Distributed Memory Algorithms

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine
SIMD Mesh Matrix Multiplication

Single Instruction, Multiple Data

- $n^2$ processors,
- $3n$ time.

Algorithm: see slide.
SIMD Mesh Matrix Multiplication

1. Precondition array
   - Shift row $i$ by $i - 1$ elements west,
   - Shift column $j$ by $j - 1$ elements north.

2. Multiply and add
   On processor $\langle i, j \rangle$:
   \[ c = \sum_k a_{ik} \ast b_{kj} \]
   - Inverted dimensions
     - Matrix $\downarrow i, \rightarrow j$.
     - Processor array $\downarrow iyproc, \rightarrow ixproc$.
   - $n$ shift and $n$ arithmetic operations.
   - $n^2$ processors.

*Maspar program: see slide.*
SIMD Cube Matrix Multiplication

Cube of $d^3$ processors

Idea

• Map $A(i, j)$ to all $P(j, i, k)$
• Map $B(i, j)$ to all $P(i, k, j)$
SIMD Cube Matrix Multiplication

Multiplication and Addition

- Each processor computes single product
  \[ P_{ijk} : c_{ijk} = a_{ik} \times b_{kj} \]
- Bars along x-directions are added
  \[ P_{0ij} : C_{ij} = \sum_k c_{ijk} \]
SIMD Cube Matrix Multiplication

Maspar Program

```c
int A[N,N], B[N,N], C[N,N];
plural int a, b, c;

a = A[iyproc, ixproc];
b = B[ixproc, izproc];
c = a*b;

for (i = 0; i < N-1; i++)
    if (ixproc > 0)
        c = xnetE[1].c
    else
        c += xnetE[1].c;

if (ixproc == 0) C[iyproc, izproc] = c;
```

- $O(n^3)$ processors,
- $O(n)$ time.
SIMD Cube Matrix Multiplication

Tree-like summation

plural x, d;
...

x = ixproc;
d = 1;
while (d < N) {
    if (x % 2 != 0) break;
    c += xnetE[d].c;
    x /= 2;
    d *= 2;
}

if (ixproc == 0) C[iyproc, izproc] = c;

• $O(\log n)$ time
• $O(n^3)$ processors

Long-distance communication required!
SIMD Hypercube Mat. Multiplication

- $d$-dimensional hypercube $\Rightarrow$ processors indexed with $d$ bits.
- $p_1$ and $p_2$ differ in $i$ bits $\Rightarrow$ shortest path between $p_1$ and $p_2$ has length $i$. 
SIMD Hypercube Matrix Multiplication

Mapping of cube with dimension $n$ to hypercube with dimension $d$.

- Hypercube of $n^3 = 2^d$ processors $\Rightarrow d = 3s$ (for some $s$).
- 64 processors $\Rightarrow n = 4, d = 6, s = 2$.
  
  Hypercube $d_5d_4d_3d_2d_1d_0$
  Cube $x\ y\ z$

- Embedding algorithm
  
  - Cube indices in binary form ($s$ bits each)
  - Concatenate indices ($3s = d$ bits)

- Better: use Gray code $G$ (see later)

  $d_5d_4d_3d_2d_1d_0$
  $G(x)\ G(y)\ G(z)$

  - Neighbor processors in cube remain neighbors in hypercube.
  - Any cube algorithm can be executed with same efficiency on hypercube.
SIMD Hypercube Matrix Multiplication

Tree summation in hypercube.

<table>
<thead>
<tr>
<th>Processors</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>$r_0$</td>
<td>$s_0$</td>
<td>$r_1$</td>
<td>$s_1$</td>
<td>$r_2$</td>
<td>$s_2$</td>
<td>$r_3$</td>
<td>$s_3$</td>
</tr>
<tr>
<td>Step 2</td>
<td>$r_0$</td>
<td>$s_0$</td>
<td>$r_1$</td>
<td>$s_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 3</td>
<td>$r_0$</td>
<td></td>
<td>$s_0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Each processor receives value from neighboring processors only.
- Only short-distance communication is required.

*Cube algorithm can be more efficient on hypercube!*
Row/Column-Oriented Matrix Multiplication

1. Load $A_i$ on every processor $P_i$.
2. For all $P_i$ do:
   
   for $j=0$ to $N-1$
   
   Receive $B_j$ from root
   
   $C_{ij} = A_i \times B_j$

3. Collect $C_i$

Broadcasting of each $B_j \Rightarrow$ Step 2 takes $O(N \log N)$ time.
Ring Algorithm

See Quinn, Figure 7-15.

- Change order of multiplication by
- Using a ring of processors.

1. Load $A_i$ and $B_i$ on every processor $P_i$.
2. For all $P_i$ do:
   
   $p = (i+1) \mod N$
   
   $j = i$
   
   for $k=0$ to $N-1$ do
   
   $C_{ij} = A_i \times B_j$
   
   $j = (j+1) \mod N$
   
   Receive $B_j$ from $P_p$

3. Collect $C_i$

Point-to-point communication $\Rightarrow$ Step 2 takes $O(N)$ time.
**Hypercube Algorithm**

Problem: How to embed ring into hypercube?

- **Simple solution** $H(i) = i$:
  - Ring processor $i$ is mapped to hypercube processor $H(i)$.
  - Massive non-neighbor communication!

- **How to preserve neighbor-to-neighbor communication?** (see Quinn, Figure 5-13)

- **Requirements for** $H(i)$:
  - $H$ must be a 1-to-1 mapping.
  - $H(i)$ and $H(i + 1)$ must differ in 1 bit.
  - $H(0)$ and $H(N - 1)$ must differ in 1 bit.

*Can we construct such a function $H$?*


Ring Successor

Assume $H$ is given.

- Given: hypercube processor number $i$
- Wanted: “ring successor” $S(i)$

$$S(i) = \begin{cases} 0, & \text{if } i = N - 1 \\ H(H^{-1}(i) + 1), & \text{otherwise} \end{cases}$$

Same technique for embedding a 2-D (or even $n$-D) mesh into an hypercube (see Quinn, Figure 5-14).
Gray Codes

Recursive construction.

• 1-bit Gray code $G_1$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$G_1(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• $n$-bit Gray code $G_n$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$G_n(i)$</th>
<th>$i$</th>
<th>$G_n(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$0G_{n-1}(0)$</td>
<td>$n - 1$</td>
<td>$1G_{n-1}(0)$</td>
</tr>
<tr>
<td>1</td>
<td>$0G_{n-1}(1)$</td>
<td>$n - 2$</td>
<td>$1G_{n-1}(1)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\frac{n}{2} - 1$</td>
<td>$0G_{n-1}(\frac{n}{2} - 1)$</td>
<td>$\frac{n}{2}$</td>
<td>$1G_{n-1}(\frac{n}{2} - 1)$</td>
</tr>
</tbody>
</table>

• Required properties preserved by construction!

$$H(i) = G(i) = i \text{ xor } \frac{i}{2}.$$
Gray Code Computation

C functions.

• Gray-Code

```c
int G(int i)
{
    return(i ^ (i/2));
}
```

• Inverse Gray-Code

```c
int G_inv(int i)
{
    int answer, mask;
    answer = i;
    mask = answer/2;
    while (mask > 0)
    {
        answer = answer ^ mask;
        mask = mask / 2;
    }
    return(answer);
}
```
Block-Oriented Algorithm

\[ A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \]
\[ B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \]
\[ C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \]
\[ \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \]

- Use block-oriented distribution introduced for shared memory multiprocessors.
  Block-matrix multiplication is analogous to scalar matrix multiplication.

- Use staggering technique introduced for 2D SIMD mesh.
  Rotation along rows and columns.

- Perform the SIMD matrix multiplication algorithm on whole submatrices.
  Submatrices are multiplied and shifted.
Analysis of Algorithm

$n^2$ matrix, $p$ processors.

- **Row/Column-oriented**
  - Computation: $n^2/p \times n/p = n^3/p^2$.
  - Communication: $2(\lambda + \beta n^2/p)$
  - $p$ iterations.

- **Block-oriented (staggering ignored)**
  - Computation: $(n/\sqrt{p})^3 = n^3/(p\sqrt{p})$.
  - Communication: $4(\lambda + \beta n^2/p)$
  - $\sqrt{p}$ iterations.

- **Comparison**
  
  
  $2p(\lambda + \beta n^2/p) > 4\sqrt{p}(\lambda + \beta n^2/p)$

  1. $p > 2\sqrt{p}$
  2. $\sqrt{p} > 2$

  True for all $p > 4$.

*Also including staggering, for larger $p$ the block-oriented algorithm performs better!*