → Task Parallelism

Task Parallelism

• Problem can be decomposed into a collection of tasks that can execute concurrently
• Tasks can be completely independent (embarrassingly parallel) or can have dependencies among them
• All tasks might be known at the beginning or might be generated dynamically
Task Parallelism

• Tasks:
  – There should be at least as many tasks as UEs (Units of Execution) - typically many, many more
  – Computation associated with each task should be large enough to offset the overhead associated with managing tasks and handling dependencies

• Types of dependencies:
  – Ordering constraints: sequential composition of task-parallel computations
  – Shared-data dependencies: several tasks have to access the same data structure

Shared Data Dependencies

• Shared data dependencies can be categorized as follows:
  – Removable dependencies: an apparent dependency that can be removed by code transformation

```c
int i, ii=0, jj=0;
for (i=0; i<N; i++) {
    ii = ii + 1;
    d[ii] = big_time-consuming_work (ii);
    jj = jj + 1;
    s[jj] = other_big_time-consuming_work (jj);
}
```

```c
for (i=0; i<N; i++) {
    d[i] = big_time-consuming_work (i);
    s[(i*i+i)/2] = other_big_time-consuming_work((i*i+i)/2);
}
```
Shared Data Dependencies

- **Separable dependencies:**
  - Write-once updates or accumulative sum on shared variables
  - Can be pulled outside the concurrent execution by replicating the shared data structure before and combine the copies into a single structure after the concurrent execution

- **Other dependencies:** non-resolvable, have to be followed
  - Protected dependencies: variables read and written during the concurrent execution

Embarrassingly Parallel Pattern

- Independent tasks
- Computation of solutions
  - Independent on distinct variables
  - Accumulated in a shared data structure (if no ordering is required)
  - ...

- Examples:
  - Vector addition
  - Ray tracing codes
  - Database searches
  - Branch and bound
Application Examples

• Low level image processing
• Mandelbrot set
• Monte Carlo Calculations

Partitioning into Regions for Individual Processes

**Shifting**
- Object shifted by $Dx$ in the $x$-dimension and $Dy$ in the $y$-dimension:
  
  \[
  x' = x + \Delta x \\
  y' = y + \Delta y
  \]
- where $x$ and $y$ are the original and $x'$ and $y'$ are the new coordinates.

**Scaling**
- Object scaled by a factor $S_x$ in $x$-direction and $S_y$ in $y$-direction:
  
  \[
  x' = x S_x \\
  y' = y S_y
  \]

**Rotation**
- Object rotated through an angle $\theta$ about the origin of the coordinate system:
  
  \[
  x' = x \cos \theta + y \sin \theta \\
  y' = -x \sin \theta + y \cos \theta
  \]
**Complexity Analysis: Sequential**

- Suppose each pixel requires one computational step and there are $n \times n$ pixels.

**Sequential**
- $t_s = n^2$ and a sequential time complexity of $O(n^2)$

---

**Pseudocode to Perform Image Shift**

**Master**

```c
for (i = 0; row = 0; i < 48; i++) /* for each process */
    for (j = 0; row = 0; j < 44; j++)
        temp_map[i][j] = 0;

for (i = 0; i < 480; i++) /* initialize temp */
    for (j = 0; j < 440; j++)
        temp_map[i][j] = 0;

for (i = 0; i < 480; i++) /* for each pixel */
    for (j = 0; j < 440; j++)
        if ((newrow < 0) || (newrow > 480) || (newcol < 0) || (newcol > 440))
            temp_map[newrow][newcol] = temp_map[oldrow][oldcol];

for (i = 0; i < 480; i++) /* update bitmap */
    for (j = 0; j < 440; j++)
        map[i][j] = temp_map[i][j];
```

**Slave**

```c
recv(row, &master); /* receive row no. */
for (oldrow = row; oldrow < (row + 10); oldrow++)
    for (oldcol = 0; oldcol < 440; oldcol++)/* transform coords */
        newrow = oldrow + delta_x;
        newcol = oldcol + delta_y; /* shift in x direction */
        send(oldrow, oldcol, newrow, newcol, &master); /* coords to master */
```
Complexity Analysis: Parallel

Parallel

- **Communication**
  - $t_{\text{comm}} = t_{\text{startup}} + mt_{\text{data}}$
  - $t_{\text{comm}} = p(t_{\text{startup}} + 2t_{\text{data}}) + 4n^2(t_{\text{startup}} + t_{\text{data}}) = O(p + n^2)$

- **Computation**
  - $t_{\text{comp}} = 2(n^2/p) = O(n^2/p)$

- **Overall Execution Time**
  - $t_p = t_{\text{comp}} + t_{\text{comm}}$
  - For constant $p$, this is $O(n^2)$.

- However, the constant hidden in the communication part far exceeds those constants in the computation in most practical situations.

Separable Dependencies Pattern

- Necessary global data are replicated and partial results are stored in local data structures
- Global results are obtained by reducing results from individual tasks
- Examples
  - Matrix-vector multiplication
  - Numerical integration
Task scheduling

• Schedule: the way in which tasks are assigned to UEs for execution
  – Minimize the overall execution of all tasks
  – Finish the work at the same time (load balance)

• Two classes of schedule:
  – Static schedule: distribution of tasks to UEs is determined at the start of the computation and not changed anymore
  – Dynamic schedule: the distribution of tasks to UEs changes as the computation proceeds

Task scheduling - example

• Embarrassingly parallel pattern
Static Schedule

• Tasks are associated into blocks
  – Blocks are assigned to UEs
  – Each UE should take approximately same amount of time to complete task

• Static schedule usually used when
  – Availability of computational resources is predictable (e.g. dedicated usage of nodes)
  – UEs are identical (e.g. homogeneous parallel computer)
  – Size of each task is nearly identical

Dynamic scheduling

• Used when
  – Effort associated with each task varies widely/is unpredictable
  – Capabilities of UEs vary widely (heterogeneous parallel machine)

• Common implementations:
  – usage of task queues: if a UE finishes current task, it removes the next task from the task-queue
  – Work-stealing:
    • each UE has its own work queue
    • once its queue is empty, a UE steals work from the task queue of another UE
Dynamic scheduling

• Trade-offs:
  – Fine grained (=shorter, smaller) tasks allow for better load balance
  – Fine grained task have higher costs for task management and dependency management

Task Parallelism using Master-Worker framework
Task Parallelism using work stealing

Divide and Conquer

```c
int solve ( Problem P )
{
    int solution;

    /* Check whether we can further partition the problem */
    if ( baseCase ( P ) ) {
        solution = baseSolve ( P ); /* No, we can't */
    } else { /* yes, we can */
        Problem subproblems [ N ];
        int subsolutions [ N ];

        subproblems = split ( P ); /* Partition the problem */
        for ( i = 0; i < N; i++ ) {
            subsolutions [ i ] = solve ( subproblems [ i ] );
        }
        solution = merge ( subsolutions );
    }
    return ( solution );
}
```
Divide and Conquer

- A problem is split into a number of smaller sub-problems
- Each sub-problem is solved independently
- Sub-solutions of each sub-problem will be merged to the solution of the final problem
  - Useful if the base case is large compared to the work needed for splitting-merging
- Problems of Divide and Conquer for Parallel Computing:
  - Amount of exploitable concurrency decreases over the lifetime
  - Trivial parallel implementation: each function call to solve is a task on its own. For small problems, no new task should be generated, but the baseSolve should be applied

Divide and Conquer

- Implementation:
  - On shared memory machines, a divide and conquer algorithm can easily be mapped to a fork/join model
    - A new task if forked(=created)
    - After this task is done, it joins the original task (=destroyed)
  - If the problem is not regular, better to use fine grained tasks and a task queue
    - Often implemented using the Master/Worker framework
  - OpenMP can be used to parallelize the loop only if it supports nesting of parallel regions which is not always true [Mat03]
Divide and Conquer

• Issues:
  – Sub-problems may not be uniform
  – May require dynamic load balancing
Divide and Conquer: Task Assignment

Example: Mergesort

```java
function mergesort(m)
    var list left, right
    if length(m) ≤ 1
        return m
    else
        middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = mergesort(left)
        right = mergesort(right)
        result = merge(left, right)
        return result
    end if
end function
```
Example: Adding a List of Numbers

• A sequential recursive definition for adding a list of numbers is

```c
int add(int *s)   /* add list of numbers, s */
{
    if (number(s) <= 2) return (n1 + n2); /* see explanation */
    else {
        Divide (s, s1, s2);   /* divide s into two parts, s1 and s2 */
        part_sum1 = add(s1);   /* recursive calls to add sub lists */
        part_sum2 = add(s2);
        return (part_sum1 + part_sum2);
    }
}
```

M-ary Divide and Conquer

• Divide and conquer can also be applied where a task is divided into more than two parts at each stage
• For example, if the task is broken into four parts, the sequential recursive definition would be

```c
int add(int *s)   /* add list of numbers, s */
{
    if (number(s) <= 4) return(n1 + n2 + n3 + n4);
    else {
        Divide (s,s1,s2,s3,s4);   /* divide s into s1,s2,s3 */
        part_sum1 = add(s1);   /* recursive calls to add */
        part_sum2 = add(s2);
        part_sum3 = add(s3);
        part_sum4 = add(s4);
        return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
    }
}
```
Bucket Sort

- One “bucket” assigned to hold numbers that fall within each region. Numbers in each bucket sorted using a sequential sorting algorithm.

- Sequential sorting time complexity: $O(n \log(n/m))$.
- Works well if the original numbers uniformly distributed across a known interval, say 0 to $a - 1$.

Parallel Version of Bucket Sort

- Assign one processor for each bucket.
Further Parallelization

- By partitioning the sequence into $m$ regions, one region for each processor
- Each processor maintains $p$ "small" buckets and separates the numbers in its region into its own small buckets
- These small buckets are then "emptied" into the $p$ final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket $i$ to processor $i$)

Another Parallel Version

Introduces new message-passing operation - all-to-all broadcast.
Analysis

- The following phases are needed:
  1. Partition numbers
  2. Sort into small buckets.
  3. Send to large buckets.
  4. Sort large buckets.

**Phase 1 — Computation and Communication**
- \( t_{comp1} = n \)
- \( t_{comm1} = t_{startup} + t_{data} n \)

**Phase 2 — Computation**
- \( t_{comp2} = n/p \)

**Phase 3 — Communication**
- If all the communications could overlap:
  - \( t_{comm3} = (p - 1)(t_{startup} + (n/p^2)t_{data}) \)

**Phase 4 — Computation**
- \( t_{comp4} = (n/p)\log(n/p) \)

**Overall**
- \( t_p = t_{startup} + t_{data} n + n/p + (p - 1)(t_{startup} + (n/p^2)t_{data}) + (n/p)\log(n/p) \)

- It is assumed that the numbers are uniformly distributed to obtain these formulas. The worst-case scenario would occur when all the numbers fell into one bucket!
All-to-all Routine

- Sends data from each process to every other process

Other Interesting Examples

- Gravitational N-Body problem
  - Barnes-Hut algorithm
  - Orthogonal recursive bisection
Algorithm Structure – Summary so far

- Task parallelism:
  - Implemented by Task queues
  - Task distribution vs. work stealing
- Divide and Conquer for recursive problems
  - Split problem into sub-problems until a lower limit in the problem size has been reached
  - Solve the sub-problem
  - Merge the results of the sub-problems into the final result

Organize by Data?

- Operations on a central data structure
  - Arrays and linear data structures
  - Recursive data structures
Geometric Decomposition

• For all applications relying on data decomposition
  – All processes should apply the same operations on different data items

• Key elements:
  – Data decomposition
  – Exchange and update operation
  – Data distribution and task scheduling

Geometric Decomposition: Example

• Scalar product and matrix vector multiplications are used to solve differential equations

• They can be performed in parallel using geometric decomposition
Scalar Product

• Scalar product:

\[ s = \sum_{i=0}^{N-1} a[i] \cdot b[i] \]

• Parallel algorithm:

\[ s = \sum_{i=0}^{N/2-1} (a[i] \cdot b[i]) + \sum_{i=N/2}^{N-1} (a[i] \cdot b[i]) \]

\[ = \sum_{i=0}^{N/2-1} (a_{\text{local}}[i] \cdot b_{\text{local}}[i]) + \sum_{i=N/2}^{N-1} (a_{\text{local}}[i] \cdot b_{\text{local}}[i]) \]

– requires communication between the processes

Matrix-Vector product in Parallel

\[ \begin{bmatrix} -50 & 30 \\ 20 & -50 & 30 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} rhs_1 \\ rhs_2 \end{bmatrix} \]

\[ \begin{bmatrix} 20 & -50 & 30 \\ 20 & -50 & 30 \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} rhs_3 \\ rhs_4 \end{bmatrix} \]
Matrix-Vector product in Parallel

- Introduction of ghost cells
  
  - Looking at the source code, e.g. ...
  
  - ...since the vector used in the matrix vector multiplication changes every iteration, you always have to update the ghost cells before doing the calculation

Recursive Data

- Typically applied in recursive data structures
  - Lists, trees, graphs
- Data decomposition: recursive data structure is completely decomposed into individual elements
- Example: prefix scan operation
  - Each process has an element of an overall structure (e.g. a linked list), e.g. an integer x
- Lets denote the value of the x on process i xi
  - At the end of the prefix scan operation process k holds the sum of all elements of xi for i=0...k
Recursive Data

- Example for eight processes

```
Before prefix scan
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>X_0</td>
<td>X_1</td>
<td>X_2</td>
<td>X_3</td>
<td>X_4</td>
<td>X_5</td>
<td>X_6</td>
<td>X_7</td>
</tr>
</tbody>
</table>
```

```
After prefix scan
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>X_0</td>
<td>X_1</td>
<td>X_2</td>
<td>X_3</td>
<td>X_4</td>
<td>X_5</td>
<td>X_6</td>
<td>X_7</td>
</tr>
</tbody>
</table>
```

Sequential implementation

- Each process forwards its sum to the next process
  - n messages/ time steps required for n processes

```
Before prefix scan
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>x_0</td>
<td>x_1</td>
<td>x_2</td>
<td>x_3</td>
<td>x_4</td>
<td>x_5</td>
<td>x_6</td>
<td>x_7</td>
</tr>
</tbody>
</table>
```

```
After prefix scan
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>x_0</td>
<td>x_1</td>
<td>x_2</td>
<td>x_3</td>
<td>x_4</td>
<td>x_5</td>
<td>x_6</td>
<td>x_7</td>
</tr>
</tbody>
</table>
```

```
Time t
```

- Each process forwards its sum to the next process
  - n messages/ time steps required for n processes
Recursive data approach

Another Example: Find the Root

- Given a forest of rooted directed trees, for each node, find the root of the tree containing the node
  - Parallel approach: for each node, find its successor’s successor, repeat until no changes
    - $O(\log n)$ vs. $O(n)$
  
- In the example, three steps are needed to converge (all the nodes have no more iterations to do)
Recursive data approach

- Very fine grained concurrency
- Restructuring of the original algorithm often required
- Parallel algorithm requires substantially more work, which can however be executed in less time-steps

Organize by Ordering or Flow of Data?

- Regular? yes → Pipeline
- Regular? no → Event-based Coordination
Pipeline pattern

• Amount of concurrency limited to the number of stages of the pipeline
• Patterns works best, if amount of work performed by various stages is roughly equal
• Filling the pipeline: some stages will be idle
• Draining the pipeline: some stages will be idle
• Non-linear pipeline: pattern allows for different execution for different data items

Pipeline pattern

• Implementation:
  – Each stage typically assigned to a process/thread
  – A stage might be a data-parallel task itself
  – Computation per task has to be large enough to compensate for communication costs between the tasks
Event-based coordination

- Pipeline pattern assumes a regular, non-changing data flow
- Event-based coordination assumes irregular interaction between tasks
- Real world example:
  - Data items might flow in both directions
    - Each data item might take a different path
- Major problem: deadlock avoidance

Supporting structures

- Supporting structures describe software constructions for parallel algorithms
SPMD

- SPMD – Single Program Multiple Data
- Each UE carries out similar/identical operations
- Interaction between UEs performance critical
  - Basically all applications scaling up to several thousand nodes/processors are written in the SPMD style

SPMD

- Basic elements:
  - Initialize: establish common context on each UE
  - Obtain unique identifier: e.g. using MPI_Comm_rank()
  - Run the same program on each UE using the unique identifier to differentiate behavior on different UEs
- Differentiation could also be done based on data items
  - Distribute data: e.g. geometric decomposition
  - Finalize
SPMD Example

• Anti-differentiation: Given a function \( f(x) \), find a function \( F(x) \) with the property that \( F'(x) = f(x) \).

• Example: \( f(x) = ax^n \longrightarrow F(x) = \frac{1}{n+1} ax^{n+1} + c \)

• Calculating the Integral of a function \( \int_a^b f(x)dx = F(b) - F(a) \)

• Graphical interpretation

Sequential Code using MPI

• Trapezoid rule

\[
\int_a^b f(x)dx = \sum_{i=1}^{n} \frac{b-a}{n} f(x_i) = \frac{b-a}{2n} \sum_{i=1}^{n} (x_i - x_{i-1}) [f(x_{i-1}) + f(x_i)]
\]

```c
#include <stdio.h>

int main( int argc, char **argv )
{
    int i, num_steps=100000;
    double x, xn, pi, step, sum=0.0;

    /* Required input: 
    - a, b : boundaries of the integral 
    - f(x): function 
    */

    step = (b-a)/num_steps;
    for ( i=0; i<num_steps; i++ ) 
    {
        x = i*step;
        xn = (i+1)*step;
        sum = sum + 0.5*(xn-x) * (f(x)+f(xn));
    }

    return (0);
}
```
Parallel Code using MPI

```c
int rank, size, start, end, i, num_steps=160000;
double x, xn, end, step, sum, lsum=0.0;
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);

step = (2*a)/num_steps;
start = rank * num_steps/size;
end = start + num_steps/size;

for (i=start; i<end; i++) {
    x = i * step;
    \textcolor{red}{n} \textcolor{blue}{=} \textcolor{green}{(i+1) \times step};
    lsum = lsum + 0.5*(x-n) \times f(x)+f(xn);
}
MPI_Allreduce (lsum, sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
MPI_Finalize ();
```

Master-Worker Pattern

- Master
- Independent Tasks
- Workers
- Master Process: 
  - Result queue
  - Task queue
- Worker Processes: 
  - Task queue
Master-Worker Pattern

• Particularly relevant for problems using task parallelism pattern where task have no dependencies
  – Embarrassingly parallel problems
• In general, it is useful if
  – Workload associated with tasks are highly variable – MW has ‘built-in’ load balancing
  – Capabilities of PEs are strongly varying
  – Tasks are not tightly coupled – each worker process typically only has to communicate with the master process but not with other workers
• Not useful usually if the computationally intensive part of the program structure is organized in a big loop

Master-Worker Pattern

• Main challenge in determining when the entire problem is complete
• Approach:
  – Two logically different entities: master process managing a work-queue, worker processes executing a task assigned to them by the master
  – Completion: explicit notification of master to worker processes typically required
• Can become very complicated for adaptive and recursive problems, where a worker can also ‘generate’ new tasks
Example Code using MPI

• Main function

```c
#define MASTERRANK 0
#define WORK_TAG 10
#define RES_TAG 11
#define NO_WORK_LEFT_TAG 12

int main ( int argc, char ** argv ) {
    int rank;
    MPI_Init ( &argc, &argv );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    if ( rank == MASTERRANK ) {
        master();
    } else {
        worker();
    }
    MPI_Finalize ();
    return (0);
}
```

Example Code using MPI

• Worker

```c
int worker ( void ) {
    int done=0; /* condition set to false */
    MPI_Status status;
    while ( !done ) {
        MPI_Recv ( &work, maxcnt, MPI_DOUBLE, MASTERRANK, 
                   MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if ( status.MPI_TAG == NO_WORK_LEFT_TAG ) {
            done = 1; /* condition set to true */
        } else {
            result = do_calculations ( work );
            MPI_Send ( &result, rescnt, MPI_DOUBLE, MASTERRANK, 
                       RES_TAG, MPI_COMM_WORLD );
        }
    }
    return (0);
}
```
Example Code using MPI

• Master part I

```c
int master ( void )
{
    int done = 0;

    /* distribute initial work */
    for ( proc = 0; proc < maxworkers; proc++ ) {
        next = get_next_work ();
        MPI_Send ( &next, xx, MPI_DOUBLE, proc, WORK_TAG,
                   MPI_COMM_WORLD);
        master_work_as_assigned ( proc, next );
    }

    while ( done < maxworkers ) {
        MPI_Recv ( &delctares, xy, MPI_DOUBLE, MPI_ANY_SOURCE,
                   PRO_TAG, comm, ampstatus);
        proc = status.MPI_SOURCE;
        store_work_result ( proc, delctares );
        next = get_next_work ();
        MPI_Recv ( &next, xx, MPI_DOUBLE, proc, WORK_TAG,
                   MPI_COMM_WORLD);
    }

    return ( 0 );
}
```

Example Code using MPI

• Master part II

```c
if ( next != NO_WORK_LEFT ) {
    MPI_Send ( &next, xx, MPI_DOUBLE, proc, WORK_TAG,
              MPI_COMM_WORLD );
} else {
    MPI_Send ( &next, 0, MPI_DOUBLE, NO_WORK_LEFT_TAG,
              MPI_COMM_WORLD);
    done ++;
}
/* end while loop */

return ( 0 );
```
Master-Worker Pattern

- Master/worker pattern works well, if a master has sufficient worker processes
- Master process can become a bottleneck if tasks are too small and number of worker processes is very large

Loop Parallelism Pattern

- In many scientific applications, the most compute intensive part is organized in a large loop
- Splitting the loop execution onto different processes is a straightforward parallelization, if the internal structure (=dependencies) allow that
- Most applications of the loop parallelism pattern rely on OpenMP
- Especially good when code cannot be massively restructured
Loop Parallelism: OpenMP Example

• Numerical integration

```c
#include <stdio.h>
#include "omp.h"

int main ( int argc, char **argv )
{
    int i, num_steps=100000;
    double x, xn, pi, step, sum=0.0;
    step = (b-a)/num_steps;
    #pragma omp parallel for private(x,xn) reduction(+:sum)
    for ( i=0; i<num_steps; i++ )
    {
        x = i * step;
        xn = (i+1) * step;
        sum = sum + 0.5*(x-n)*f(x)+f(xn);
    }
    return (0);
}
```

Fork/Join Pattern

• Useful if the number of concurrent tasks varies during execution
  – Tasks are created dynamically (= forked)
  – Tasks are terminated when done (= join with parents)
Fork/Join Pattern

- Can be useful for divide and conquer algorithms
- Often used with OpenMP
  - Can be used with MPI – 2 dynamic process management as well
- Creating and terminating processes/threads has a significant overhead

Reduction Pattern

- Concurrently executing processes or threads cooperate
- A collection of data items is reduced to a single item by repeatedly combining them pairwise with a binary operator
- Exploit concurrency in reduction operation

Serial reduction (computing sum of a[0] through a[3])
Three-Based Reduction

- n steps for $2^n$ units of execution
- When reduction operator is associative
- Especially attractive when only one task needs result

Recursive-Doubling Reduction

- n steps for $2^n$ units of execution
- If all units of execution need the result of the reduction
Advantages

• Better than tree-based approach with broadcast
  – Each units of execution has a copy of the reduced value at the end of n steps
  – In tree-based approach with broadcast to send the result to all the processors:
    • In recursive approach reduction takes n steps
    • Broadcast cannot begin until reduction is complete
    • Broadcast takes n steps (architecture dependent)
    • O(n) vs. O(2n)

Summary:
Algorithm vs Supporting Space

• Patterns can be hierarchically composed so that a program uses more than one pattern

<table>
<thead>
<tr>
<th></th>
<th>Task parallelism</th>
<th>Divide and conquer</th>
<th>Geometric decomposition</th>
<th>Recursive data</th>
<th>Pipeline</th>
<th>Event-based coordination</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>★★★★</td>
<td>★★</td>
<td>★★★★</td>
<td>★</td>
<td>★★★</td>
<td>★★</td>
</tr>
<tr>
<td>Loop Parallelism</td>
<td>★★★★</td>
<td>★★</td>
<td>★★</td>
<td>★</td>
<td>★★★</td>
<td>★★</td>
</tr>
<tr>
<td>Master/Worker</td>
<td>★★★★</td>
<td>★★</td>
<td>★★</td>
<td>★</td>
<td>★★★</td>
<td>★★</td>
</tr>
<tr>
<td>Fork/Join</td>
<td>★★</td>
<td>★★★★</td>
<td>★★</td>
<td>★★★★</td>
<td>★★★★</td>
<td>★★★★</td>
</tr>
</tbody>
</table>