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)

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

2. **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

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

4. **Map**
   - Assign the larger tasks to cores or threads

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

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" → "Accumulate-thread" → Output

Load Balancing Strategies:
Assigning Portions of the Overall Task to the Threads

Recursive Equal Bisection
Recursive Unequal Bisection

Local algorithms
Each core checks its load against its neighboring cores and adjusts what it is handling

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
Design Patterns: Divide and Conquer

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

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

\[ \sum_{i=0}^{7} \]

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

\[ \sum_{i=0}^{9} \]

\[ \sum_{i=0}^{7} \]

\[ \sum_{i=0}^{13} \]

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

Another Block Schedule Example: 1D Heat Transfer Equation

\[ \rho c \frac{\partial T}{\partial t} = k \frac{\partial^2 T}{\partial x^2} \]

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

\[ \Delta T = \left( \frac{h}{\rho C} \right) \left( \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} \right) \]

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

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

Performance as a Function of Number of Threads

2D Heat Transfer Equation

\[
\frac{\partial^2 T}{\partial x^2} + k \frac{\partial^2 T}{\partial y^2} = \frac{\Delta T}{\Delta x \Delta y}
\]

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

The Decomposition Order Matters (think cache)

float Array[N][M];

B

A

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

float Array[N][M];
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

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

```
float* Array[ANZ][NY][NX];
```

```
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

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 * Column j of B = Element (i,j) of C

\[ \sum_{k=0}^{\text{SIZE}} A[i][k] \cdot B[k][j] \rightarrow C[i][j] \]

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

Row i of A * Row k of B = Element (i,j) of C

\[ \sum_{j=0}^{\text{SIZE}} A[i][k] \cdot B[k][j] \rightarrow C[i][j] \]

Performance vs. Matrix Size (MegaMultiplies / Sec)

Performance vs. Number of Threads (MegaMultiplies / Sec)