Introduction to Parallel Computing

Task decomposition and mapping

Alexandre David
Overview

- Introduction to parallel algorithms
- Decomposition techniques
- Task interactions
- Load balancing
Introduction

- Parallel algorithms have the added dimension of *concurrency*.
- Typical tasks:
  - Identify concurrent works.
  - Map them to processors.
  - Distribute inputs, outputs, and other data.
  - Manage shared resources.
  - Synchronize the processors.

There are other courses specifically on concurrency. We won’t treat the problems proper to concurrency such as deadlocks, livelocks, theory on semaphores and synchronization. However, we will use them, and when needed, apply techniques to avoid problems like deadlocks.
Decomposing problems

- Decomposition into concurrent tasks.
  - No unique solution.
  - Different sizes.
  - Decomposition illustrated as a directed graph:
    - Nodes = tasks.
    - Edges = dependency.

Many solutions are often possible but few will yield good performance and be scalable. We have to consider the computational and storage resources needed to solve the problems.

Size of the tasks in the sense of the amount of work to do. Can be more, less, or unknown. Unknown in the case of a search algorithm is common.

Dependency: All the results from incoming edges are required for the tasks at the current node.

We will not consider tools for automatic decomposition. They work fairly well only for highly structured programs or options of programs.
Example: Matrix * Vector

N tasks, 1 task/row:

Task dependency graph?
Example: database query processing

\[
\text{MODEL} = \text{``CIVIC'' AND YEAR = 2001 AND (COLOR = \text{``GREEN'' OR COLOR = \text{``WHITE}')}}
\]

<table>
<thead>
<tr>
<th>ID#</th>
<th>Model</th>
<th>Year</th>
<th>Color</th>
<th>Dealer</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>4523</td>
<td>Civic</td>
<td>2002</td>
<td>Blue</td>
<td>MN</td>
<td>$18,000</td>
</tr>
<tr>
<td>3476</td>
<td>Corola</td>
<td>1999</td>
<td>White</td>
<td>IL</td>
<td>$15,000</td>
</tr>
<tr>
<td>7623</td>
<td>Camry</td>
<td>2001</td>
<td>Green</td>
<td>NY</td>
<td>$21,000</td>
</tr>
<tr>
<td>9834</td>
<td>Prius</td>
<td>2001</td>
<td>Green</td>
<td>CA</td>
<td>$18,000</td>
</tr>
<tr>
<td>6734</td>
<td>Civic</td>
<td>2001</td>
<td>White</td>
<td>OR</td>
<td>$17,000</td>
</tr>
<tr>
<td>5342</td>
<td>Altima</td>
<td>2001</td>
<td>Green</td>
<td>FL</td>
<td>$19,000</td>
</tr>
<tr>
<td>3845</td>
<td>Maxima</td>
<td>2001</td>
<td>Blue</td>
<td>NY</td>
<td>$22,000</td>
</tr>
<tr>
<td>8354</td>
<td>Accord</td>
<td>2000</td>
<td>Green</td>
<td>VT</td>
<td>$18,000</td>
</tr>
<tr>
<td>4395</td>
<td>Civic</td>
<td>2001</td>
<td>Red</td>
<td>CA</td>
<td>$17,000</td>
</tr>
<tr>
<td>7352</td>
<td>Civic</td>
<td>2002</td>
<td>Red</td>
<td>WA</td>
<td>$18,000</td>
</tr>
</tbody>
</table>

Table 3.1 A database storing information about used vehicles.

The question is: How to decompose this into concurrent tasks? Different tasks may generate intermediate results that will be used by other tasks.
How much concurrency do we have here? How many processors to use? Is it optimal?
Is it better or worse? Why?
Granularity

- **Number** and size of tasks.
  - Fine-grained: many small tasks.
  - Coarse-grained: few large tasks.

- Related: *degree of concurrency.*
  (Nb. of tasks executable in parallel).
  - Maximal degree of concurrency.
  - Average degree of concurrency.

- Previous matrix*vector fine-grained.
- Database example coarse grained.

Degree of concurrency: Number of tasks that can be executed in parallel.

Average degree of concurrency is a more useful measure.

**Assume** that the tasks in the previous database examples have the same granularity. What’s their average degrees of concurrency? $7/3 = 2.33$ and $7/4 = 1.75$.

Common sense: Increasing the granularity of decomposition and utilizing the resulting concurrency to perform more tasks in parallel increases performance. However, there is a limit to granularity due to the nature of the problem itself.
Coarser Matrix * Vector

N tasks, 3 task/row:
Granularity

- Average degree of concurrency if we take into account varying amount of work?
- Critical path = longest directed path between any start & finish nodes.
- Critical path length = sum of the weights of nodes along this path.
- Average degree of concurrency = total amount of work / critical path length.

Weights on nodes denote the amount of work to be done on these nodes. Longest path → shortest time needed to execute in parallel.
Database example

Critical path (3).

Critical path length = 27.
Av. deg. of concurrency = 63/27.

Critical path (4).

Critical path length = 34.
Av. deg. of conc. = 64/34.
Exercise

(a) • Maximum degree of concurrency.
(b) • Critical path length.
(c) • Maximum possible speedup.
(d) • Minimum number of processes to reach this speedup.
• Maximum speedup if we limit the processes to 2, 4, and 8.
Interaction between tasks

- Tasks often share data.
- Task interaction graph:
  - Nodes = tasks.
  - Edges = interaction.
  - Optional weights.
  - Task dependency graph is a sub-graph of the task interaction graph.

Another important factor is interaction between tasks on different processors. Share data implies synchronization protocols (mutual exclusion, etc) to ensure consistency.

Edges generally undirected. When directed edges are used, they show the direction of the flow of data (and the flow is unidirectional).

Dependency between tasks implies interaction between them.
Processes and mapping

- Tasks run on processors.
- Process: processing agent executing the tasks. Not exactly like in your OS course.
- Mapping = assignment of tasks to processes.
- API exposes processes and binding to processors not always controlled.
  - Scheduling of threads is not controlled.
  - What makes a good mapping?

Here we are not talking directly on the mapping to processors. A processor can execute two processes.

Good mapping:
- Maximize concurrency by mapping independent tasks to different processes.
- Minimize interaction by mapping interacting tasks on the same process.

Can be conflicting, good trade-off is the key to performance.

Decomposition determines degree of concurrency.
Mapping determines how much concurrency is utilized and how efficiently.
Notice that the mapping keeps one process from the previous stage because of dependency: We can avoid interaction by keeping the same process.
Processes vs. processors

- Processes = logical computing agent.
- Processor = hardware computational unit.
- In general 1-1 correspondence but this model gives better abstraction.
- Useful for hardware supporting multiple programming paradigms.

Now remains the question:
How do you decompose?

Example of hybrid hardware: cluster of MP machines. Each node has shared memory and communicates with other nodes via MPI.
1. Decompose and map to processes for MPI.
2. Decompose again but suitable for shared memory.
Decomposition techniques

- Recursive decomposition.
  - Divide-and-conquer.
- Data decomposition.
  - Large data structure.
- Exploratory decomposition.
  - Search algorithms.
- Speculative decomposition.
  - Dependent choices in computations.
Recursive decomposition

- Problem solvable by divide-and-conquer:
  - Decompose into sub-problems.
    - Do it recursively.
  - Combine the sub-solutions.
    - Do it recursively.
  - Concurrency: The sub-problems are solved in parallel.

Small problem is to start and finish: with one process only.
Recall on the quicksort algorithm:

• Choose a pivot.
• Partition the array.
• Recursive call.
• Combine result: nothing to do.
Minimal number

![Diagram of minimal number]

**Figure 3.9** The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers.
Data decomposition

- 2 steps:
  - Partition the data.
  - Induce partition into tasks.
- How to partition data?
- Partition output data:
  - Independent “sub-outputs”.
- Partition input data:
  - Local computations, followed by combination.

- 1-D, 2-D, 3-D block decomposition.

Partitioning of input data is a bit similar to divide-and-conquer.
Matrix multiplication by block

\[
\begin{pmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{pmatrix}
\begin{pmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{pmatrix}
\]

(a)

(b)

Figure 3.10  (a) Partitioning of input and output matrices into $2 \times 2$ submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a).

We can partition further for the tasks. Notice the dependency between tasks. What is the task dependency graph?
Intermediate data partitioning

\[
\begin{array}{c}
\text{A}_{1,1} \\
\text{A}_{2,1} \\
\text{A}_{1,2} \\
\text{A}_{2,2}
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
\text{B}_{1,1} \\
\text{B}_{2,1} \\
\text{B}_{1,2} \\
\text{B}_{2,2}
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
\text{D}_{1,1} \\
\text{D}_{2,1} \\
\text{D}_{1,2} \\
\text{D}_{2,2}
\end{array}
\]

\[
\begin{array}{c}
\text{C}_{1,1} \\
\text{C}_{1,2} \\
\text{C}_{2,1} \\
\text{C}_{2,2}
\end{array}
\]

Linear combination of the intermediate results.
Owner-compute rule

- Process assigned to some data
  - is responsible for all computations associated with it.
- Input data decomposition:
  - All computations done on the (partitioned) input data are done by the process.
- Output data decomposition:
  - All computations for the (partitioned) output data are done by the process.

Important rule, very useful, in particular stresses locality.
Suitable for search algorithms. Partition the search space into smaller parts and search in parallel. We search the solution by a tree search technique.
Performance anomalies

**Work depends on the order of the search!**

![Diagram showing two search trees with different total work calculations.](image)

**Figure 3.19** An illustration of anomalous speedups resulting from exploratory decomposition.
Speculative decomposition

- Dependencies between tasks are not known a-priori.
  - How to identify independent tasks?
  - Conservative approach: identify tasks that are guaranteed to be independent.
  - Optimistic approach: schedule tasks even if we are not sure – may roll-back later.

Not possible to identify independent tasks in advance. Conservative approaches may yield limited concurrency. Optimistic approach = speculative. Optimistic approach is similar to branch prediction algorithms in processors.
So far...

- Decomposition techniques.
  - Identify tasks.
  - Analyze with task dependency & interaction graphs.
  - Map tasks to processes.
- Now properties of tasks that affect a good mapping.
  - Task generation, size of tasks, and size of data.
Task generation

- Static task generation.
  - Tasks are known beforehand.
  - Apply to well-structured problems.

- Dynamic task generation.
  - Tasks generated on-the-fly.
  - Tasks & task dependency graph not available beforehand.

The well-structured problem can typically be decomposed using data or recursive decomposition techniques.

Dynamic tasks generation: Exploratory or speculative decomposition techniques are generally used, but not always. Example: quicksort.
**Task sizes**

- Relative amount of time for completion.
  - Uniform – same size for all tasks.
    - Matrix multiplication.
  - Non-uniform.
    - Optimization & search problems.

Typically the size of non-uniform tasks is difficult to evaluate beforehand.
Size of data associated with tasks

- Important because of locality reasons.
- Different types of data with different sizes
  - Input/output/intermediate data.
  - Size of context – cheap or expensive communication with other tasks.
Characteristics of task interactions

- Static interactions.
  - Tasks and interactions known beforehand.
  - And interaction at pre-determined times.
- Dynamic interactions.
  - Timing of interaction unknown.
  - Or set of tasks not known in advance.
- Regular interactions.
  - The interaction graph follows a pattern.
- Irregular interactions.
  - No pattern.

Static vs. dynamic.
Static or dynamic interaction pattern.
Dynamic harder to code, more difficult for MPI.
Example: Image Dithering

The color of each pixel is determined as the weighted average of its original value and the values of the neighboring pixels. Decompose into regions, 1 task/region. Pattern is a 2-D mesh. Regular pattern.
Characteristics of task interactions

- Data sharing interactions:
  - Read-only interactions.
    - Read only data associated with other tasks.
  - Read-write interactions.
    - Read & modify data of other tasks.

Read-only vs. read-write.
Read-only example: matrix multiplication (share input). Read-write example: 15-puzzle with shared priority list of states to be explored; Priority given by some heuristic to evaluate the distance to the goal.
Characteristics of task interactions

- One-way interactions.
  - Only one task initiates and completes the communication \textit{without} interrupting the other one.
- Two-way interactions.
  - Producer – consumer model.

One-way vs. two-way.

One-way more difficult with MPI since MPI has an explicit send & receive set of calls. Conversion one-way to two-way with polling or another thread waiting for communication.
Mapping techniques for load balancing

- Map tasks onto processes.
- Goal: minimize overheads.
  - Communication.
  - Idling.
- Uneven load distribution may cause idling.
  - Constraints from task dependency → wait for other tasks.

Minimizing communication may contradict minimizing idling. Put tasks that communicate with each other on the same process but may unbalance the load -> distribute them but increase communication.

Load balancing is not enough to minimize idling.
Example

![Diagram showing two mappings of a hypothetical decomposition with a synchronization.](image)

**Figure 3.23** Two mappings of a hypothetical decomposition with a synchronization.

Global balancing OK but due to task dependency P4 is idling.
Mapping techniques

- Static mapping.
  - NP-complete problem for non-uniform tasks.
  - Large data compared to computation.
- Dynamic mapping.
  - Dynamically generated tasks.
  - Task size unknown.

Even static mapping may be difficult: The problem of obtaining an optimal mapping is an NP-complete problem for non-uniform tasks. In practice simple heuristics provide good mappings.

Cost of moving data may out-weight the advantages of dynamic mapping.

In shared address space dynamic mapping may work well even with large data, but be careful with the underlying architecture (NUMA/UMA) because data may be moved physically.
Schemes for static mapping

- Mappings based on data partitioning.
- Mappings based on task graph partitioning.
- Hybrid mappings.
Array distribution scheme

- Combine with “owner computes” rule to partition into sub-tasks.

**row-wise distribution**

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
<th>P₆</th>
</tr>
</thead>
</table>

**column-wise distribution**

| P₀ | P₁ | P₂ | P₃ | P₄ | P₅ | P₆ |

1-D block distribution scheme.

Data partitioning mapping.
Mapping data = mapping tasks.
Simple block-distribution.
Block distribution cont.

<table>
<thead>
<tr>
<th></th>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th></th>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
<th>$P_6$</th>
<th>$P_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_4$</td>
<td>$P_5$</td>
<td>$P_6$</td>
<td>$P_7$</td>
<td></td>
<td>$P_8$</td>
<td>$P_9$</td>
<td>$P_{10}$</td>
<td>$P_{11}$</td>
<td></td>
<td>$P_{12}$</td>
<td>$P_{13}$</td>
<td>$P_{14}$</td>
<td>$P_{15}$</td>
</tr>
</tbody>
</table>

(a) (b)

*Generalize to higher dimensions: 4x4, 2x8.*
Example: Matrix*Matrix

- Partition output of $C=A*B$.
- Each entry needs the same amount of computation.
- Blocks on 1 or 2 dimensions.
- Different data sharing patterns.
- *Higher dimensional* distributions
  - means we can use *more processes*.
  - sometimes *reduces* interaction.

In the case of matrix $n*n$ multiplication, 1-D $\rightarrow n$ processes at most, 2-D $n^2$ processes at most.
O(n²/sqrt(p)) vs. O(n²) shared data.
Imbalance problem

- If the amount of *computation* associated with data *varies* a lot then *block decomposition* leads to *imbalances*.
- Example: LU factorization (or Gaussian elimination).

Exercise on LU-decomposition.
LU factorization

- Non singular square matrix $A$ (invertible).
- $A = L \times U$.
- Useful for solving linear equations.

\[ A = L \times U \]
LU factorization

In practice we work on $A$.
LU algorithm

Proc LU(A) begin
for k := 1 to n-1 do
    for j := k+1 to n do
    endfor
endfor
for j := k+1 to n do
    for i := k+1 to n do
    endfor
endfor

Normalize L
U[k,j] := A[k,j]/L[k,k]


\( L[i,k] \) \( U[k,j] \)
Load imbalance for individual tasks. Load imbalance from dependencies.
Cyclic and block-cyclic distributions

- Idea:
  - Partition an array into many more blocks than available processes.
  - Assign partitions (tasks) to processes in a round-robin manner.
  - Each process gets several non adjacent blocks.
Block-Cyclic Distributions

a) Partition 16x16 into 2*4 groups of 2 rows. 
\( \alpha p \) groups of \( n/\alpha p \) rows.

b) Partition 16x16 into square blocks of size 4*4 distributed on 2*2 processes. 
\( \alpha^2 p \) groups of \( n/\alpha^2 p \) squares.

Reduce the amount of idling because all processes have a sampling of tasks from all parts of the matrix.

But lack of locality may result in performance penalties + leads to high degree of interaction. Good value for \( \alpha \) to find a compromise.
Randomized distributions

Irregular distribution with regular mapping!
Not good.
**1-D randomized distribution**

\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] \]

\[
\text{random}(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]
\]

\[
\text{mapping} = \begin{array}{cccc}
8 & 2 & 6 & 0 \\
3 & 7 & 11 & 1 \\
9 & 5 & 4 & 10
\end{array}
\]

\[
P_0 \quad P_1 \quad P_2 \quad P_3
\]

**Figure 3.32** A one-dimensional randomized block mapping of 12 blocks onto four processes (i.e., \( \alpha = 3 \)).
2-D randomized distribution

2-D block random distribution.

Block mapping.
Graph partitioning

- For sparse data structures and data dependent interaction patterns.
  - Numerical simulations. Discretize the problem and represent it as a mesh.
  - Sparse matrix: assign equal number of nodes to processes & minimize interaction.
  - Example: simulation of dispersion of a water contaminant in Lake Superior.
Discretization

Figure 3.34  A mesh used to model Lake Superior.
Partitioning Lake Superior

Random partitioning. Partitioning with minimum edge cut.

Finding an exact optimal partitioning is an NP-complete problem.

Minimum edge cut from a graph point of view. Keep locality of data with processes to minimize interaction.
Mappings based on task partitioning

- Partition the task dependency graph.
  - Good when static task dependency graph with known task sizes.

Mapping on 8 processes.

Determining an optimal mapping is NP-complete. Good heuristics for structured graphs.

Binary tree task dependency graph: occurs in recursive decompositions as seen before. The mapping minimizes interaction. There is idling but it is inherent to the task dependency graph, we do not add more.

This example good on a hypercube. See why?
Hierarchical mappings

- Combine several mapping techniques in a structured (hierarchical) way.
- Task mapping of a binary tree (quicksort) does not use all processors.
  - Mapping based on task dependency graph (hierarchy) & block.
Schemes for dynamic mapping

- Centralized Schemes.
  - Master manages pool of tasks.
  - Slaves obtain work.
  - Limited scalability.
- Distributed Schemes.
  - Processes *exchange tasks* to balance work.
  - Not simple, many issues.

Centralized schemes are easy to implement but present an obvious bottleneck (the master).

**Self-scheduling:** slaves pick up work to do whenever they are idle.

Bottleneck: tasks of size M, it takes t to assign work to a slave → at most M/t processes can be kept busy.

**Chunk-scheduling:** a way to reduce bottlenecks by getting a group of tasks. Problem for load imbalances.

Distributed schemes more difficult to implement.

How do you choose sender & receiver? i.e. if A is overloaded, which process gets something?

Initiate transfer by sender or receiver? i.e. A overloaded sends work or B idle requests work?

How much work to transfer?

When to transfer?

Answers are application specific.
Minimizing interaction overheads

• Maximize data locality.
  • Minimize volume of data-exchange.
  • Minimize frequency of interactions.

• Minimize contention and hot spots.
  • Share a link, same memory block, etc…
  • Re-design original algorithm to change the interaction pattern.

Minimize volume of exchange $\rightarrow$ maximize temporal locality. Use higher dimensional distributions, like in the matrix multiplication example. We can store intermediate results and update global results less often.

Minimize frequency of interactions $\rightarrow$ maximize spatial locality.

Related to the previously seen cost model for communications.

Changing the interaction pattern: For the matrix multiplication example, the sum is commutative so we can re-order the operations modulo $\sqrt{p}$ to remove contention.
Minimizing interaction overheads

- Overlapping computations with interactions – to reduce idling.
  - Initiate interactions in advance.
  - Non-blocking communications.
  - Multi-threading.
- Replicating data or computation.
- Group communication instead of point to point.
- Overlapping interactions.

Replication is useful when the cost of interaction is greater than replicating the computation. Replicating data is like caching, good for read-only accesses. Processing power is cheap, memory access is expensive – also apply at larger scale with communicating processes.

Collective communication such as broadcast. However, depending on the communication pattern, a custom collective communication may be better.