Transformation based parallel programming

Program parallelization techniques.

1. **Program Mapping**
   - Program Partitioning. Dependence Analysis.
   - Scheduling & Load balancing.
   - Code distribution.

2. **Data Mapping**.
   - Data partitioning.
   - Communication between processors.
   - Data distribution. Indexing of local data.

Program and data mapping should be **consistent**.
Sequential code:

\[ x=3 \]
For \( i = 0 \) to \( p-1 \).
\[ y(i) = i\times x; \]
Endfor

Dependence analysis:

Scheduling: Replicate \( x = 3 \) (instead of broadcasting).

\[
\begin{array}{cccc}
0 & 1 & 2 & p-1 \\
0x & 1x & 2x & \cdots & (p-1)x \\
\end{array}
\]
SPMD Code:

```c
int x, y, i;
x = 3;
i = mynode();
y = i * x;
```

Data and program distribution:

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Parallel (one node)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td></td>
</tr>
<tr>
<td>Array y [0, 1, ..., p - 1]</td>
<td>Element y</td>
</tr>
<tr>
<td><strong>program</strong></td>
<td></td>
</tr>
<tr>
<td>For i = 0 to p-1</td>
<td>y = i * x</td>
</tr>
<tr>
<td>y(i) = i * x</td>
<td></td>
</tr>
</tbody>
</table>
Dependence Analysis

- For each task, define the input and output sets.

  ![Diagram](INPUT → Task → OUTPUT)

  **Example:** \( S : A = C + B \)

  \[ \text{IN}(S) = \{C,B\} \]

  \[ \text{OUT}(S) = \{A\}. \]

- Given a program with two tasks: \( S_1, S_2. \)
  If changing execution order of \( S_1 \) and \( S_2 \) affects the result. \( \implies S_2 \) depends on \( S_1. \)

- **Type of dependence:**
  1. Flow dependence (true data dependence).
  2. Output dependence.
  3. Anti dependence.
  4. Control dependence (e.g. if \( A \) then \( B \)).
• **Flow Dependence:** \( \text{OUT}(S_1) \cap \text{IN}(S_2) \neq \emptyset \)

\[
S_1 : A = x + B \\
S_2 : C = A + 3
\]

S2 is dataflow-dependent on S1.

• **Output Dependence:** \( \text{OUT}(S_1) \cap \text{OUT}(S_2) \neq \emptyset \).

\[
S_1 : A = 3 \\
S_2 : A = x
\]

S2 is output-dependent on S1.

• **Anti Dependence:** \( \text{IN}(S_1) \cap \text{OUT}(S_2) \neq \emptyset \).

\[
S_1 : B = A + 3 \\
S_2 : A = x + 5
\]

S2 is anti-dependent on S1.
Coarse-grain dependence graph.

Tasks operate on data items of large sizes and perform a large chunk of computations.

**Ex:**

- $S_1: A = f(X,B)$
- $S_2: C = g(A)$
- $S_3: A = h(A,C)$
Delete redundant dependence edges

The deletion should not affect the correctness.
An anti or output dependence edge can be deleted if it is subsumed by another dependence path.

```
S_1 -> Flow -> S_2 -> Flow -> S_3
```

CS, UCSB

Tao Yang
**Iteration space** – all iterations of a loop and data dependence between iteration statements.

1 D Loop:

\[
\text{For } i = 1 \text{ to } n \\
S_i : a_i = b_i + c_i
\]

\[
S_1 \quad S_2 \quad S_3 \ldots \quad S_n
\]

\[
\text{For } i = 1 \text{ to } n \\
S_i : a_i = a_{i-1} - 1
\]

\[
S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n
\]

2 D Loop:

\[
\text{For } i = 1 \text{ to } n \\
\text{For } j = 1 \text{ to } n \\
S_{ij} : x_{ij} = x_{i-1,j} + 1
\]

\[
S_{11} \quad S_{12} \quad S_{13} \\
S_{21} \quad S_{22} \quad S_{23} \\
S_{31} \quad S_{32} \quad S_{33}
\]
Program Partitioning

Purpose:

- Increase task grain size.
- Reduce unnecessary communication.
- Ease the mapping of a large number of tasks to a small number of processors.

Actions: Group several tasks together as one task.

Loop partitioning techniques:

- Loop blocking/unrolling.
- Interior loop blocking.
- Loop interchange.
Loop blocking/unrolling

Given:

For \( i = 1 \) to \( 2n \)

\[
S_i : a_i = b_i + c_i
\]

\[
\begin{array}{ccc}
S_1 & S_2 & S_3 & S_4 & \ldots & S_{2n-1} & S_{2n} \\
1 & 2 & & & & n
\end{array}
\]

After transformation:

\[
\Rightarrow \text{For } i = 1 \text{ to } n
\]

\[
\text{do : } S_{2i-1}, S_{2i}
\]
General 1D Loop Blocking

Given: \( \text{For } i = 1 \text{ to } r \cdot p \)
\[ S_i : a(i) = b(i) + c(i) \]

Blocking this loop by a factor of \( r \):
\[
\begin{align*}
\text{For } j = 0 \text{ to } p-1 \\
\text{For } i = r \cdot j + 1 \text{ to } r \cdot j + r \\
a(i) = b(i) + c(i)
\end{align*}
\]

SPMD code on \( p \) nodes.
\[
\begin{align*}
\text{me} &= \text{mynode();} \\
\text{For } i = r \cdot \text{me} + 1 \text{ to } r \cdot \text{me} + r \\
a(i) &= b(i) + c(i)
\end{align*}
\]
Block the interior loop and make it one task.

**Example:**

For \( i = 1 \) to 4

For \( j = 1 \) to 4

\[ x_{i,j} = x_{i,j-1} + 1 \]

**After blocking:**

For \( i = 1 \) to 4

For \( j = 1 \) to 4

\[ x_{i,j} = x_{i,j-1} + 1 \]

The above example preserves the parallelism.
Partitioning may reduce parallelism

For $i = 1$ to $4$

For $j = 1$ to $4$

$$x_{i,j} = x_{i-1,j} + 1$$

No parallelism!
**Loop Interchange**

**Definition:** A program transformation that changes the execution order of a loop program.

**Actions:** Swap the loop control statements.

**Example:**

For $i = 1$ to 4
   For $j = 1$ to 4
      $x_{i,j} = x_{i-1,j} + 1$

**After loop interchange:**

For $j = 1$ to 4
   For $i = 1$ to 4
      $x_{i,j} = x_{i-1,j} + 1$
**Why loop interchange?**

**Usage:** Help loop partitioning to exploit more parallelism.

**Example.** *Interior loop blocking after interchange.*

For $j = 1$ to 4

For $i = 1$ to 4

\[ x_{ij} = x_{i-1j} + 1 \]
Execution order after loop interchange

Loop interchange alters the execution order.

For $i = 1$ to $3$
For $j = 1$ to $3$
$S_{i,j}$:

For $j = 1$ to $3$
For $i = 1$ to $3$
$S_{i,j}$:
Not every loop interchange is legal

Loop interchange is not legal if the new execution order violates data dependence.

For $i = 1$ to $3$
For $j = 1$ to $3$

$S_{ij}$: $X(i, j) = X(i-1, j+1) + 1$

Legal?

For $j = 1$ to $3$
For $i = 1$ to $3$

$S_{ij}$: $X(i, j) = X(i-1, j+1) + 1$
Interchanging triangular loops

For $i=1$ to $10$  $\iff$  For $j=2$ to $10$

For $j=i+1$ to $10$  For $i=1$ to $j-1$
Transformation for loop interchange

How to derive the new bounds for $i$ and $j$ loops?

- **Step 1:** List all inequalities regarding $i$ and $j$ from the original code.
  
  \begin{align*}
  i &\leq 10, \quad i \geq 1, \\
  j &\leq 10, \quad j \geq i + 1.
  \end{align*}

- **Step 2:** Derive bounds for loop $j$.
  \begin{itemize}
  \item Extract all inequalities regarding the upper bound of $j$.
    \begin{align*}
    j &\leq 10.
    \end{align*}

    The upper bound is 10.
  \item Extract all inequalities regarding the lower bound of $j$.
    \begin{align*}
    j &\geq i + 1.
    \end{align*}

    The lower bound is 2 since $i$ could be as low as 1.
  \end{itemize}

- **Step 3:** Derive bounds for loop $i$ when $j$
value is fixed (now loop $i$ is an inner loop).

- Extract all inequalities regarding the upper bound of $i$.

$$i \leq 10, \quad i \leq j - 1.$$  

The upper bound is $\min(10, j - 1)$.

- Extract all inequalities regarding the lower bound of $i$.

$$i \geq 1.$$  

The lower bound is $1$. 
Data Partitioning and Distribution

Data structure is divided into *data units* and assigned to processor local memories.

**Why?**

- Not enough space for replication for solving large problems.
- Partition data among processors so that data accessing is localized for tasks.

**Ex:** \( y = A_{n \times n} \cdot x \)

Distribute array \( A \) among \( p \) nodes. But replicate \( x \) to all processors.
Corresponding Task Mapping: \( (r = n/p) \)

\[
\begin{array}{ccc}
P_0 & P_1 & \cdots \\
A_1x & A_{r+1}x & \\
A_2x & A_{r+2}x & \cdots \\
& & \\
A_rx & A_{2r}x & \\
\end{array}
\]
1D Data Mapping Methods

1D array $\longrightarrow$ 1D processors.

- Assume that data items are counted from $0, 1, \cdots n - 1$.
- Processors are numbered from 0 to $p - 1$.

Mapping methods: Let $r = \left\lfloor \frac{n}{p} \right\rfloor$.

- 1D Block

\[
\begin{array}{cccc}
\hline
\text{Data} & \Rightarrow & \text{Proc} \\
\hline
i & \left\lfloor \frac{i}{r} \right\rfloor \\
\hline
\end{array}
\]
- **1D Cyclic**

  \[
  \begin{array}{cccccccc}
  p & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 \\
  \text{Data} & \implies & \text{Proc} \\
  i & & i \mod p
  \end{array}
  \]

- **1D Block Cyclic.**

  First the array is divided into a set of units using block partitioning (block size \( b \)). Then these units are mapped in a cyclic manner to \( p \) processors.

  \[
  \begin{array}{cccccccc}
  p & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 \\
  \text{Data} & \implies & \text{Proc} \\
  i & & \lfloor \frac{i}{b} \rfloor \mod p
  \end{array}
  \]
Data items are counted from 0, 1, \cdots n - 1. Processors are numbered from 0 to p - 1.

**Methods:**

- **Column-wise block.** (call it (*,block))
  \[
  \text{Data } (i, j) \Rightarrow \text{Proc } \left\lfloor \frac{j}{r} \right\rfloor
  \]

- **Row-wise block.** (call it (block,*))
  \[
  \text{Data } (i, j) \Rightarrow \text{Proc } \left\lfloor \frac{i}{r} \right\rfloor
  \]
• **Row cyclic.** (cyclic,*)
  
  Data \((i, j) \Rightarrow \text{Proc } i \mod p.\)

• **Others:** Column cyclic. Column block cyclic.
  Row block cyclic \ldots.\
Data elements are counted as \((i, j)\) where
\[0 \leq i, j \leq \cdots n - 1.\]

Processors are numbered as \((s, t)\) where
\[0 \leq s, t \leq \cdots q - 1\] where \(q = \sqrt{p}\). Let \(r = \left\lceil \frac{n}{q} \right\rceil\).

- (Block, Block)

\[
\text{Data } (i, j) \Rightarrow Proc \left( \left\lfloor \frac{i}{r} \right\rfloor, \left\lfloor \frac{j}{r} \right\rfloor \right)
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Proc (0,0)</td>
<td>Proc (0,1)</td>
<td>Proc (0,2)</td>
<td>Proc (0,3)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• **(Cyclic, Cyclic)**
  
  Data \((i, j) \Rightarrow \text{Proc} \ (i \mod q, j \mod q, )\)

• **Others:** (Block, Cyclic), (Cyclic, Block), (Block Cyclic, Block Cyclic).
Program & data mapping: Consistency

Criteria:

- Sufficient parallelism is provided by partitioning.
- Also the number of distinct units accessed by each task is minimized.

A simple mapping heuristic:

“Owner Computes Rule”. If task \( x \) modifies data item \( i \), then processor that owns \( i \) executes \( x \).
An Example of “Owner computes rule”

Sequential code:

For $i = 0$ to $r^*p-1$

$S_i : a[i] = 3.$

Data distribution:

Map data $a(i)$ to node $proc_{-}map(i)$.

Data array $a(i)$ are distributed to processors such that if processor $x$ executes $a(i) = 3$, then $a(i)$ is assigned to processor $x$.

SPMD code on $p$ processors:

```c
me=mynode();
For i =0 to rp-1
    if ( proc_{-}map(i) == me) a[i] = 3.
```
Define: \(proc\_map(i) = \left\lfloor \frac{i}{r} \right\rfloor\).

Data distribution:
Processor 0 owns data \(a(0), a(1), \ldots, a(r - 1)\).
Processor 1 owns data \(a(r), a(r + 1), \ldots, a(2r - 1)\).
\ldots

Code distribution:

```java
me=mynode();
For i =0 to rp-1
    if ( proc_map(i) == me) a[i] = 3.
```

Comments: General, but with extra loop overhead.
**Optimization:** Blocking by a factor of $r$.

\[
\begin{aligned}
\text{For } j = 0 \text{ to } p-1 \\
\quad \text{For } i = r^*j \text{ to } r^*j+r-1 \\
\quad \quad a[i] = 3.
\end{aligned}
\]

**Optimized SPMD code on p processors:**

\[
\begin{aligned}
\text{me=mynode();} \\
\text{For } i = r^*\text{me} \text{ to } r^*\text{me}+r-1 \\
\quad a[i] = 3.
\end{aligned}
\]
SPMD code with 1D cyclic mapping

Define: \( \text{proc\_map}(i) = i \mod p \).

Data distribution:
Processor 0 owns data \( a(0), a(p), a(2p), \ldots \).
Processor 1 owns data \( a(1), a(p+1), a(2p+1), \ldots \).

Optimized SPMD code on \( p \) processors:

\[
\text{me}=\text{mynode}();
\text{For } i = \text{me} \text{ to } r\times p-1 \text{ step } p
\begin{align*}
a[i] &= 3. \\
\end{align*}
\]
Global Data Space vs. Local Address

Sequential program ⇒ Global data address
Distributed program ⇒ Local data address

Data indexing in

```
me=mynode();
For i =0 to rp-1
    if (proc_map(i) == me) a[i] = 3.
```

Problem: “a(i)=3” uses “i” as the index function and the value of i is in a range between 0 to rp – 1. Each processor has to allocate the entire array!

Data localization: Allocate r units for each processor, translate the global index i to a local index which accesses the local memory only.
From global address to local address

Use 1D block mapping.

SPMD code.

```c
int a[r]; /* Not entire array! */
me=mynode();
For i =0 to rp-1
    if ( proc_map(i) == me) a[local(i)] = 3.
```
Mapping Function for 1D Block:

\[ \text{Local}(i) = i \mod r. \]

Ex. \( p=2, \ r=3. \)

\[ \begin{array}{c|c}
\text{Proc 0} & \text{Proc 1} \\
\hline
0 \to 0 & 3 \to 0 \\
1 \to 1 & 4 \to 1 \\
2 \to 2 & 5 \to 2 \\
\end{array} \]

Mapping Function for 1D Cyclic:

\[ \text{Local}(i) = \left\lfloor \frac{i}{p} \right\rfloor. \]

Ex. \( p=2. \)

\[ \begin{array}{c|c}
\text{proc 0} & \text{proc 1} \\
\hline
0 \to 0 & 1 \to 0 \\
2 \to 1 & 3 \to 1 \\
4 \to 2 & 5 \to 2 \\
6 \to 3 & \\
\end{array} \]
Given: data item $i$.

- **1D Block**

  Processor ID:

  \[ \text{proc\_map}(i) = \left\lfloor \frac{i}{r} \right\rfloor \]

  Local data address:

  \[ \text{Local}(i) = i \mod r \]

- **1D Cyclic**

  Processor ID:

  \[ \text{proc\_map}(i) = i \mod p \]

  Local data address:

  \[ \text{Local}(i) = \left\lfloor \frac{i}{p} \right\rfloor. \]
**Program Parallelization**

Program

→ Code Partitioning

→ Data Partitioning

Tasks +
dependence

→ mapping

→ scheduling

P processors

Data

→ mapping

→ P processors

parallel code

**Techniques**

- cyclic/block partitioning
- Loop interchange, unrolling, blocking
- Dependence analysis
- Task scheduling
- Task mapping, Data mapping.
  (cyclic/ block mapping)
- Data indexing and communication.