Systolic Arrays

This is a form of **pipelining**, sometimes in more than one dimension.

The term “systolic” was first used in this context by H.T. Kung, then at CMU; it refers to the “pumping” action of a heart.

Machines have been constructed based on this principle, notable the iWARP, fabricated by Intel.

---

**Systolic Matrix Multiplication**

- Processors are arranged in a 2-D grid.
- Each processor accumulates one element of the product.
- The elements of the matrices to be multiplied are “pumped through” the array.

---

**Systolic Matrix Multiplication Illustrated with two 3x3 matrices**

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0,2</td>
<td>a0,1</td>
<td>a0,0</td>
</tr>
<tr>
<td>a1,2</td>
<td>a1,1</td>
<td>a1,0</td>
</tr>
<tr>
<td>a2,2</td>
<td>a2,1</td>
<td>a2,0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b0,0</td>
<td>b0,1</td>
<td>b0,2</td>
</tr>
<tr>
<td>b1,0</td>
<td>b1,1</td>
<td>b1,2</td>
</tr>
<tr>
<td>b2,0</td>
<td>b2,1</td>
<td>b2,2</td>
</tr>
</tbody>
</table>
```

Alignments in time:

- **Columns of A:**
  - a0,0
  - a1,0
  - a2,0

- **Rows of B:**
  - b0,0
  - b1,0
  - b2,0

- **A0,0 * B0,0 + A0,1 * B1,0 + A1,0 * B0,0:**

---

**Systolic Matrix Multiplication Illustrated with two 3x3 matrices**

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0,2</td>
<td>a0,1</td>
<td></td>
</tr>
<tr>
<td>a1,2</td>
<td>a1,1</td>
<td>a1,0</td>
</tr>
<tr>
<td>a2,2</td>
<td>a2,1</td>
<td>a2,0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b0,0</td>
<td>b0,1</td>
<td>b0,2</td>
</tr>
<tr>
<td>b1,0</td>
<td>b1,1</td>
<td>b1,2</td>
</tr>
<tr>
<td>b2,0</td>
<td>b2,1</td>
<td>b2,2</td>
</tr>
</tbody>
</table>
```

Alignments in time:

- **Columns of A:**
  - a0,0
  - a1,0

- **Rows of B:**
  - b0,0
  - b1,0

- **A0,0 * B0,0 + A0,1 * B1,0:**

---

**Systolic Matrix Multiplication Illustrated with two 3x3 matrices**

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0,2</td>
<td>a0,1</td>
<td></td>
</tr>
<tr>
<td>a1,2</td>
<td>a1,1</td>
<td>a1,0</td>
</tr>
<tr>
<td>a2,2</td>
<td>a2,1</td>
<td>a2,0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b0,0</td>
<td>b0,1</td>
<td>b0,2</td>
</tr>
<tr>
<td>b1,0</td>
<td>b1,1</td>
<td>b1,2</td>
</tr>
<tr>
<td>b2,0</td>
<td>b2,1</td>
<td>b2,2</td>
</tr>
</tbody>
</table>
```

Alignments in time:

- **Columns of A:**
  - a0,0
  - a1,0

- **Rows of B:**
  - b0,0
  - b1,0

- **A0,0 * B0,0 + A0,1 * B1,0:**

---

**Systolic Matrix Multiplication Illustrated with two 3x3 matrices**

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0,2</td>
<td>a0,1</td>
<td></td>
</tr>
<tr>
<td>a1,2</td>
<td>a1,1</td>
<td>a1,0</td>
</tr>
<tr>
<td>a2,2</td>
<td>a2,1</td>
<td>a2,0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b0,0</td>
<td>b0,1</td>
<td>b0,2</td>
</tr>
<tr>
<td>b1,0</td>
<td>b1,1</td>
<td>b1,2</td>
</tr>
<tr>
<td>b2,0</td>
<td>b2,1</td>
<td>b2,2</td>
</tr>
</tbody>
</table>
```

Alignments in time:

- **Columns of A:**
  - a0,0
  - a1,0

- **Rows of B:**
  - b0,0
  - b1,0

- **A0,0 * B0,0 + A0,1 * B1,0:**
Systolic Matrix Multiplication
Illustrated with two 3x3 matrices

A Related Algorithm:
Cannon’s Method

- Let’s take another view of systolic multiplication: Consider the rows and columns of the matrices to be multiplied as strips that are slide past each other.

- The strips are staggered so that the correct elements are multiplied at each time step.
First step

Second step

Third step

Fourth step

Fifth step

Cannon's Method

- Rather than have some processors idle,
- wrap the array rows and columns so that every processor is doing something on each step.
- In other words, rather than feeding in the elements, they are rotated around,
- starting in an initially staggered position as in the systolic model.
- We also change the order of products slightly, to make it correspond to more natural storage by rows and columns.
Application of Cannon’s Technique

- Consider matrix multiplication of $2^n \times n$ matrices on a distributed memory machine, on say, $n^2$ processing elements.
- An obvious way to compute is to think of the PE’s as a matrix, with each computing one element of the product.
- We would send each row of the matrix to $n$ processors and each column to $n$ processors.
- In effect, in the obvious way, each matrix is stored a total of $n$ times.

Obvious Matrix Multiply

Columns of $b$ distributed to each PE in column.

Rows x Columns on respective PEs.

Cannon’s Method

- Cannon’s method avoids storing each matrix $n$ times, instead cycling ("piping") the elements through the PE array.
- (It is sometimes called the "pipe-roll" method.)
- The problem is that this cycling is typically too fine-grain to be useful for element-by-element multiply.

Partitioned Multiplication

- Partitioned multiplication divides the matrices into blocks.
- It can be shown that multiplying the individual blocks as if elements of matrices themselves gives the matrix product.

Block Multiplication

one block of product

columns of blocks

rows of blocks
Cannon’s Method is Fine for Block Multiplication

- The blocks are aligned initially as the elements were in our description.
- At each step, entire blocks are transmitted down and to the left of neighboring PE’s.
- Memory space is conserved.

Exercise

- Analyze the running time for the block version of Cannon’s method for two n x n matrices on p processors, using t_{comp} as the unit operation time and t_{comm} as the unit communication time and t_{start} as the per-message latency .
- Assume that any pair of processors can communicate in parallel.
- Each block is (n/sqrt(p)) x (n/sqrt(p)).

Fox’s Algorithm

- Also for block matrix multiplication, it has a resemblance to Cannon’s algorithm.
- The difference is that on each cycle:
  - A row block is broadcast to every other processor in the row.
  - The column blocks are rolled cyclically.

Fox vs. Cannon

Synchronous Computations

PP Chapter 6
Barriers

- Mentioned earlier
- Synchronize all of a group of processes
- Used in both distributed and shared-memory
- Issue: Implementation & cost

Counter Method for Barriers

- One-phase version
  - Use for distributed-memory
  - Each processor sends a message to the others when barrier reached.
  - When each processor has received a message from all others, the processors pass the barrier

Counter Method for Barriers

- Two-phase version
  - Use for shared-memory
  - Each processor sends a message to the master process.
  - When the master has received a message from all others, it sends messages to each indicating they can pass the barrier.
  - Easily implemented with blocking receives, or semaphores (one per processor).

Tree Barrier

- Processors are organized as a tree, with each sending to its parent.
- Fan-in phase: When the root of the tree receives messages from both children, the barrier is complete.
- Fan-out phase: Messages are then sent down the tree in the reverse direction, and processes pass the barrier upon receipt.

Butterfly Barrier

- Essentially a fan-in tree for each processor, with some sharing toward the leaves.
- Advantage is that no separate fan-out phase is required.
**Barrier Bonuses**

- To implement a barrier, it is only necessary to increment a count (shared memory) or send a couple of messages per process.
- These are communications with null content.
- By adding content to messages, barriers can have added utility.

**Data Parallel Computations**

- PP Section 6.2.1
- `forall` statement:
  ```
  forall( j = 0; j < n; j++ )
  {
    ... body done in parallel for all j ...
  }
  ```

**Data Parallel Computations**

- `forall` synchronization assumptions
  - There are different interpretations of `forall`, so you need to “read the fine print”.
  - Possible assumptions from weakest to strongest:
    - No implied synchronization
    - Implied barrier at the end of each loop body
    - Implied barrier before each assignment
    - Each machine instruction synchronized, SIMD-fashion

**Example: Prefix-Sum**

- Assume that n is a power of 2.
- Assume shared memory.
- Assume barrier before assignments
- for( j = 0; j < log(n); j++ )
  ```
  forall( i = 2^j; i < n; i++ )
  ```
  ```
  x[i] += x[i - 2^j];
  ```
  effectively buffered new value
Implementing \texttt{forall} using SPMD:

Assuming PP: “Synchronous Iteration” (barrier at end of body)

\begin{itemize}
  \item \texttt{for( j = 0; j < \log(n); j++ )}
    \texttt{forall( i = 0; i < n; i++)}
    \texttt{Body(i);}
    \texttt{implimentable in SPMD as:}
    \begin{itemize}
    \item \texttt{for( j = 0; j < \log(n); j++ )}
      \begin{itemize}
      \item \texttt{i = my_process_rank();}
      \item \texttt{Body(i);}
      \item \texttt{barrier();}
      \end{itemize}
    \end{itemize}
  \end{itemize}

\begin{itemize}
  \item \texttt{forall processes implicit}
\end{itemize}

\textbf{Example: Iterative Linear Equation Solver}

\begin{itemize}
  \item \texttt{for( iter = 0; iter < numIterations; iter++)}
    \texttt{forall( i = 0; i < n; i++)}
    \begin{itemize}
    \item \texttt{double sum = 0;}
    \item \texttt{for( j = 0; j < n; j++ )}
      \texttt{sum += a[i][j]*x[j];}
    \item \texttt{x[i] = sum;}
    \end{itemize}
\end{itemize}

\textbf{Outer forall processes implicit}

\textbf{Iterative Linear Equation Solver: Translation to SPMD}

\begin{itemize}
  \item \texttt{for( iter = 0; iter < numIterations; iter++)}
    \texttt{forall( i = 0; i < n; i++)}
    \begin{itemize}
    \item \texttt{i = my_process_rank();}
    \item \texttt{double sum = 0;}
    \item \texttt{for( j = 0; j < n; j++ )}
      \texttt{sum += a[i][j]*x[j];}
    \item \texttt{new_x[i] = sum;}
    \item \texttt{all gather new_x to x (implied barrier)}
    \end{itemize}
\end{itemize}

\textbf{Nested forall's}

\begin{itemize}
  \item \texttt{for( iter = 0; iter < numIterations; iter++)}
    \texttt{forall( i = 0; i < m; i++)}
    \begin{itemize}
    \item \texttt{forall( j = 0; j < n; j++)}
      \texttt{Body(i, j)}
    \end{itemize}
\end{itemize}

\textbf{Example of nested forall's: Laplace Heat equation}

\begin{itemize}
  \item \texttt{for( iter = 0; iter < numIterations; iter++)}
    \texttt{forall( i = 0; i < m; i++)}
    \begin{itemize}
    \item \texttt{forall( j = 0; j < n; j++)}
      \texttt{Body(i, j)}
      \texttt{x[i][j] = (x[i-1][j] + x[i][j-1] + x[i+1][j] + x[i][j+1]) / 4.0;}
    \end{itemize}
\end{itemize}

\textbf{Exercise}

\begin{itemize}
  \item \texttt{How would you translate nested forall's to SPMD?}
\end{itemize}
Cellular Automata

- Synchronous computation
- Infinitely-large grid (finite occupancy)
- Various dimensions (1D, 2D, 3D, ...)
- Typically fine-grain
- If distributed, still need to communicate across boundaries, once per cycle.

CA Results

- Universal computers (embedded turing machines)
- Self-reproduction
- Diffusion equations
- Artificial Life
- Digital Physics

Example Simulation Models

- “Game of Life”
- Gas particles: Billiard-ball model
- Ising model: Ferro-magnetic spins
- Heat equation simulation
- Percolation models
- Wire models
- Lattice Gas models

Oil&Water Simulation
(Bruce Boghosian, Boston Univ.)

Surfactant Formation
(Bruce Boghosian, Boston Univ.)