Parallel Program Design Patterns and Strategies

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University
Contents

1. Communication Design Patterns
2. Task Mapping Design Patterns
3. Functional Decomposition Design Patterns
4. Data Decomposition Design Patterns

The goal of this section is to look at some of the common design and programming patterns one encounters in parallel programming and to understand some of the nuances one encounters in deciding which to use.
Communications Design Patterns for Parallelism

Thread-to-Thread

Broadcast

Reduction

Scatter

Gather
Task Mapping Design Patterns

Decentralized (Peer)

“Peer-threads”
Task Mapping Design Patterns

Manager / workers

Input

“Manager-thread”

“Worker-threads”

Output
Task Mapping Design Patterns

Map-Reduce

Input → "Map-thread" → "Worker-threads" → "Accumulate-thread" → Output
Task Mapping Design Patterns

Pipeline

Requires some sort of queue between the stages
The Functional (or Task) Decomposition Design Pattern

Overall Problem

Thread 0

Thread 1

Thread 2

Thread 3
The Functional (or Task) Decomposition Design Pattern

Credit: Maxis (Sim Park)
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 **Data Decomposition Design Pattern**.
1. Red / Black (Even / Odd)

2. Divide-and-Conquer (Reduction)

3. Block Decomposition
Red/Black or Even/Odd

Remember our friend, the Bubble Sort

NUMN = 6

<table>
<thead>
<tr>
<th>original</th>
<th>Step #</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 5 5 3 3 1</td>
<td>0</td>
</tr>
<tr>
<td>5 6 3 5 1 3</td>
<td>1</td>
</tr>
<tr>
<td>4 3 6 1 5 2</td>
<td>2</td>
</tr>
<tr>
<td>3 4 1 6 2 5</td>
<td>3</td>
</tr>
<tr>
<td>2 1 4 2 6 4</td>
<td>4</td>
</tr>
<tr>
<td>1 2 2 4 4 6</td>
<td>5</td>
</tr>
</tbody>
</table>

#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] );
        }
    }
}
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., \(N = 4\):

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

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

\[
\sum_{i=0}^{3} = \sum_{i=0}^{1} + \sum_{i=2}^{3} \quad \sum_{i=4}^{7} = \sum_{i=4}^{5} + \sum_{i=6}^{7} \quad \sum_{i=8}^{11} = \sum_{i=8}^{9} + \sum_{i=10}^{11} \quad \sum_{i=12}^{15} = \sum_{i=12}^{13} + \sum_{i=14}^{15}
\]
Load Balancing Strategies for Divide-and-Conquer

Recursive Equal Bisection

Recursive Unequal Bisection
Subdivisions don’t necessarily have to be the same size
Block Decomposition: 
1D Heat Transfer Example

You have a steel bar. Each section of the bar starts out at a different temperature. There are no incoming heat sources or outgoing heat sinks (i.e., ignore boundary conditions). Ready, go! How do the temperatures change over time?

The fundamental differential equation here is:

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

where:
- \( \rho \) is the density in kg/m\(^3\)
- \( C \) is the specific heat capacity measured in Joules / (kg⋅°K)
- \( k \) is the coefficient of thermal conductivity measured in Watts / (meter⋅°K)

(These units work because a Watt is a Joule/second.)

In English, this all means that temperatures, left to themselves, try to even out. The greater the temperature differential, the faster the evening-out process goes.
As a side note: the quantity $k/(\rho C)$ has the unlikely units of m$^2$/sec)
1D Domain (Data) Decomposition: Partitioning Strategies

1D Block

1D Cyclic

1D Cyclic
Should you allocate the data as one large global-memory block (i.e., shared)?

Or, should you allocate it as separate blocks, each local to its own core (i.e., private)?

Does it matter?

Stay tuned! This will be discussed when we get to the Cache section.
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 Compute-to-Communicate Ratio

Compute : Communicate = 4 : 2

Compute : Communicate ratio = 2 : 2

Compute : Communicate ratio = 1 : 2
Which Compute-to-Communicate Ratio is Best?

Compute : Communicate = 4 : 2

Compute : Communicate ratio = 2 : 2

Compute : Communicate ratio = 1 : 2

Think if it as a *Goldilocks and the Three Bears* sort of thing. :-)

Too little compute:communicate and you are spending all your time sharing data values across threads and doing too little computing.

Too much compute:communicate and you are not spreading out your problem among enough threads to get good parallelism.

It’s difficult to find the “sweet spot” without running experiments.
Performance as a Function of Number of Nodes

MegaNodes Computed Per Second

# of Nodes to Compute

# of Threads
Performance as a Function of Number of Threads

MegaNodes Computed Per Second vs. Number of Threads

- 8192 nodes
- 6144 nodes
- 4094 nodes
- 2048 nodes
- 1024 nodes
- 512 nodes
- 256 nodes

Number of Nodes vs. Number of Threads

Oregon State University
Computer Graphics

mjb – March 29, 2016
2D Heat Transfer Equation

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

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

\[ \Delta T_{i,j} = \left( \frac{k}{\rho C} \right) \left( \frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{(\Delta x)^2} + \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{(\Delta y)^2} \right) \Delta t \]
In addition to the issues of size of the compute block, you also have issues of direction.
Direction Issue: 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];
2D Compute-to-Communicate Ratio

Intracore computing

Intercore communication

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

where N is the dimension of compute nodes per core

The 2D Compute : Communicate ratio is sometimes referred to as Area-to-Perimeter
3D Heat Transfer Equation

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

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

\[ \Delta T = \frac{k}{\rho C} \left( \frac{\Delta^2 T}{\Delta x^2} + \frac{\Delta^2 T}{\Delta y^2} + \frac{\Delta^2 T}{\Delta z^2} \right) \]
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
Direction Issue: 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 nodes per core

---

In 3D the Compute : Communicate ratio is sometimes referred to as *Volume-to-Surface*
An Example of Where Decomposition Order Really Matters: Matrix Multiply

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

\[
\sum_{i} \sum_{j} A[i][k] \times B[k][j] \rightarrow C[i][j]
\]

**Problem:** Column j of the B matrix is not doing a unit stride
An Example of Where Decomposition Order Really Matters: 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

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

- i-k-j
- j-i-k
- k-j-i
- j-j-k
- j-k-i
- k-i-j
Performance vs. Number of Threads (MegaMultiplies / Sec)

i-k-j

j-i-k

k-j-i

j-j-k

j-k-i

k-i-j