Parallel Programming Patterns

Salvatore Orlando
Recurrent parallel patterns

- Each pattern, or forms of parallelism, corresponds to:
  - specific structures of the parallel tasks
  - specific techniques of partitioning/allocating data
  - specific structures of communication

- We will study the most common ones, by instantiating them on a message passing model

- Before illustrating patterns, some preliminaries about modeling parallel programs
Preliminaries: Decomposition, Tasks, and Dependency Graphs

- The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently.

- A given problem may be decomposed into tasks in many different ways.
  - Tasks may be of same, different, or even indeterminate sizes/granularities.

- Decomposition modeled/illustrated in the form of Task Dependency Graph (TDG):
  - Directed Acyclic Graph
  - Nodes = Tasks
  - Directed Edges = Control Dependency among tasks
  - Node Labels = computational size/weight of the task.
Ex.: Multiplying a Dense Matrix with a Vector

Computation of each element of output vector $y$ is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into $n$ tasks. The figure highlights the portion of the matrix and vector accessed by Task 1.

**Observations:** While tasks share data (namely, the vector $b$), they do not have any dependencies - i.e., no task needs to wait for the (partial) completion of any other. All tasks are of the same size in terms of number of operations. *Is this the maximum number of tasks we could decompose this problem into?*
Example: Database Query Processing

Consider the execution of the query:

MODEL = "CIVIC" AND YEAR = 2001 AND
(COLOR = "GREEN" OR COLOR = "WHITE")

on the following database:

<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>Corolla</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>
Example: Database Query Processing

- The execution of the query can be divided into subtasks in various ways. Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.

\[
\text{(MODEL = "CIVIC" AND YEAR = 2001) AND (COLOR = "GREEN" OR COLOR = "WHITE")}
\]
Example: Database Query Processing

- Note that the same problem can be decomposed into subtasks in other ways as well. This because the AND operation is associative.

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

- The number of tasks into which a problem is decomposed determines its granularity.
- Decomposition into a **large number** of tasks results in **fine-grained decomposition**. Decomposition into a **small number** of tasks results in a **coarse grained decomposition**.

\[ \text{Task} = \text{Computation of 3 elements of } y \]

\[ n=4 \text{ independent tasks} \]

*In figure, note Task 1*
Degree of concurrency

- The number of tasks that can be executed in parallel is the *degree of concurrency* of a decomposition.
- Since the number of tasks that can be executed in parallel may change over program execution, the *maximum degree of concurrency* is the maximum number of such tasks at any point during execution.
  - *What is the maximum degree of concurrency of the database query examples?*
- The *average degree of concurrency* is the average number of tasks that can be processed in parallel over the execution of the program.
  - *Assuming that each tasks in the database example takes identical processing time, and we have enough processors to execute independent task in parallel, what is the average degree of concurrency in each decomposition?*
  - If the average degree of concurrency is similar to the maximum, the parallel system is used very efficiently.
- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.
Degree of concurrency
Critical path

• A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.

• The longest such path determines the shortest time in which the program can be executed in parallel
  – measured in terms of number of tasks, or sum of the weights of the tasks involved

• The length of the longest path in a task dependency graph is called the critical path length
  – It corresponds to the minimum execution time of a parallel program
Critical path length

- Task dependency graphs of the two database query
  - Which are the critical paths?

- If each task takes 10 time units, what is the shortest parallel execution time for each decomposition? How many processors are needed in each case to achieve this minimum parallel execution time? What is the maximum degree of concurrency?
Limits on Parallel Performance

• It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity.

• There is an inherent bound on how fine the granularity of a computation can be.
  – *For example, in the case of multiplying a dense matrix with a vector, there can be no more than \((n^2)\) concurrent tasks*
  – *i.e., all the multiplications performed in parallel*

• Concurrent tasks may also have to exchange data with other tasks. This results in communication overhead, which increases along with the parallelism increase (finer granularity).
  – *The tradeoff between the granularity of a decomposition and associated overheads often determines performance bounds.*
Task Interaction Graphs

- Subtasks generally exchange data with others in a decomposition.
  - For example, even in the trivial decomposition of the dense matrix-vector product, if the vector is not replicated across all tasks but partitioned, they will have to communicate elements of the vector.
- The graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a *task interaction graph*.

- Task interaction graph (TIG)
  - Nodes = tasks
  - Edges (usually indirect ones) = interaction or data exchange
  - Node labels = Computation weight of tasks
  - Edge labels = amount of data exchanged
Example of Task interaction graph (TIG)

- Matrix $\times$ Vector, where matrix $A$ is **sparse**
  - Unlike a dense matrix-vector product though, only non-zero elements of matrix $A$ participate in the computation

\[
\sum_{0 \leq j \leq 11, A[i, j] \neq 0} A[i, j] \cdot b[j].
\]

- Independent task = Computation of a single element of the result vector $y$
- To optimize the usage of memory, $b$ is **partitioned among the tasks**
  - $b[i]$ assigned to task $i$ This is NOT the optimal solution
- TIG : models the needed interactions/communications to gather all the missing elements of vector $b$
Task Interaction Graphs, Granularity, and Communication

In general, if the granularity of a decomposition is finer, the associated overhead increases.

**Example:** sparse matrix-vector product

Assume that each node takes unit time to process and each interaction (edge) causes an overhead of a unit time.

Thus **node 0** is an independent task that involves a (useful) computation of one time unit and overhead (communication) of three time units

⇒ **computation-to-communication ratio:** 1/3

Now, if we consider **nodes 0, 4, and 8** as one task ⇒ thus increasing **task granularity**

→ the task has useful computation totaling to **three** time units and communication corresponding to **four** time units (four edges). Clearly, this is a more favorable ratio than the former case (3/4)
Task, processes and mapping

• In general, the number of tasks in a decomposition exceeds the number of processing elements available.

• For this reason, a parallel algorithm must also provide a mapping of tasks to a few processes in order to **agglomerate tasks**, thus obtaining a **optimal granularity**

• This **mapping phase** is often called **agglomeration phase** in many textbook

**Note:** We refer to the mapping as being **from tasks to processes (virtual processors)**, as opposed to physical processors. We aggregate tasks into processes and rely on the system to map these processes to physical processors. We use the term ‘process’, not in the UNIX sense of a process, rather, simply as a collection of tasks and associated data.

*The tasks of MPI are actually processes exchanging messages ...*
Processes and Mapping

• Appropriate mapping of tasks to processes is critical to the parallel performance of an algorithm

• Mappings are determined starting from both task dependency and task interaction graphs (TDG and TIG)

• TDG can be used to ensure that work is equally spread across all processes at any point (minimum idling and optimal load balance)

• TIG can be used to make sure that processes need minimum interaction with other processes (minimum communication, and thus overheads)
Processes and Mapping

An appropriate mapping must minimize parallel execution time by:

- Mapping independent tasks to different processes
- Assigning tasks on critical path to processes as soon as they become available
- Minimizing interaction between processes by mapping tasks with dense interactions to the same process

Note: These criteria often conflict with each other. For example, a decomposition into one task (or no decomposition at all) minimizes interaction but does not result in a speedup at all! Can you think of other such conflicting cases?
Processes and Mapping: Example of the database query

- These mappings were obtained by viewing the dependency graph in terms of **levels** (no two nodes in a level have dependencies)
  - Tasks within a single level are assigned to different processes
  - Tasks on the critical path are assigned to the same process
Processes and Mapping: Example of the database query

![Diagram showing processes and mapping with tasks and P0, P1, P2, and P3 nodes at different times.](Image)
Data parallelism vs. task parallelism

• How do we decompose/partition a problem to obtain the parallel tasks?
  – we can decompose on either the data or the control

• Data parallelism
  – Perform the same operation onto different data items in parallel
  – The parallelism comes from a data decomposition (domain decomposition), and grows with the size of the data
  – Data parallelism facilitates very high speedups, and is usually very fine-grained (e.g., SIMD and SEE/vectorization)

• Task parallelism
  – Perform distinct computations - or tasks - at the same time. The parallelism comes from a control/functional decomposition.
Data parallelism

- **Sequential algorithm:**
  - function $F$ applied on a data structure $I$ to yield $O$: $O = F(I)$

- **Data Parallelism or Domain Decomposition**
  - Partition on data
  - $I = \{I_1, \ldots, I_n\}$
  - $O = \{O_1, \ldots, O_n\}$, where $O_j = F(I_j)$
Task parallelism

- Sequential algorithm:
  - function $F$ applied on a data structure $I$ to yield $O$: $O = F(I)$

- Task Parallelism or Functional Parallelism
  - Partition on the control
  - Pipeline (good only in presence of a stream of input data):
    - $O = F(I) = F_m(F_{m-1}(...F_1(I)))$
  - Task graph
    - $O = F(I) = F_3(F_1(I), F_2(I))$

- Combination of Task & Data Parallelism
Recurrent parallel algorithmic patterns
Embarrassingly Parallel

- Problems that can be solved with a large set of completely independent sub-tasks
  - Usually data-parallel
  - Completely Disconnected TDG
An example of implementation

- **Static partitioning** of the input data
- **N slaves** are created dynamically
- Each slave is assigned a partition of the input

- This pattern does not work if the assigned tasks are unbalanced
Mandelbrot set: Embarrassingly parallel

• For each point $c$ in the Complex Plane compute function $\text{color}(c)$ as follows:
  – The real and imaginary parts of $c$ correspond to its Cartesian coordinates
  – Iterative functions computed:
    \[ Z_{k+1} = Z_k^2 + c \]
    – $Z_0 = 0$
    – The function iterates until $|Z|>2$ or a user-defined maximum number of iterations is reached
    – The function returns the number of iterations $k$, which is mapped to a color

• The cost of function $\text{color}(c)$ is non uniform
Mandelbrot set

- We know that
  - $|Z_k|$ will always be smaller than 2
  - $|Z_k|$ will become larger than 2 and increase indefinitely
- If $|Z_k| > 2$ for some $k$, then the complex $c$ does not belong to the Mandelbrot set
Mandelbrot graphic representation

- On the right, the *black* points belong to the *Mandelbrot set*, and the *white* points not

- We **color** the points that do not belong to the *Mandelbrot set*
  - The used colors depend on the number of **iterations** $k$ that are needed to obtain $|Z_k| > 2$
Mandelbrot set algorithm

- We only compute the belonging/not belonging to the Mandelbrot set for a very limited and discrete number of complex numbers!
- Number of pixels on the display:  \( \text{disp\_width} \times \text{disp\_height} \)
- The corresponding complex plane is bounded:
  - abscisses: \([\text{real\_min} + \text{real\_max}]\)
  - ordinates: \([\text{imag\_min} + \text{imag\_max}]\)
- The complex plane must be discretized with respect to the number of pixels \(\text{displ\_width} \times \text{displ\_height}\)

```c
scale_real = (\text{real\_max} - \text{real\_min})/\text{disp\_width};
scale_imag = (\text{imag\_max} - \text{imag\_min})/\text{disp\_height};

for (x = 0; x < \text{disp\_width}; x++) /* screen coordinates x and y */
  for (y = 0; y < \text{disp\_height}; y++) {
    c.real = \text{real\_min} + ((\text{float})x * scale_real);
    c.imag = \text{imag\_min} + ((\text{float})y * scale_imag);
    color = cal\_pixel(c);
    display(x, y, color);
  }
```

Visualize pixel \((x,y)\) with the correct color
Mandelbrot set

Example:
- abscisses: $[-2 \div +2]$
- ordinates: $[-2 \div +2]$
Computing the Mandelbrot set in parallel

- We can apply a domain partition technique
  - data parallelism
  - embarrassingly parallel
  - fine grain parallelism: each parallel task is assigned the computation of a single pixel/point
  - coarser grain parallelism: we can assign groups of pixels to each task

- How long does it take each task?
  - The number of iterations $k$ is unknown !!
  - It can have very short or very long tasks
  - Potential Load Imbalance
Processor Farm (or Work Pool)

Three logical entities:
- **Emitters**: create job *statically or dynamically*
- **Workers**: ask for a job to execute
  - Requests can be issues when previous job has been completed
  - **Pre-fetching**: When the current job has been received, ask for the next one
- **Collectors**: gather the results of the computation
Processor Farm

- What is the right number of workers?

- Given
  - $T_{\text{work}}$, time to complete a task
  - $T_{\text{comm}}$, communication time (send task rec. results)

- The Communication Bandwidth is $B_{\text{comm}} = 1/T_{\text{comm}}$

- The Computation Bandwidth is $B_{\text{comp}} = n*1/T_{\text{work}}$
  where $n$ is the number of workers

- The optimal $n$ is when $B_{\text{comm}} = B_{\text{comp}}$:
  $$1/T_{\text{comm}} = n * 1/T_{\text{work}}$$
  $$n = T_{\text{work}}/T_{\text{comm}}$$

- In addition:
  - We could use *pre-fetching* to overlap communication and computation
  - Use a single master to emit jobs and collect results, and interpret a result as a request for the next job.
Work pool per Mandelbrot

Dynamic Task Assignment
Work Pool/Processor Farms

- Emission/collection activities performed by the same process (master)
  - messages to communicate results are also interpreted as a request for a new job
Pseudo-code of processor farm

• Suppose we have \( W \) workers, with \( W \) smaller then the number of rows \( n_{\text{rows}} \) in the plane

• Use a medium granularity:
  – Assign \( \text{rows} \) rather than points
  – Smaller communication costs:
    • Send \( \text{row id} \) to the workers (not the coordinates of all points)
    • Receive one message per row from workers (not distinct messages per point)

• “Synchronization” before the end of the application
• The variable \( \text{count} \) is for the number of rows currently assigned
• No pre-fetching
Master

count = 0; /* counter for termination*/
row = 0; /* row being sent */
for (k = 0; k < procno; k++) { /* assuming procno<disp_height */
    send(&row, P_k, data_tag); /* send initial row to process */
    count++;
    row++;
}

do {
    recv (&slave, &r, color, P_ANY, result_tag);
    count--; /* reduce count as rows received */
    if (row < disp_height) {
        send (&row, P_slave, data_tag); /* send next row */
        row++;
        count++;
    } else
        send (&row, P_slave, terminator_tag); /* terminate */
    rows_recv++;
    display (r, color); /* display row */
} while (count > 0);
Pseudo-Code: Slave

Slave

recv(y, P\text{\texttt{master}}, \text{ANYTAG}, \text{source\_tag}); /* receive 1st row to compute */
while (source\_tag == data\_tag) {
    c.imag = imag\_min + ((\text{float}) y * scale\_imag);
    for (x = 0; x < disp\_width; x++) { /* compute row colors */
        c.real = real\_min + ((\text{float}) x * scale\_real);
        color[x] = cal\_pixel(c);
    }
    send(&i, &y, color, P\text{\texttt{master}}, result\_tag); /* row colors to master */
}
recv(y, P\text{\texttt{master}}, \text{ANYTAG}, \text{source\_tag}); /* receive next row */
Complexity Analysis (sketch)

- Time to compute the Mandelbrot function on a single point is $T_{\text{avg}}$
- Sequential time for the $n$ points in the plane is $n \cdot T_{\text{avg}}$
- Communication overheads are:
  - $n_{\text{rows}} \cdot (T_{\text{startup}} + T_{\text{data}})$ requests
  - $n_{\text{rows}} \cdot (T_{\text{startup}} + n_{\text{cols}} \cdot T_{\text{data}})$ results
  - total = $2 \cdot n_{\text{rows}} \cdot T_{\text{startup}} + n_{\text{rows}} \cdot (1 + n_{\text{cols}}) \cdot T_{\text{data}}$
  - If $n_{\text{rows}}$ is significantly smaller than $n_{\text{rows}} \cdot n_{\text{cols}}$, i.e. $n_{\text{rows}} \ll n$, the total time can be approximated with $n \cdot T_{\text{data}}$ (without considering the startup times)
- We can achieve good speedup if $n \cdot T_{\text{avg}}$ is much more expensive than $n \cdot T_{\text{data}}$
  - It is possible to use pre-fetching
Map-Reduce

- **Map**
  - *Data parallelism*: partition the input domain, and apply in parallel a sequential function (*Map*). Independent tasks.

- **Reduce**
  - To obtain the final result it is needed a *reduction operation* on the partial results of the *map*
    - associative operation of reduction
    - example of associative operations are *min, max, sum*

- An example of Map-Reduce pattern is employed for summing \( n \) numbers on \( m \) processors

![Diagram of Map-Reduce process](image)
Implementation of Map-Reduce

- **Master-slave**

- At beginning the master
  - communicate to all the slave in multicast all data (in this case the slave has to decide on which partition to apply the computation), or
  - communicate to each slave a distinct data partition (*scatter*)

- At the end the master
  - waits to collect data from the slaves, and performs the reduction (*reduction*)

- We can use specific routines (*scatter/reduce*) to optimize communications
Pseudo-Code of Map-Reduce: sum of $n$ integers (point-to-point send/receive routines)

- The master sends the partitions to $m$ slaves, and receives non-deterministically the results. Then, performs the reduction.

Master

```c
s = n/m; /* number of numbers for slaves*/
for (i = 0, x = 0; i < m; i++, x = x + s)
    send(&numbers[x], s, P_i); /* send s numbers to slave */

sum = 0;
for (i = 0; i < m; i++) { /* wait for results from slaves */
    recv(&part_sum, P_ANY);
    sum = sum + part_sum; /* accumulate partial sums */
}
```

Slave

```c
recv(numbers, s, P_master); /* receive s numbers from master */
part_sum = 0;
for (i = 0; i < s; i++) /* add numbers */
    part_sum = part_sum + numbers[i];
send(&part_sum, P_master); /* send sum to master */
```
Pseudo-Code of Map-Reduce: sum of $n$ integers (scatter/reduce)

- scatter and reduce_add are collective operations, to be invoked by all the processes
- The slaves, before calling the reduce, perform the partial sums

```c
Master
s = n/m; /* number of numbers */
scatter(numbers, &s, P_group, root=master); /* send numbers to slaves */
reduce_add(&sum, &s, P_group, root=master); /* results from slaves */

Slave
scatter(numbers, &s, P_group, root=master); /* receive s numbers */
...... /* add numbers */
reduce_add(&part_sum, &s, P_group, root=master); /* send sum to master */
```
Reduction

- The reduction can be implemented more efficiently
  - Baseline: $m$ send/receive point-to-point operations, serialized at the master
- Do it in parallel!
  - $\log(m)$ parallel steps and
Divide and Conquer

• The same as in sequential
  – Decompose into smaller independent problems
  – Use recursion
  – Parallelism comes naturally from task independence

• Sometimes it is used in a parallel setting, even if it may not be a good idea in sequential setting
  – Example: find the minimum of a list
  – Goal: achieve decomposition of the given task
  – Pro: decomposition can be decided dynamically at run-time, in order to keep the best parallelism degree
    • Especially when it is difficult to predict the cost of each task

• Issues:
  – Initial and final stages have small parallelism degree
  – Sub-tasks may not be balanced
Divide-and-Conquer SUM

- The **divide step** is useful for the decomposition
- Each decomposition generates a new task/process
Divide-and-Conquer SUM

- Reduction achieves computation of partial results
- Tasks end in reverse creation order
Divide-and-Conquer SORT

- The pivot is used to partition the data recursively
- Every partition can be assigned to a new independent task/process
- The above Figure identifies the corresponding TDG
Other opportunities for data decomposition

• So far, we have only considered *input* data decomposition

• It is also possible to decompose w.r.t. the *output* data
  – A processor produces a partition of the output
  – The input data is partitioned accordingly
  – More often input data is partially/totally replicated

• By partitioning on the *output*
  – We can infer the *computation assignment* (scheduling) from this data partitioning
  – Each processor should be in charge of producing its own output partition
  – ⇒ owner computes rule
Typical issues in output data partitioning

• Suppose a problem
  – Has input $I$ being partitioned in $I=\{I_1, I_2, \ldots, I_n\}$
  – The algorithm must compute $O=f(I)$

• The output can be partitioned as $O=\{O_1, O_2, \ldots, O_n\}$
  – In good cases $O_1=f(I_1)$, $O_2=f(I_2)$, ...
  – In many cases $O_1=f(1,I)$, $O_2=f(2,I)$, ...
    • The output is a function of the whole input

• There are cases where the partial outputs should be reduced
  – $O = \text{reduce} \ (O_1, O_2, \ldots, O_n)$
Output data partitioning
Matrix-Vector multiplication example

- Problem size: $c[N] = A[N,N] \times b[N]$
- Partition the output vector $c[N]$ among $p$ processors
- Every processor computes independently $N/p$ elements of $c$
  - A copy of $b$ is needed
  - $N/p$ rows of $A$ are needed

\[
A \times b = c
\]

\[
N/p
\]
Output data partitioning
Matrix-Matrix multiplication example

- Problem size: $C[N,N] = A[N,N] \times B[N,N]$
- Partition the matrix $C[N,N]$ among $p^2$ processors
- Every processor computes independently $N^2/p^2$ elements of $C$
  - $N/p$ rows of $A$ are needed
  - $N/p$ cols of $B$ are needed

\[
A \times B = C \quad \text{with area} \quad N^2/p^2
\]
Frequent Itemsets Mining

- Problem definition:
  - 2 inputs: the transaction set $T$, the itemsets $I$
  - 1 output: an array $F$ storing the frequencies of $I$ in $T$
- It is possible to partition w.r.t. the input or the output

<table>
<thead>
<tr>
<th>Database Transactions</th>
<th>Items</th>
<th>Itemset Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C</td>
<td>A, B, C</td>
<td>1</td>
</tr>
<tr>
<td>B, D, E, F, K, L</td>
<td>D, E</td>
<td>3</td>
</tr>
<tr>
<td>A, B, F, H, L</td>
<td>C, F, G</td>
<td>0</td>
</tr>
<tr>
<td>D, E, F, H</td>
<td>A, E</td>
<td>2</td>
</tr>
<tr>
<td>F, G, H, K</td>
<td>C, D</td>
<td>1</td>
</tr>
<tr>
<td>A, E, F, K, L</td>
<td>D, K</td>
<td>2</td>
</tr>
<tr>
<td>G, H, L</td>
<td>C, D, K</td>
<td>0</td>
</tr>
<tr>
<td>D, E, F, K, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F, G, H, L</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Frequent Itemsets Mining

- **Partition the input** $T$:
  - Replicate both $I$ and $F$
  - $F$ must be reduced from local $F$ (sum reduction)

<table>
<thead>
<tr>
<th>Database Transactions</th>
<th>Itemsets</th>
<th>Itemset Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C, E, G, H</td>
<td>A, B, C</td>
<td>1</td>
</tr>
<tr>
<td>B, D, E, F, K, L</td>
<td>D, E</td>
<td>2</td>
</tr>
<tr>
<td>A, B, F, H, L</td>
<td>C, F, G</td>
<td>0</td>
</tr>
<tr>
<td>D, E, F, H</td>
<td>A, E</td>
<td>1</td>
</tr>
<tr>
<td>F, G, H, K,</td>
<td>C, D</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>D, K</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>B, C, F</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>C, D, K</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database Transactions</th>
<th>Itemsets</th>
<th>Itemset Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, E, F, K, L</td>
<td>A, B, C</td>
<td>0</td>
</tr>
<tr>
<td>B, C, D, G, H, L</td>
<td>D, E</td>
<td>1</td>
</tr>
<tr>
<td>G, H, L</td>
<td>C, F, G</td>
<td>0</td>
</tr>
<tr>
<td>D, E, F, K, L</td>
<td>A, E</td>
<td>1</td>
</tr>
<tr>
<td>F, G, H, L</td>
<td>C, D</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>D, K</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>B, C, F</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>C, D, K</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Items</th>
<th>Set 1</th>
<th></th>
<th>Set 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C</td>
<td>0</td>
<td></td>
<td>A, B, C</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>D, E</td>
<td>1</td>
<td></td>
<td>D, E</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>C, F, G</td>
<td>0</td>
<td></td>
<td>C, F, G</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>G, H, L</td>
<td>1</td>
<td></td>
<td>G, H, L</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>D, E, F, K, L</td>
<td>1</td>
<td></td>
<td>D, E, F, K, L</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>F, G, H, L</td>
<td>0</td>
<td></td>
<td>F, G, H, L</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>B, C, F</td>
<td>0</td>
<td></td>
<td>B, C, F</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>C, D, K</td>
<td>0</td>
<td></td>
<td>C, D, K</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Frequent Itemsets Mining

- Partition the output $F$
  - Replicate $T$ and partition the input $I$
  - The final output is just the union of the local results

<table>
<thead>
<tr>
<th>Database Transactions</th>
<th>Itemsets</th>
<th>Itemset Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C, E, G, H</td>
<td>A, B, C</td>
<td>1</td>
</tr>
<tr>
<td>B, D, E, F, K, L</td>
<td>D, E</td>
<td>3</td>
</tr>
<tr>
<td>A, B, F, H, L</td>
<td>C, F, G</td>
<td>0</td>
</tr>
<tr>
<td>D, E, F, H</td>
<td>A, E</td>
<td>2</td>
</tr>
<tr>
<td>F, G, H, K,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A, E, F, K, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B, C, D, G, H, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G, H, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D, E, F, K, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F, G, H, L</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database Transactions</th>
<th>Itemsets</th>
<th>Itemset Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C, E, G, H</td>
<td>A, B, C</td>
<td>1</td>
</tr>
<tr>
<td>B, D, E, F, K, L</td>
<td>D, E</td>
<td>3</td>
</tr>
<tr>
<td>A, B, F, H, L</td>
<td>C, F, G</td>
<td>0</td>
</tr>
<tr>
<td>D, E, F, H</td>
<td>A, E</td>
<td>2</td>
</tr>
<tr>
<td>F, G, H, K,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A, E, F, K, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B, C, D, G, H, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G, H, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D, E, F, K, L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F, G, H, L</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database Transactions</th>
<th>Itemsets</th>
<th>Itemset Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, D</td>
<td>C, D</td>
<td>1</td>
</tr>
<tr>
<td>D, K</td>
<td>D, K</td>
<td>2</td>
</tr>
<tr>
<td>B, C, F</td>
<td>B, C, F</td>
<td>0</td>
</tr>
<tr>
<td>C, D, K</td>
<td>C, D, K</td>
<td>0</td>
</tr>
</tbody>
</table>
Static Matrix Partitions

- Well studied in scientific computations
- Goals:
  - load balancing and/or locality
- Stripe distribution:
  - blocks of rows/columns distributed over an 1-d mesh logical topology

<table>
<thead>
<tr>
<th>row-wise distribution</th>
<th>column-wise distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>$P_0$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$P_1$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$P_2$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$P_3$</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$P_4$</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$P_5$</td>
</tr>
<tr>
<td>$P_6$</td>
<td>$P_6$</td>
</tr>
<tr>
<td>$P_7$</td>
<td>$P_7$</td>
</tr>
</tbody>
</table>
Static Matrix Partitions

- Block distribution over a 2-d mesh (logical) topology:

<table>
<thead>
<tr>
<th>mesh 4x4</th>
<th>mesh 2x8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>$P_0$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$P_1$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$P_2$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$P_3$</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$P_4$</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$P_5$</td>
</tr>
<tr>
<td>$P_6$</td>
<td>$P_6$</td>
</tr>
<tr>
<td>$P_7$</td>
<td>$P_7$</td>
</tr>
<tr>
<td>$P_8$</td>
<td>$P_8$</td>
</tr>
<tr>
<td>$P_9$</td>
<td>$P_9$</td>
</tr>
<tr>
<td>$P_{10}$</td>
<td>$P_{10}$</td>
</tr>
<tr>
<td>$P_{11}$</td>
<td>$P_{11}$</td>
</tr>
<tr>
<td>$P_{12}$</td>
<td>$P_{12}$</td>
</tr>
<tr>
<td>$P_{13}$</td>
<td>$P_{13}$</td>
</tr>
<tr>
<td>$P_{14}$</td>
<td>$P_{14}$</td>
</tr>
<tr>
<td>$P_{15}$</td>
<td>$P_{15}$</td>
</tr>
</tbody>
</table>
Static Matrix Partitions

- $C = A \times B$ : Product matrix $\times$ matrix, size $n^2$, on $p^2$ processors
- Partition over the output $C$ (*block vs. row*)
  - different decompositions require different amounts of data replication
- Data allocated on each processor (num. procs: $P = p^2$)
  - block $\frac{n^2}{p^2} + 2p \frac{n^2}{p^2} = \frac{n^2}{p^2} + 2 \frac{n^2}{p}$
  - row stripe $2 \frac{n^2}{P} + n^2 = 2 \frac{n^2}{p^2} + \frac{n^2}{p}$
  
  for $P=4$ $(p=2)$ the block distribution already becomes favorable (w.r.t. the amount data allocated on each processor)
Cyclic Partitioning

- The cyclic partition is useful to balance the load \textit{statically}.
- A mix of block and cyclic to trade-off between locality and load balancing:
  - block(2)-cyclic over the first dimension of the matrix, over an 1-d mesh of 4 processors.
  - block(4)-cyclic over both the dimensions of the matrix, over a 2-d mesh of 2x2 processors.
Cyclic Partitioning

- (block(2)-cyclic, *)
  - Cyclic blocks of size 2 on the first dimension
  - No partitioning on the second dimensions
  - uni-dimensional mesh (4-ary 1-cube)

Round-robin assignment of blocks of 2 rows

\[
P_0 = p^{(0)}
\]
\[
P_1 = p^{(1)}
\]
\[
P_2 = p^{(2)}
\]
\[
P_3 = p^{(3)}
\]
Cyclic Partitioning

- (block(4)-cyclic, block(4)-cyclic)
  - Cyclic blocks of size 4 on the first and second dimension
  - bi-dimensional mesh (2-ary 2-cube)

\[
\begin{align*}
P_0 &= p_{(0,0)} \\
P_1 &= p_{(0,1)} \\
P_2 &= p_{(1,0)} \\
P_3 &= p_{(1,1)}
\end{align*}
\]
Cyclic Partitioning

- Cyclic partitioning (e.g., block(2)-cyclic on both dimensions) helps with load balancing
- But it may not be sufficient with particularly irregular patterns
  - Compare $P_0$ with $P_{13}$
- Other options: introduce randomization, but we can have drawbacks on the regularity of communication patterns
Synchronized data-parallel computations

- In many data-parallel applications processes need to synchronize at fixed time steps
- E.g. iterative algorithms
  - Each process works on owned data
  - At the end of iteration, processes may need to share partial results, or to request data from other processes
  - Synchronization is needed to make sure updated information is shared among processes
- Typical mechanism is **Barrier**: a collective operation allowing all processes to wait for each other
Shared Memory: centralized counter

Increment of the counter is within a critical section
Message Passing: centralized counter

2n messages are serialized at the master process
Distributed Barrier

Reduction+broadcast
Example: Prefix Sum

- Given a list of integers \( x_0, x_1, \ldots, x_{n-1} \), compute the partial sum up to index \( i \), for every \( i \)
  - \( x_0, x_0+x_1, x_0+x_1+x_2, x_0+x_1+x_2+x_3, \ldots \)
- The optimal sequential algorithm takes \( O(n) \)
  
  \[
  \text{for } (i=1; i<n; i++) \\
  x[i] = x[i-1]+x[i];
  \]
  - It cannot be made parallel due to loop carried dependencies
- Strategy: try to compute partial results for subsequences in parallel, and then combine
  - Can be used for associative operations other than sum
    
    \[
    \text{for } (j=0; j<\log(n); j++) \\
    \text{forall } (i=0; i<n; i++) \\
    \quad \text{if } (i>=2^j) \quad x[i] = x_{\text{old}}[i]+x_{\text{old}}[i-2^j];
    \]
- Note the \text{forall} construct: the \( n-2^j \) (active) body instances are independent and can be executed in parallel
  - \( \log(n) \) parallel steps
Prefix Sum

- **Step 1 (j = 0)**
  - \( \sum_{i=0}^{0} \), \( \sum_{i=1}^{1} \), \( \sum_{i=2}^{2} \), \( \sum_{i=3}^{3} \), \( \sum_{i=4}^{4} \), \( \sum_{i=5}^{5} \), \( \sum_{i=6}^{6} \), \( \sum_{i=7}^{7} \), \( \sum_{i=8}^{8} \), \( \sum_{i=9}^{9} \), \( \sum_{i=10}^{10} \), \( \sum_{i=11}^{11} \), \( \sum_{i=12}^{12} \), \( \sum_{i=13}^{13} \), \( \sum_{i=14}^{14} \), \( \sum_{i=15}^{15} \)

- **Step 2 (j = 1)**
  - \( \sum_{i=0}^{0} \), \( \sum_{i=0}^{1} \), \( \sum_{i=0}^{2} \), \( \sum_{i=0}^{3} \), \( \sum_{i=0}^{4} \), \( \sum_{i=0}^{5} \), \( \sum_{i=0}^{6} \), \( \sum_{i=0}^{7} \), \( \sum_{i=0}^{8} \), \( \sum_{i=0}^{9} \), \( \sum_{i=0}^{10} \), \( \sum_{i=0}^{11} \), \( \sum_{i=0}^{12} \), \( \sum_{i=0}^{13} \), \( \sum_{i=0}^{14} \), \( \sum_{i=0}^{15} \)

- **Step 3 (j = 2)**
  - \( \sum_{i=0}^{0} \), \( \sum_{i=0}^{1} \), \( \sum_{i=0}^{2} \), \( \sum_{i=0}^{3} \), \( \sum_{i=0}^{4} \), \( \sum_{i=0}^{5} \), \( \sum_{i=0}^{6} \), \( \sum_{i=0}^{7} \), \( \sum_{i=0}^{8} \), \( \sum_{i=0}^{9} \), \( \sum_{i=0}^{10} \), \( \sum_{i=0}^{11} \), \( \sum_{i=0}^{12} \), \( \sum_{i=0}^{13} \), \( \sum_{i=0}^{14} \), \( \sum_{i=0}^{15} \)

- **Final step (j = 3)**
  - \( \sum_{i=0}^{0} \), \( \sum_{i=0}^{1} \), \( \sum_{i=0}^{2} \), \( \sum_{i=0}^{3} \), \( \sum_{i=0}^{4} \), \( \sum_{i=0}^{5} \), \( \sum_{i=0}^{6} \), \( \sum_{i=0}^{7} \), \( \sum_{i=0}^{8} \), \( \sum_{i=0}^{9} \), \( \sum_{i=0}^{10} \), \( \sum_{i=0}^{11} \), \( \sum_{i=0}^{12} \), \( \sum_{i=0}^{13} \), \( \sum_{i=0}^{14} \), \( \sum_{i=0}^{15} \)
Prefix Sum: Analysis

- The parallel algorithm takes $\log(n)$ parallel steps with $n$ processors.
- The same algorithm implemented in a sequential manner takes $n \log(n)$ steps.
- We can measure the speed-up as the ratio of the above.
  $$\text{Speedup}(n) = \frac{n \log(n)}{\log(n)} = n$$
  - It seems to be optimal!
  - **Wrong !!!!!!!!!!!!!!!!!!**
- You should always compare with the best sequential algorithm, which is linear in this case.

$$\text{Speed-Up}(n) = \frac{n}{\log(n)}$$

- Observations:
  - We are considering complexity, you should consider the actual running times.
  - We are not taking into account communication costs!
Prefix Sum: Shared Memory Implementation

- Domain decomposition
- Fine-grain (number of processors $p = \text{number of elements } n$)
  \[
i = \text{myrank}(); \quad \text{// thread ID}
\]
  \[
  \text{for (j=0; j<log(n); j++) }
  \]
  \[
  \text{if (i>=2^j) } x[i] = x_{\text{old}}[i]+x_{\text{old}}[i-2^j];
  \]
  \[
  \text{barrier(); } \quad \text{// iteration synch}
  \]
- Coarse-grain ($p<n$) with block partitioning
  \[
k = \text{myrank}(); \quad \text{// thread ID}
\]
  \[
  \text{for (j=0; j<log(n); j++) }
  \]
  \[
  \text{for (i=k*n/p; i<(k+1)*n/p; i++) }
  \]
  \[
  \text{if (i>=2^j) } x[i] = x_{\text{old}}[i]+x_{\text{old}}[i-2^j];
  \]
  \[
  \text{barrier(); } \quad \text{// iteration synch}
  \]
- At the end of each iteration, we need to swap the addresses of $x$ and $x_{\text{old}}$
- The multi-computer implementation (distributed vs. parallel) requires explicit communication of updated portions of the array
Prefix sum: message passing (p=2)

1 = 2^0 communications from P_0 to P_1

2 = 2^1 communications from P_0 to P_1

4 = 2^2 communications from P_0 to P_1

8 = 2^3 communications from P_0 to P_1
Synchronized data parallel iterative methods (geometric domain decomposition)

- **Properties:**
  - Regular (geometric) data decomposition
  - Regular data access patterns
  - Communication costs for partitioning the problem is amortized along many iterations
  - Little sharing and small messages
  - Partition size can be used to tune task granularity and communication costs
- Many scientific computations can be parallelized with this approach
Heat diffusion with finite differences

• Given a metal plate, for which we know the temperature at the edges, what is the temperature distribution inside the plate?

• Solution:
  – Split the square in a fine-grained 2-dimensional mesh of points.
  – The temperature of each point $h_{i,j}$ can be computed with an iterative method as a function of its neighbors:
    $$ h_{i,j} = \frac{(h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1})}{4} $$
  – There are $n^2$ points
  – Iterate the computation $k$ times or until convergence
Heat diffusion with finite differences

• Indeed, this is a method to iteratively solve (Jacobi method) a sparse system of linear equations
  – The unknowns are the values a function $h_{i,j}=f(x_i, y_j)$ for each point $(x_i, y_j)$ of a discrete 2D space
  – The solutions of the equations correspond to the steady state of a function in a 2D physical space (Laplace differential equation), where the conditions on the border are kept constant
Heat diffusion with finite differences

- Constant temperature at the four borders of a 2D plate
- 300° at the bottom border, and 0° at the other three borders
- The iso-temperature curves turn out to be smoother if the grid is denser (more points, more unknowns)

\[ \Delta = 0.25 \]

\[ \Delta = 0.03125 \]
Heat diffusion with finite differences

Stencil
Sequential algorithm

- **Pseudo-code:**

```plaintext
for (iteration = 0; iteration < limit; iteration++) {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
    for (i = 1; i < n; i++) /* update points */
        for (j = 1; j < n; j++)
            h[i][j] = g[i][j];
}
```

- Here we are using a fixed number of iterations
- $g$ is a temporary array for storing intermediate values
- Very regular data access patterns:
  get temperature of your neighbors at distance 1
Fine grain distribution

- Given $N^2$ processors denoted with $P_{i,j}$
- Each processor is responsible for computing $h_{i,j}$
- Pseudo-code:

```c
for (iteration = 0; iteration < limit; iteration++) {
    send(&h, P_{i-1,j}); /* asynchronous sends */
    send(&h, P_{i+1,j});
    send(&h, P_{i,j-1});
    send(&h, P_{i,j+1});
    recv(&w, P_{i-1,j}); /* asynchronous receives */
    recv(&x, P_{i+1,j});
    recv(&y, P_{i,j-1});
    recv(&z, P_{i,j+1});

    h = 0.25 * (w + x + y + z);
}
```

- Too much communications
- Too many synchronization points
Increase the task granularity

- Domain decomposition with granules greater than *single points*
  - BLOCK,BLOCK
  - *-,BLOCK (*stripe*)
(block, block) distribution

- Given $n^2$ points and $p^2$ processors
  - Every processor is given $\frac{n^2}{p^2}$ points
  - The four borders, each of size $\frac{n}{p}$ need to be exchanged with the neighbors
  - These four borders are called *ghost area*
  - The communication volume is proportional to $4 \times \frac{n}{p}$
  - The computational cost is proportional to $\frac{n^2}{p^2}$

- What if we increase granularity?
Heat diffusion: SPMD with (block, block)

- Matrix of $n^2$ elements distributed (block, block) on a grid of $p^2$ processes
- Each process $P_{i,j}$ is thus assigned a block of $n^2/p^2$ elements
- Code executed by a generic SPMD process $P_{i,j}$

```c
<declaration of array h(0:n/p+1, 0:n/p+1) with ghost area>
<declaration of array temp. g(1:n/p, 1:n/p)>
for (iteration = 0; iteration < limit; iteration++) {
    < border exchange of h[][] with
    the 4 neighbors $P_{i-1,j}, P_{i+1,j}, P_{i,j-1}, P_{i,j+1}>$
    for (i = 1; i <= n/p; i++)
        for (j = 1; j <= n/p; j++)
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);
    for (i = 1; i <= n/p; i++) /* Update h[][] */
        for (j = 1; j <= n/p; j++)
            h[i][j] = g[i][j];
}
```

- We have to differentiate the code for blocks/processes on the grid borders
  - $P_{ij}$ where $i,j=0,p-1$
  - less partners with which to exchange communications
- With asynchronous communications, we can postpone the receives
  - the receive must however be completed before accessing the ghost area
Increasing granularity

- 64 points
- 64 computations
- 4 points sent per processor
- $4 \times n_{\text{procs}} = 4 \times 64 = \text{communication volume 256}$

- 64 points
- 64 computations
- 16 points sent per processor
- $16 \times n_{\text{procs}} = 16 \times 4 = \text{communication volume 64}$
Surface / Volume

- The *communication cost* is proportional to the surface (perimeter) of the data partition
- The *computational cost is* proportional to the volume (area) of the data partition
- In two dimensions:
  - The surface increases as $n$
  - The volume increases as $n^2$
- In three dimensions:
  - The surface increases as $n^2$
  - The volume increases as $n^3$
- The ratio *communication/computation* decreases when increasing the partition size
  - Good!
  - This effect is less visible for high dimensional problems
Given \( n^2 \) points and \( p^2 \) processors
- Every processor is given \( \frac{n^2}{p^2} \) points
- Those points form a \( n \times \frac{n}{p^2} \) rectangle
- Two borders, each of size \( n \) need to be exchanged with the neighbors
- The data exchanged are proportional to \( n \) (2 sends and 2 receives)

In the (block,block) distribution
- The data exchanged are proportional to \( \frac{n}{p} \) (4 sends and 4 receives)

To minimize the communication volume:
- (block,block) is better

To minimize the number of communications:
- \( T_{\text{comm}} \) proportional to \( (T_{\text{startup}} + T_{\text{data}}) \)
- If \( T_{\text{startup}} \) is larger than \( T_{\text{data}} \), then striped distribution is better because it minimizes the number of communications (2 vs. 4)
Three golden rules (and wrap-up)

• Social is evil!

• Minimize Volume of data exchanged
• Minimize Frequency of interactions
• Minimize Contention and hot-spots
Pipeline

- Pipeline is a special kind of task-parallelism
- The computation is partitioned into stages, that are executed sequentially
- *Asymptotically*, a pipeline achieves a speed-up equal to the number of stages

![Diagram of a pipeline with stages P0 to P5](image-url)
Pipeline Example

- Frequency filter removing specific frequencies from a digital signal (audio, video, images, …)
- A stream of raw signals enters the pipeline
- A stream of cleaned signals exits the pipeline
Pipeline performance

- $p$ processors, input stream with $n$ elements, tasks with the same cost $t$
  - $(p-1)$ steps to fill the pipeline: time: $(p-1)t$
  - $n$ steps to produce the output: $nt$

- Space-time Diagram
A different Space-Time Diagram

- Input Instances (not processes) on the Y axis
- Processes (not input instances) on the diagram
Pipeline instances

- Pipelining is efficient when we have:
  - **Job Parallelism:**
    - Independent instances of the same problem (different data, same algorithm)
    - The algorithm can be decomposed into a cascading sequence of tasks
    - The various instances (data) may form the input stream
  - **Data Parallelism:**
    - The algorithm processing a portion of the input can be decomposed in a cascading sequence of tasks
    - The partitions of the input become the stream of the pipeline
  - **Pipeline without stream:**
    - The job can be decomposed in a sequence of tasks generating/working on partial results
    - Each stage *early feeds* the next one
Pipeline without stream

Dependency graph

\[ f_4 (f_3 (f_2 (f_1(data)))) \]
Issues: non balanced Pipelines

- The throughput (amount of data per second) of the pipeline depends on the slowest stage
- Given a slow stage
  - Next stages are in idle waiting
  - Previous stages may have buffering problems
- Possible solution:
  - **coarser grain**: assign many “fast” tasks or a few “slow” tasks to a processor
  - Decrease the parallelism degree
Introduce more Parallelism

- Combine Data and Pipeline Parallelism

Still unbalanced, due to the central slower stage

Increment data parallelism

Further subdivide in pipeline
Introduce more Parallelism

- Introduce further parallelism by creating a \textit{farm} whose worker is the slower stage
  - \textbf{Warning}! The stage should be \textit{stateless}: has to depend on the current input only

The first stage forwards different inputs to the replicas of the slow stage, and the last collect the results
Exploratory Search

- There are problems where the goal is to find the best solution in a *solution space*, and this is achieved by *exploring* such space.
  - E.g. optimization problems, games
- The parallelization technique adopted in this case is called *exploratory decomposition*
  - The solution space (or search space) is partitioned (statically or dynamically), and partitions are processed (searched) independently
Example: The 15 Puzzle

- The search space can be represented with a tree
  - Each children in the tree corresponds to a move
  - Each sub-tree can be searched independently
Features of Exploratory Decomposition

• In all the previous decompositions every task result is important

• In exploratory decomposition, when one of the tasks find the solution, the others may become useless
  – If we are looking for a valid solution, other tasks can be forced to end

• If we are looking for the best solution
  – sub-trees are associated with an expected goodness of the solution they contain
  – the most promising tasks are explored first
  – since tasks work in parallel, the exploration order may be different from the serial case, with impacts on the performance
Serial work vs. Parallel work

- Tasks are explored depth-first
- case (a):
  - the solution is at the beginning of third task
  - the parallel implementation immediately reached the solution, and all the other tasks are terminated
- case (b):
  - the solution is at the end of the first task
  - the parallel implementation will have explored everything

<table>
<thead>
<tr>
<th>Case</th>
<th>Total Serial Work</th>
<th>Total Parallel Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>$2m + 1$</td>
<td>$1$</td>
</tr>
<tr>
<td>(b)</td>
<td>$m$</td>
<td>$4m$</td>
</tr>
</tbody>
</table>
Static vs. Dynamic Scheduling/Mapping

• A good design of a parallel algorithm implies to reduce communications, and balance the process workload

• In matrix related problems, static block decompositions /cyclic may achieve both goals

• When interactions are not so regular, we may still exploit a static approach
  – Group small tasks to balance the load
  – Map tasks to minimize communications

• When
  – Tasks do not have uniform cost
  – Tasks have unknown dependencies
  – Tasks are created at run-time
    
    Scheduling and Mapping must be dynamic !
Irregular Static Allocation

- So far we have seen techniques such as cyclic partitioning which help with load balancing
  - With dense matrices
  - Regular interactions/computation
- When dealing with sparse matrices, or irregular interaction patterns, we must use irregular allocations
Irregular Static Allocation: Lake Pollution Example

- Model the lake as a mesh of points/sets of triangles
Irregular Static Allocation

- Each point is assigned to a processor
- Interaction only between neighboring points
- Adoption of *graph partitioning algorithms*, to identify 8 regions, each assigned to a processor
- Interactions only at the perimeter of each region
Dynamic allocation

- Dynamic allocation is mandatory when:
  - The task graph is created dynamically
  - The cost of each task is not known
  - The architecture is non uniform, e.g. sharing computing power on a cluster of workstations
- Load balancing policies are used
- Work Pool is still a feasible solution
  - Upgraded to allow workers to submit new tasks
Centralized Load Balancing

- Slaves can submit new tasks
- The master maintains a dynamic queue of available tasks
Distributed Load Balancing

• Goal:
  – Remove centralization
  – Favor data exchange among neighbors

• Push/Sender-Initiated:
  – The worker that generates a new task sends it to another worker (if it cannot handle it)

• Pull/Receiver-Initiated:
  – When a worker is idle, it asks other workers for a job to execute
Partner selection

- The partner task can be selected:
  - At random, among all the workers
  - Global Round Robin (GRR):
    - A global variable/marker points to the “next” worker
    - A Worker needing a partner reads and increments the global variable
    - Implements a global Round Robin
  - Local Round Robin (LRR):
    - Every processor keeps a private (un-synchronized) pointer to the next available worker
    - No overhead due to sharing a global variable
    - Approximates a global Round Robin
Static versus dynamic mapping

Task Generation

Task Cost

Computing Power

Static Mapping

Dynamic Mapping

STATIC

DYNAMIC

KNOWN

UNKNOWN

KNOWN, HOMOGENEOUS

UNKNOWN, NON HOMOGENEOUS

Static versus dynamic mapping

Task Generation

Task Cost

Computing Power

Static Mapping

Dynamic Mapping

STATIC

DYNAMIC

KNOWN

UNKNOWN

KNOWN, HOMOGENEOUS

UNKNOWN, NON HOMOGENEOUS