Design Patterns for Parallel Programming

Roberto Cavicchioli
roberto.cavicchioli@unimore.it
4 Common Steps to Creating a Parallel Program

Partitioning

Decomposition → Assignment → Orchestration → Mapping

Sequential computation → Tasks → Processes → Parallel program → Processors
Decomposition (Amdahl’s Law)

✓ Identify concurrency and decide at what level to exploit it

✓ Break up computation into tasks to be divided among processes
  – Tasks may become available dynamically
  – Number of tasks may vary with time

✓ Enough tasks to keep processors busy
  – Number of tasks available at a time is upper bound on achievable speedup
Specify mechanism to divide work among core
  – Balance work and reduce communication

Structured approaches usually work well
  – Code inspection or understanding of application
  – Well-known design patterns

As programmers, we worry about partitioning first
  – Independent of architecture or programming model
  – But complexity often affect decisions!
Orchestration and Mapping (Locality)

✓ Computation and communication concurrency

✓ Preserve locality of data

✓ Schedule tasks to satisfy dependences early
Parallel Programming by Pattern

✓ Provides a cookbook to systematically guide programmers
  – Decompose, Assign, Orchestrate, Map
  – Can lead to high quality solutions in some domains

✓ Provide common vocabulary to the programming community
  – Each pattern has a name, providing a vocabulary for discussing solutions

✓ Helps with software reusability, malleability, and modularity
  – Written in prescribed format to allow the reader to quickly understand the solution and its context

✓ Otherwise, too difficult for programmers, and software will not fully exploit parallel hardware
Patterns for Parallelizing Programs

4 Design Spaces

<table>
<thead>
<tr>
<th>Algorithm Expression</th>
<th>Software Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Finding Concurrency</td>
<td>✓ Supporting Structures</td>
</tr>
<tr>
<td>- Expose concurrent tasks</td>
<td>- Code and data structuring patterns</td>
</tr>
<tr>
<td>✓ Algorithm structure</td>
<td>✓ Implementation Mechanisms</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>
Here’s my algorithm. Where’s the concurrency?
Here’s my algorithm. Where’s the concurrency?

- 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
Here’s my algorithm. Where’s the concurrency?

✅ Task decomposition
  - Parallelism in the application

✅ Data decomposition
  - Same computation is applied to small data chunks derived from large data set
Here’s my algorithm. Where’s the concurrency?

- **Task decomposition**
  - Parallelism in the application

- **Data decomposition**
  - Same computation many data

- **Pipeline decomposition**
  - Data assembly lines
  - Producer-consumer chains
Guidelines for Task Decomposition

✓ Algorithms start with a good understanding of the problem being solved

✓ Programs often naturally decompose into tasks
  – Two common decompositions are
    • Function calls and
    • Distinct loop iterations

✓ Easier to start with many tasks and later fuse them, rather than too few tasks and later try to split them
Guidelines for Task Decomposition

✓ Flexibility
  – Program design should afford flexibility in the number and size of tasks generated
    • Tasks should not be tied to a specific architecture
    • Fixed tasks vs. Parameterized tasks

✓ Efficiency
  – Tasks should have enough work to amortize the cost of creating and managing them
  – Tasks should be sufficiently independent so that managing dependencies doesn’t become the bottleneck

✓ Simplicity
  – The code has to remain readable and easy to understand, and debug
Guidelines for Data Decomposition

✓ Data decomposition is often implied by task decomposition

✓ Programmers need to address task and data decomposition to create a parallel program
  – Which decomposition to start with?

✓ Data decomposition is a good starting point when
  – Main computation is organized around manipulation of a large data structure
  – Similar operations are applied to different parts of the data structure
Common Data Decompositions

✓ Array data structures
  – Decomposition of arrays along rows, columns, blocks

✓ Recursive data structures
  – Example: decomposition of trees into sub-trees
Guidelines for Data Decomposition

✓ Flexibility
  – Size and number of data chunks should support a wide range of executions

✓ Efficiency
  – Data chunks should generate comparable amounts of work (for load balancing)

✓ Simplicity
  – Complex data compositions can get difficult to manage and debug
Data is flowing through a sequence of stages
- Assembly line is a good analogy

What’s a prime example of pipeline decomposition in computer architecture?
- Instruction pipeline in modern CPUs

What’s an example pipeline you may use in your UNIX shell?
- Pipes in UNIX: `cat foobar.c | grep bar | wc`

Other examples
- Signal processing
- Graphics
Re-engineering for Parallelism
**Reengineering for Parallelism**

- Parallel programs often start as sequential programs
  - Easier to write and debug
  - Legacy codes

- How to reengineer a sequential program for parallelism:
  - Survey the landscape
  - Pattern provides a list of questions to help assess existing code
  - Many are the same as in any reengineering project
  - Is program numerically well-behaved?

- Define the scope and get users acceptance
  - Required precision of results
  - Input range
  - Performance expectations
  - Feasibility (back of envelope calculations)
Define a testing protocol

Identify program hot spots: where is most of the time spent?
  – Look at code
  – Use **profiling** tools

Parallelization
  – Start with **hot spots** first
  – Make sequences of small changes, each followed by **testing**
  – Pattern provides guidance
Example: Molecular dynamics

✔ Simulate motion in large molecular system
  – Used for example to understand drug-protein interactions

✔ Forces
  – Bonded forces within a molecule
  – Long-range forces between atoms

✔ Naïve algorithm has \( n^2 \) interactions: not feasible

✔ Use cutoff method: only consider forces from neighbors that are “close enough”
Sequential Molecular Dynamics Simulator

// pseudo code
real[3,n] atoms
real[3,n] force
int [2,m] neighbors

function simulate(steps)
    for time = 1 to steps and for each atom
        Compute bonded forces
        Compute neighbors
        Compute long-range forces
        Update position
    end loop
end function
Finding Concurrency Design Space

- Decomposition Patterns
- Dependency Analysis Patterns
- Design Evaluation
Main computation is a loop over atoms

Suggests task decomposition
  - Task corresponds to a loop iteration
    - Update a single atom
  - Additional tasks
    - Calculate bonded forces
    - Calculate long range forces
    - Find neighbors
    - Update position

There is data shared between the tasks
Understand Data Dependences

- Bonded forces
- Neighbor list
- Long-range forces
- Update position
- Read
- Write
- Accumulate

next time step

atoms[3,n]
forces[2,n]
neighbors[2,m]
Evaluate Design

✓ What is the target architecture?
  – Shared memory, distributed memory, message passing, …

✓ Does data sharing have enough special properties (read only, accumulate, temporal constraints) that we can deal with dependences efficiently?

✓ If design seems OK, move to next design space
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

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

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

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

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

\[ R_1 \cap W_2 = \phi \]
\[ R_2 \cap W_1 = \phi \]
\[ W_1 \cap W_2 = \phi \]
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
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
- Recursive? (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
For recursive programs: divide and conquer

- Subproblems may not be uniform
- May require dynamic load balancing
Operations on a central data structure
- Arrays and linear data structures
- Recursive data structures

![Decision Diagram]

- 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 / sqrt(length(d));
    // Components of force along displacement
    d = t * (d / length(d));
    acc[i] += d * mass[j];
    acc[j] += -d * mass[i];
  }
}
```
✓ 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
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
  - $O(\log n) \text{ vs. } O(n)$
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.
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
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
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
**SPMD Pattern**

- 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
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);
}
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();}

WARNING!! Block vs Cyclic work distribution
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
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
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
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
n steps for $2^n$ units of execution

If all units of execution need the result of the reduction
Better than tree-based approach with broadcast

- Each unit 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. \( O(2n) \)
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>
We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their own set of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.
The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.