Design Patterns for Parallel Programming II
Recap: Common Steps to Parallelization

1. **Decomposition**: Sequential computation of tasks is divided into smaller, manageable parts.
2. **Assignment**: Tasks are assigned to individual processors.
3. **Orchestration**: The coordination and communication between processors to ensure parallel execution.
4. **Mapping**: Assignment of execution units to processors.

The diagram illustrates the process with tasks being assigned to processors, showing the steps from sequential computation to parallel execution.
Recap: Decomposing for Concurrency

MPEG Decoder

- Task decomposition
  - Parallelism in the application
- Data decomposition
  - Same computation many data
- Pipeline decomposition
  - Data assembly lines
  - Producer-consumer chains
Dependence Analysis

- Given two tasks how to determine if they can safely run in parallel?
Bernstein’s Condition

- $R_i$: set of memory locations read (input) by task $T_i$
- $W_j$: set of memory locations written (output) by task $T_j$

Two tasks $T_1$ and $T_2$ are parallel if
- input to $T_1$ is not part of output from $T_2$
- input to $T_2$ is not part of output from $T_1$
- outputs from $T_1$ and $T_2$ do not overlap
Example

\[ T_1 \]
\[ a = x + y \]

\[ T_2 \]
\[ b = x + z \]

\[ R_1 = \{ x, y \} \]
\[ W_1 = \{ a \} \]

\[ R_2 = \{ x, z \} \]
\[ W_2 = \{ b \} \]

\[ R_1 \cap W_2 = \phi \]
\[ R_2 \cap W_1 = \phi \]
\[ W_1 \cap W_2 = \phi \]
Patterns for Parallelizing Programs

4 Design Spaces

**Algorithm Expression**
- Finding Concurrency
  - Expose concurrent tasks
- Algorithm Structure
  - Map tasks to units of execution to exploit parallel architecture

**Software Construction**
- Supporting Structures
  - Code and data structuring patterns
- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs

---

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
Organize by Tasks?

Recursive?

- yes → Divide and Conquer
- no → Task Parallelism
Task Parallelism

- Ray tracing
  - Computation for each ray is a separate and independent

- Molecular dynamics
  - Non-bonded force calculations, some dependencies

- Common factors
  - Tasks are associated with iterations of a loop
  - Tasks largely known at the start of the computation
  - All tasks may not need to complete to arrive at a solution
Divide and Conquer

- For recursive programs: divide and conquer
  - Subproblems may not be uniform
  - May require dynamic load balancing
Organize by Data?

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

Recursive?

yes → Recursive Data

no → Geometric Decomposition
Geometric Decomposition

- Gravitational body simulator
  - Calculate force between pairs of objects and update accelerations

```c
VEC3D acc[NUM_BODIES] = 0;

for (i = 0; i < NUM_BODIES - 1; i++) {
    for (j = i + 1; j < NUM_BODIES; j++) {
        // Displacement vector
        VEC3D d = pos[j] - pos[i];
        // Force
        t = 1 / sqr(length(d));
        // Components of force along displacement
        d = t * (d / length(d));

        acc[i] += d * mass[j];
        acc[j] += -d * mass[i];
    }
}
```
Recursive Data

- Computation on a list, tree, or graph
  - Often appears the only way to solve a problem is to sequentially move through the data structure

- There are however opportunities to reshape the operations in a way that exposes concurrency
Recursive Data 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. $O(n)$

Step 1

Step 2

Step 3
Work vs. Concurrency Tradeoff

- Parallel restructuring of find the root algorithm leads to $O(n \log n)$ work vs. $O(n)$ with sequential approach

- Most strategies based on this pattern similarly trade off increase in total work for decrease in execution time due to concurrency
Organize by Flow of Data?

- In some application domains, the flow of data imposes ordering on the tasks
  - Regular, one-way, mostly stable data flow
  - Irregular, dynamic, or unpredictable data flow

```
Regular?       yes
    
no

Event-based Coordination

Pipeline
```
Pipeline Throughput vs. Latency

- Amount of concurrency in a pipeline is limited by the number of stages

- Works best if the time to fill and drain the pipeline is small compared to overall running time

- Performance metric is usually the throughput
  - Rate at which data appear at the end of the pipeline per time unit (e.g., frames per second)

- Pipeline latency is important for real-time applications
  - Time interval from data input to pipeline, to data output
Event-Based Coordination

- In this pattern, interaction of tasks to process data can vary over unpredictable intervals.

- Deadlocks are likely for applications that use this pattern.
Supporting Structures
- SPMD
- Loop parallelism
- Master/Worker
- Fork/Join
Single Program Multiple Data: create a single source-code image that runs on each processor

- Initialize
- Obtain a unique identifier
- Run the same program each processor
  - Identifier and input data differentiate behavior
- Distribute data
- Finalize
Example: Parallel Numerical Integration

\[ f(x) = \frac{4.0}{1+x^2} \]

```c
static long num_steps = 100000;

void main()
{
    int i;
    double pi, x, step, sum = 0.0;

    step = 1.0 / (double) num_steps;
    for (i = 0; i < num_steps; i++)
    {
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x*x);
    }

    pi = step * sum;
    printf("Pi = %f\n", pi);
}
```
Computing Pi With Integration (MPI)

```c
static long num_steps = 100000;
void main(int argc, char* argv[]) {
    int i_start, i_end, i, myid, numprocs;
    double pi, mypi, x, step, sum = 0.0;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_BCAST(&num_steps, 1, MPI_INT, 0, MPI_COMM_WORLD);
    i_start = myid * (num_steps/numprocs);
    i_end = i_start + (num_steps/numprocs);
    step = 1.0 / (double) num_steps;
    for (i = i_start; i < i_end; i++) {
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x*x);
    }
    mypi = step * sum;
    MPI_REDUCE(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
        printf("Pi = \%f\n", pi);
    MPI_Finalize();
}
```
Block vs. Cyclic Work Distribution

```c
static long num_steps = 100000;

void main(int argc, char* argv[]) {
    int i_start, i_end, i, myid, numprocs;
    double pi, mypi, x, step, sum = 0.0;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    MPI_BCAST(&num_steps, 1, MPI_INT, 0, MPI_COMM_WORLD);
    i_start = my_id ∗ (num_steps/numprocs)
    i_end = i_start + (num_steps/numprocs)
    step = 1.0 / (double) num_steps;
    for (i = myid; i < num_steps; i += numprocs) {
        x = (i + 0.5) ∗ step
        sum = sum + 4.0 / (1.0 + x*x);
    }
    mypi = step ∗ sum;

    MPI_REDUCE(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,0,MPI_COMM_WORLD);

    if (myid == 0)
        printf("Pi = %f\n", pi);

    MPI_Finalize();
}
```
SPMD Challenges

● Split data correctly

● Correctly combine the results

● Achieve an even distribution of the work

● For programs that need dynamic load balancing, an alternative pattern is more suitable
Loop Parallelism Pattern

- Many programs are expressed using iterative constructs
  - Programming models like OpenMP provide directives to automatically assign loop iteration to execution units
  - Especially good when code cannot be massively restructured

```c
#pragma omp parallel for
for(i = 0; i < 12; i++)
    C[i] = A[i] + B[i];
```
Master/Worker Pattern

Independent Tasks

A  B  C  D  E

A  B  C  E

D

worker  worker  worker  worker

master
Master/Worker Pattern

- Particularly relevant for problems using task parallelism pattern where tasks have no dependencies
  - Embarrassingly parallel problems

- Main challenge in determining when the entire problem is complete
Fork/Join Pattern

● Tasks are created dynamically
  ■ Tasks can create more tasks

● Manages tasks according to their relationship

● Parent task creates new tasks (fork) then waits until they complete (join) before continuing on with the computation
Communication Patterns

- Point-to-point
- Broadcast
- Reduction
Serial Reduction

- When reduction operator is not associative
- Usually followed by a broadcast of result
Tree-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
Recursive-doubling Reduction

- 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
    - Reduction takes n steps
    - Broadcast cannot begin until reduction is complete
    - Broadcast takes n steps (architecture dependent)
    - $O(n)$ vs. $O(2n)$
Summary
### Algorithm Structure and Organization

<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><strong>SPMD</strong></td>
<td>★★★★★</td>
<td>★★★★</td>
<td>★★★★★</td>
<td>★★</td>
<td>★★★★</td>
<td>★★★</td>
</tr>
<tr>
<td><strong>Loop Parallelism</strong></td>
<td>★★★★★</td>
<td>★★★</td>
<td>★★★★</td>
<td>★★</td>
<td>★★★★</td>
<td>★★★</td>
</tr>
<tr>
<td><strong>Master/Worker</strong></td>
<td>★★★★★</td>
<td>★★★</td>
<td>★</td>
<td>★★</td>
<td>★★★★★</td>
<td>★★</td>
</tr>
<tr>
<td><strong>Fork/Join</strong></td>
<td>★★</td>
<td>★★★★★</td>
<td>★★</td>
<td>★★★★★</td>
<td>★★★★★</td>
<td>★★★★★</td>
</tr>
</tbody>
</table>

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