Parallel Program Design Patterns and Strategies

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University
Data Decomposition

Suppose you are doing image processing operations on a grid of pixels using 4 cores. Each pixel needs to see its neighboring pixels to get the job done. How would you distribute the pixels among the 4 cores?

How we distribute data and then communicate among the distributions is called a Design Pattern.
Foster’s Methodology: PCAM(R)

**Partition** the Problem
Think about how to break the problem up into its fundamental units of computing

Examine the Required **Communication**
- **Local:** each task communicates with other tasks within a core – hopefully often
- **Global:** each task communicates with a large number of other tasks between cores – hopefully seldom

**Agglomerate (or Aggregate)**
Combine the small partitioned tasks into larger tasks

**Map**
Assign the larger tasks to cores or threads

**Reduce**
Combine multi-results into one result

Reference: Ian Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995
Types of Parallel Communications

- Thread-to-Thread

- Broadcast

- Reduction

- Scatter

- Gather
PCAMR Rules of Thumb

1. Focus effort on the most time-consuming computation
2. Focus effort on whatever data is accessed most frequently
3. Focus effort on maximizing the $Compute : Communicate$ Ratio
4. Use agglomeration to reduce communication by increasing locality
5. If agglomeration replicates data, be sure this does not affect the scalability of the algorithm by restricting the range of problem sizes and processor costs
6. Does the number of tasks scale with the problem size? (Not the size of each task!)
7. Place tasks that can execute concurrently on different cores
8. Place tasks that communicate frequently on the same core to increase locality
9. Be sure the Manager is not a bottleneck
10. If you are using cyclic or probabilistic load balancing, be sure you have enough tasks to keep everyone busy
Paradigms for Task Scheduling / Mapping

Decentralized (Peer)

Input  

Output

“Peer-threads”
Paradigms for Task Scheduling / Mapping

Manager / workers

Input → "Manager-thread" → "Worker-threads" → Output
Paradigms for Task Scheduling / Mapping

Map-Reduce

Input → "Map-thread" → "Worker-threads" → "Accumulate-thread" → Output
Paradigms for Task Scheduling / Mapping

Pipeline

Requires some sort of queue between the stages
Load Balancing Strategies: Assigning Portions of the Overall Task to the Threads

Recursive Equal Bisection

Recursive Unequal Bisection

Subdivisions don’t necessarily have to be equal in size

Local algorithms

Each core checks its load against its neighboring cores and adjusts what it is handling
Load Balancing Strategies:  
Assigning Portions of the Overall Task to the Threads

Cyclic mappings

If there are $N$ cores, allocate every $N$th task to a particular core. This is like using \texttt{chunksize} in \texttt{omp parallel for}.

Probabilistic methods

Allocate each task to a randomly-chosen core.
Design Patterns

1. Replicating computation

2. Red / Black (Even / Odd)

3. Divide-and-Conquer (Reduction)

4. Block Scheduling
#include <algorithm>

for( int i = 0; i < NUMN; i++ )
{
    int first = i % 2; // 0 if i is 0, 2, 4, ...
                        // 1 if i is 1, 3, 5, ...

    #pragma omp parallel for default(none),shared(A,first)
    for( int j = first; j < NUMN-1; j += 2 )
    {
        {
            std::swap( A[j], A[j+1] );
        }
    }
}
Design Patterns: Divide and Conquer

\[ \sum_{i=0}^{2^N-1} = \sum_{i=0}^{2^{N-1}-1} + \sum_{i=2^{N-1}}^{2^N-1} \]

e.g., \( N = 4 \):

\[ \sum_{i=0}^{15} \]

\[ \sum_{i=0}^{7} + \sum_{i=8}^{15} \]

\[ \sum_{i=0}^{3} + \sum_{i=4}^{7} + \sum_{i=8}^{11} + \sum_{i=12}^{15} \]

\[ \sum_{i=0}^{1} + \sum_{i=2}^{3} + \sum_{i=4}^{5} + \sum_{i=6}^{7} + \sum_{i=8}^{9} + \sum_{i=10}^{11} + \sum_{i=12}^{13} + \sum_{i=14}^{15} \]
Design Patterns: Block Scheduling

Example: A diagonally-dominant matrix solution

- Break the problem into blocks
- Solve within the block
- Handle borders separately after a Barrier

\[
\begin{bmatrix}
0 & 0 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
? \\
?
\end{bmatrix} = \begin{bmatrix}
? \\
?
\end{bmatrix}
\]

\[
\begin{bmatrix}
? \\
?
\end{bmatrix} \begin{bmatrix}
? \\
?
\end{bmatrix} = \begin{bmatrix}
? \\
?
\end{bmatrix}
\]

\[
\begin{bmatrix}
? \\
?
\end{bmatrix} \begin{bmatrix}
? \\
?
\end{bmatrix} = \begin{bmatrix}
? \\
?
\end{bmatrix}
\]

Barrier
Share results across boundaries
Another Block Schedule Example:
1D Heat Transfer Equation

\[ \rho C \frac{\partial T}{\partial t} = k \left( \frac{\partial^2 T}{\partial x^2} \right) \]

\[ \frac{\Delta T}{\Delta t} = \frac{k}{\rho C} \left( \frac{\Delta^2 T}{\Delta x^2} \right) \]

\[ \Delta T_i = \left( \frac{k}{\rho C} \right) \left( \frac{T_{i-1} - 2T_i + T_{i+1}}{(\Delta x)^2} \right) \Delta t \]
Partition: 1D Domain (Data) Decomposition

1D Block

1D Cyclic

1D Cyclic
Communication, Agglomeration, Mapping: 1D Compute-to-Communicate Ratio

Intracore computing

Intercore communication

Compute : Communicate ratio = N : 2

where N is the number of compute cells per core
1D Domain (Data) Decomposition

Compute : Communicate = 4 : 2

Compute : Communicate ratio = 2 : 2

Compute : Communicate ratio = 1 : 2
Performance as a Function of Number of Nodes

MegaNodes Computed Per Second vs. # of Nodes to Compute
Performance as a Function of Number of Threads

The graph illustrates the performance (in MegaNodes Computed Per Second) as a function of the number of threads for different numbers of nodes. The performance peaks at a certain number of threads for each number of nodes, after which it starts to decline. The number of nodes used for each test is indicated by different line styles and colors:

- 8192 (blue diamonds)
- 6144 (red squares)
- 4094 (green triangles)
- 2048 (purple crosses)
- 1024 (teal asterisks)
- 512 (orange circles)
- 256 (light blue stars)

The peak performance is observed at around 8 threads for 8192 nodes, 4 threads for 6144 nodes, and so on, indicating an optimal number of threads for each node count.
2D Heat Transfer Equation

\[
\rho C \frac{\partial T}{\partial t} = k \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)
\]

\[
\Delta T_{i,j} = \left( \frac{k}{\rho C} \right) \left( \frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{(\Delta x)^2} + \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{(\Delta y)^2} \right) \Delta t
\]
2D Domain (Data) Decomposition

- 2D Block, Block
- 2D Block, Cyclic
- 2D Cyclic, Block
- 2D Cyclic, Cyclic
The Decomposition Order Matters (think cache)

float Array[A][B];

In 2D problems, this is often (but not always) thought of as:

float Array[NY][NX];

0 0
0 1
0 2
0 3
0 ...
0 \(B - 1\)

1 0
1 1
1 2
1 3
1 ...
1 \(B - 1\)

\(A - 1\) 0
\(A - 1\) 1
\(A - 1\) 2
\(A - 1\) 3
\(A - 1\) ...
\(A - 1\) \(B - 1\)
2D Compute-to-Communicate Ratio

Intracore computing

Intercore communication

Compute : Communicate ratio = $N^2 : 4N = N : 4$

where $N$ is the dimension of compute cells per core

The 2D Compute : Communicate ratio is sometimes referred to as Area-to-Perimeter
\[ \rho C \frac{\partial T}{\partial t} = k \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right) \]

\[ \Delta T_{i,j,k} = \left( k \frac{1}{\rho C} \left( \frac{T_{i-1,j,k} - 2T_{i,j,k} + T_{i+1,j,k}}{(\Delta x)^2} + \frac{T_{i,j-1,k} - 2T_{i,j,k} + T_{i,j+1,k}}{(\Delta y)^2} + \frac{T_{i,j,k-1} - 2T_{i,j,k} + T_{i,j,k+1}}{(\Delta z)^2} \right) \right) \Delta t \]

\[ \frac{\Delta T}{\Delta t} = \frac{k}{\rho C} \left( \frac{\Delta^2 T}{\Delta x^2} + \frac{\Delta^2 T}{\Delta y^2} + \frac{\Delta^2 T}{\Delta z^2} \right) \]
3D Domain (Data) Decomposition

3D Block, *, *

3D *,Block, *

3D **,Block

3D Cyclic, *, *

3D *,Cyclic, *

3D **,Cyclic
3D Domain (Data) Decomposition

3D Cyclic, Cyclic, Cyclic

3D Cyclic, Cyclic, Cyclic
The Decomposition Order Matters (think cache)

float Array[A][B][C];

In 3D problems, this is often (but not always) thought of as:

float Array[NZ][NY][NX];
3D Compute-to-Communicate Ratio

Compute : Communicate ratio = $N^3 : 6N^2 = N : 6$

where $N$ is the dimension of compute cells per core

In 3D the Compute : Communicate ratio is sometimes referred to as *Volume-to-Surface*
Functional (or Task) Decomposition

Overall Problem

Thread 0
Thread 1
Thread 2
Thread 3
Functional (or Task) Decomposition

Credit: Maxis (Sim Park)
Matrix Multiply

The usual approach is multiplying the entire A row * entire B column
This is equivalent to computing a single dot product

Row i of A
\[ \sum \text{A}[i][k] \]

Column j of B
\[ \sum \text{B}[k][j] \]

Element (i,j) of C
\[ \text{C}[i][j] \]

for( i = 0; i < SIZE; i++ )
for( j = 0; j < SIZE; j++ )
for( k = 0; k < SIZE; k++ )

\[ \sum \text{A}[i][k] \times \text{B}[k][j] \times \text{C}[i][j] \]

Problem: Column j of the B matrix is not doing a unit stride
Matrix Multiply

Scalable Universal Matrix Multiply Algorithm (SUMMA)
Entire A row * one element of B row
Equivalent to computing one item in many separate dot products

\[
\text{for}( \ i = 0; \ i < \text{SIZE}; \ i++ ) \\
\text{for}( \ k = 0; \ k < \text{SIZE}; \ k++ ) \\
\quad \text{for}( \ j = 0; \ j < \text{SIZE}; \ j++ ) \\
\quad \quad A[i][k] \times B[k][j] \quad \text{Add to} \quad C[i][j]
\]
Performance vs. Matrix Size (MegaMultiplies / Sec)
Performance vs. Number of Threads (MegaMultiplies / Sec)

- i-j-k
- i-k-j
- j-k-i
- j-i-k
- k-i-j
- k-j-i