Parallel Systems

Part 2: Parallel Programming Models and Parallelization Phases

foils by M. Allen, T. Fahringer, M. Gerndt, M. Quinn, B. Wilkinson
Programming Models (1)

- The programming model
  - determines the basic concepts of the parallel implementation and
  - abstracts from the hardware as well as from the programming language or API.

- The names used for programming models differ in the literature.
Programming Models (2)

- **Sequential Model**: The sequential program is automatically parallelized.
  - Advantage: familiar programming model
  - Disadvantage: limitations in compiler analysis

- **Message Passing Model**: The application consists of a set of processes with separate address spaces. The processes exchange messages by explicit send/receive operations.
  - Advantage: full control of performance aspects
  - Disadvantage: complex programming

Assembly programming for parallel architectures!
Message Passing Programming Model

Node A
- process
- send (Y)

Node B
- process
- receive (Y')

Y -> message -> Y'

Process
Memory
Programming Models (3)

- **Shared Memory Model**: Application consists of parallel threads, accessing shared data structures protected by synchronization operations.
  - **Thread-Based SM Model**: Explicit programming of cooperating threads.
    - Advantage: portability and shared memory
    - Disadvantage: complex synchronization
  - **Directive-Based SM Model**: High-level parallel operations, e.g. parallel loops with implicit synchronization.
    - Advantage: Easier to program
    - Disadvantage: Difficult to control performance and locality
  - **Remote Memory Access Model**: Parallel threads accessing mainly private data structures. Shared data structures are accessed via special operations.
    - Advantage: Can be easily combined with message passing
    - Disadvantage: Complex programming
Shared Memory Programming Model

Processor

Memory

Thread (Process)

read(X)

Thread (Process)

write(X)

System

Shared variable

X
Programming Models (4)

Data Parallel Programming Model: Synchronized execution of parallel operations on large distributed data structures.

– Advantage: great potential for massive parallelism, can be applied to shared memory and distributed memory programming model

– Disadvantage: limited to data parallelism
Functional Programming Model: Parallel or sequential execution of different functional units of a program connected through data and control flow.

- Advantages:
  - easy to understand,
  - can be applied to shared memory and distributed memory programming model
  - load balancing becomes difficult
  - good potential for function calls inside of loops

- Disadvantage:
  - limited potential in most programs
Concepts of Parallel Programming

- **Task**: arbitrary piece of work performed by a single process or thread

- **Thread**: is an abstract entity as part of a process that executes tasks. Defines a unit for scheduling.

- **Process**: is an active entity with resources that executes tasks. Defines a unit for resources.

- **Processor/Core**: is a physical resource executing processes
Phases in the Parallelization Process

1. Decomposition
2. Assignment
3. Orchestration
4. Mapping

Partitioning

Sequential computation → Tasks → Processes → Parallel program → Processors

Processors: P0, P1, P2, P3
Decomposition

- Dividing computation and data into pieces
- Domain decomposition
  - Divide data into pieces
  - Determine how to associate computations with the data
- Functional decomposition
  - Divide computation into pieces
  - Determine how to associate data with the computations
Example Domain Decompositions

Data Structure

Primitive Tasks

1-D

2-D

3-D
Example Functional Decomposition

1. Acquire Patient Images
2. Register Images
3. Determine Image Locations
4. Track Position of Instruments
5. Display Image
Simulate Flooding of the Danube with the Grid

Applications are complex and dynamically constructed from services. Different organisations cooperate to predict the flooding behavior of the Danube by using Grid sensors, computing and data storage resources as well as modeling and simulation services.
Functional Decomposition

- Breaking the computation into a collection of tasks
  - Tasks may be of variable length
  - Tasks require data to be executed
  - Tasks may become available dynamically

- Amdahl’s law: If $s$ is the execution time for inherently sequential computations, the speedup is limited by
  \[
  speedup(p) = \frac{time(1)}{time(p)} = \frac{time(1)}{s + (parallel\_time(p))} \leq \frac{time(1)}{s}
  \]

- If 20% of the sequential execution time is in sequential regions, the speedup is limited to 5 independent of the number of processors.
Example
Data Parallelism

parallel operation
$n^2$ parallel tasks

reduction
$n^2$ sequential tasks

\[
speedup(p) = \frac{2n^2}{\frac{n^2}{p} + n^2} = \frac{2p}{p + 1} \leq 2
\]
Example
Data Parallelism

parallel operation

\[ n^2 \text{ parallel tasks} \]

reduction

\[ p \text{ parallel tasks} \]
with \( n^2/p \) operations

\[ + \]

\[ p \text{ sequential tasks} \]

\[
\text{speedup}(p) = \frac{2n^2}{\frac{n^2}{p} + \frac{n^2}{p} + p} \approx \frac{n^2}{p} \cdot p
\]
Data Parallelism vs Functional Parallelism

Data parallelism

- The same operations are executed in parallel for the elements of large data structures, e.g. arrays.
- Tasks are the operations on each individual element or on subsets of the elements.
- Whether tasks are of same length or variable length depends on the application. Many data parallel applications have tasks of same length.
Example: Data Parallelism

```
for (i=0; i<n; i++)
    a[i] = b[i] + c[i]
```
Example: Data Parallelism, Variable Length

```
for (i=0; i<n; i++)
    for (j=0; j<=i; j++)
        a[i] = a[i] + b[i][j]
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

+=

Task0
Task1
Task2
Task3
Task4
Task5
Task6
Data Parallelism vs Functional Parallelism

- Functional parallelism
  - Entirely different calculations can be performed concurrently on either the same or different data.
  - The tasks are usually specified via different functions or different code regions.
  - The degree of available functional parallelism is usually modest.
  - Tasks are of different length in most cases.
Example: Functional Parallelism

- The functions or statements can be executed in parallel.

- These are different operations $\rightarrow$ functional parallelism
Phases in the Parallelization Process

Partitioning

Decomposition

Assignment

Orchestration

Mapping

Sequential computation → Tasks → Processes → Parallel program → Processors
Assignment

Assignment means specifying the mechanism by which tasks will be distributed onto processes.

- Goals:
  - Balance workload
  - Reduce interprocess communication
  - Reduce assignment overhead

- Assignment time
  - Static: fixed assignment during compilation or program creation
  - Dynamic: adaptive assignment during execution
Load imbalance

Ruler Result

\[ \frac{28509800 \text{ clocks}}{1000000} = 190.095 \text{ ms} \]
Example: Balance Workload (1)

```
for (i=0;i<n;i++)
    a[i] = b[i] + c[i]
```
Example: Balance Workload (2)

- **Static Load Balancing**
  - 2 different assignments
- **Dynamic Load Balancing**
  - postpone assignment until execution time

```c
for (i=0; i<n; i++)
    for (j=0; j<i; j++)
        a[i] = a[i] + b[i][j]
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+-----+-----+-----+-----+-----+-----+-----|
|      |      |      |      |      |      |      |
```

Task assignments:
- Task0: ◇ ◇
- Task1: ● ● ● ●
- Task2: ◆ ◆ ◆
- Task3: ◆◆◆◆
- Task4: ◆◆◆◆
- Task5: ◄ ◄ ◄ ◄
- Task6: ◆ ◆ ◆ ◆ ◆ ◆
Phases in the Parallelization Process

- Decomposition
- Assignment
- Orchestration
- Mapping

Sequential computation → Tasks → Processes → Parallel program → Processors
Orchestration

- Implementation in a given programming model and programming language

- Means for
  - Naming and accessing data
  - Exchanging data
  - Synchronization

- Questions
  - How to organize data structures?
  - How to schedule assigned tasks to improve locality?
  - Whether to communicate in large or small messages?

- Performance goal
  - Reduction of communication and synchronization overhead
  - Reduction of parallelization overhead
  - Reduction of idle time
  - Ideal load balancing

- Sometimes goals can be conflicting
  - reduction of communication versus load balancing
  - Reduction of parallelization overhead versus load balancing
Example: Loop (1)

```c
#pragma omp for schedule(static)
for (i=0; i<n; i++)
    a[i] = b[i] + c[i]
```
Example: Loop Cyclic Distribution (2)

```c
#pragma omp for schedule(static,2)
for (i=0; i<n; i++)
    for (j=0; j<i; j++)
        a[i]=a[i]+b[i][j]
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>+=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Task0  Task1  Task2  Task3  Task4  Task5  Task6
Example: Dynamic Distribution

(3)

```c
#pragma omp for schedule(dynamic)
for (i=0; i<n; i++)
    for (j=0; j<i; j++)
        a[i] = a[i] + b[i][j]
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>++</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Task0:  
- Task1:  
- Task2:  
- Task3:  
- Task4:  
- Task5:  
- Task6:  

Diagram shows the distribution of tasks among variables a[0] to a[6].
Phases in the Parallelization Process

- **Decomposition**
- **Assignment**
- **Orchestration**
- **Mapping**

Sequential computation → Tasks → Processes → Parallel program → Processors
Mapping (1)

- Mapping processes to processors
- Done by the program/library and/or operating system
- Shared memory system: mapping done by operating system
- Distributed memory system: mapping done by user or runtime library
Mapping (2)

Goal
- If there are more processes than processors: put multiple related processes on the same processor.
- This may also be an option for heavily interacting processes no matter how many processors are available.
- Exploit locality in network topology.
  - place processes close to needed data
  - place processes close to those processes that require interaction

Conflicting goals of mapping
- Maximize processor utilization
- Minimize interprocessor communication
Mapping Example
Optimal Mapping

- Finding optimal mapping is NP-hard
- Must rely on heuristics
Mapping Decision Tree

➢ Static number of tasks
  o Structured communication
    • Constant computation time per task
      – Agglomerate tasks to minimize communication
      – Divide tasks by the number of processors
    • Variable computation time per task
      – Cyclically map tasks to processors
  o Unstructured communication
    • Use a static load balancing algorithm

➢ Dynamic number of tasks
  o Frequent communications between tasks
    • Use a dynamic load balancing algorithm
  o Many short-lived tasks
    • Use a dynamic load balancing algorithm
# Performance Goals

<table>
<thead>
<tr>
<th>Step</th>
<th>Architecture-Dependent?</th>
<th>Major Performance Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decomposition</strong></td>
<td>Mostly No</td>
<td>➢ Expose enough concurrency</td>
</tr>
<tr>
<td><strong>Assignment</strong></td>
<td>Mostly no</td>
<td>➢ Balance workload</td>
</tr>
<tr>
<td></td>
<td></td>
<td>➢ Reduce communication volume</td>
</tr>
<tr>
<td><strong>Orchestration</strong></td>
<td>Should not be</td>
<td>➢ Reduce unnecessary communication via data locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>➢ Reduce communication and synchronization costs as seen by the processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>➢ Reduce serialization at shared resources</td>
</tr>
<tr>
<td><strong>Mapping</strong></td>
<td>Yes</td>
<td>➢ Put related processes on the same processor if necessary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>➢ Exploit locality in network topology</td>
</tr>
</tbody>
</table>
Programming Language or API

- A parallel programming language or API implements a programming model.

  - Message Passing
    - Message Passing Interface (MPI)
    - Parallel Virtual Machine (PVM)
  
  - Shared Memory:
    - Thread-based: Java Threads, Posix Threads
    - Directive-based: OpenMP
    - Remote Memory Access: MPI single sided communication, Shmem library on CRAY T3E
Parallelization Strategies

- Frequently used patterns for parallel applications:
  - Single Program Multiple Data - SPMD
  - Embarrassingly Parallel
  - Master / Slave
  - Work Pool
  - Divide and Conquer
  - Pipeline
  - Competition
Application Structure: Single Program Multiple Data

- Single program is executed in a replicated fashion.
- Processes or threads execute the same operations on different data.
- Loosely-synchronous: Sequence of phases of computation and communication/synchronization.
SPMD: Shared Memory Version

- All processes execute the same program and have direct access to a shared address space.
- A program is a sequence of one or more sections.
- A section may be serial, parallel or replicate:
  - A **serial section** is executed by exactly one process. The first process arriving at a serial section is assigned to it; all others skip it.
  - A **parallel section** is expressed e.g. a doall loop. All iterations can be executed in parallel, without synchronization.
  - A **replicate section** is executed by all processes.
- The model provides **barrier synchronization**.
This model assumes a distributed-memory multiprocessing (DMMP). All processes are created when a program is initiated, and terminate at its end.

- Each process operates in a separate address space.
- Data are distributed to processors
- All processes execute the same program (but operate on separate sets of data)

The basic **compilation scheme** transforms a program in three steps:

- Distribute data to processors
- Enforce Owner Computes Paradigm: On each processor, perform exactly those computations which assign values to local data
- Insert communication for all accesses to non-local data.
Parallelization Strategy
distributed data

- Data parallelism
- SPMD Parallelism
- all processors execute the same program on different sets of data

User
  - writes sequential program: global addressing scheme
  - specifies data distribution

Compiler should (in practice often the user)
  - perform work distribution
  - insert communication
  - generate parallel program
Parallelization Strategy
replicated data

- every process has a copy of replicated data
- all updates are done by all processors
- no communication implied
Jacobi Iteration: Sequential

!HPF$ PROCESSORS PROC(P, P)
!HPF$ DISTRIBUTE (BLOCK, BLOCK) :: UNEW, UOLD
REAL UNEW(1:N,1:N), UOLD(0:N+1,0:N+1)
REAL (*,*) UOLD

DO 40 J=1,N
   DO 40 I=1,N
      UNEW(I,J) = 0.25*(UOLD(I-1,J) + UOLD(I+1,J) +
                       UOLD(I,J-1) + UOLD(I,J+1))
   40 CONTINUE

WRITE (*,*) UNEW

STOP
END
Jacobi algorithm

\[
X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8}. \quad (2.1)
\]

(Images: Ian Foster)
Jacobi overlapping matrix
Jacobi overlapping matrix
Jacobi Iteration: Message Passing

C PROCESSOR STRUCTURE PROC(P, P)

C CODE FOR PROCESSOR (P1, P2)

PARAMETER (P = 4, N = 256)
PARAMETER (LEN = N/P)
IMPLICIT INTEGER (I-P)
IMPLICIT REAL (A-H,Q-Z)

C DECLARE LOCAL ARRAYS TOGETHER WITH OVERLAP AREA

C    local declaration

REAL UOLD(0 :LEN+1,0 :LEN+1), UNEW(1 :LEN,1 :LEN)

C DATA OWNED LOCALLY IS UOLD (1 :LEN,1 :LEN),
UNEW(1 :LEN,1 :LEN)
IF (P1.GT.1) SEND (UOLD(1,1 :LEN)) TO PROC(P1-1,P2)
IF (P1.LT.P) SEND (UOLD(LEN,1 :LEN)) TO PROC(P1+1,P2)
IF (P2.GT.1) SEND (UOLD(1 :LEN,1)) TO PROC(P1,P2-1)
IF (P2.LT.P) SEND (UOLD(1 :LEN,LEN)) TO PROC(P1,P2+1)

**Receive data from other processors and assign to overlap area**

IF (P1.GT.1) RECEIVE (UOLD(0,1 :LEN)) FROM PROC(P1-1,P2)
IF (P1.LT.P) RECEIVE (UOLD(LEN+1,1 :LEN)) FROM PROC(P1+1,P2)
IF (P2.GT.1) RECEIVE (UOLD(1 :LEN,0)) FROM PROC(P1,P2-1)
IF (P2.LT.P) RECEIVE (UOLD(1 :LEN,LEN+1)) FROM PROC(P1,P2+1)

**Send data to other processors**

IF (P1.GT.1) SEND (UOLD(1,1 :LEN)) TO PROC(P1-1,P2)
IF (P1.LT.P) SEND (UOLD(LEN,1 :LEN)) TO PROC(P1+1,P2)
IF (P2.GT.1) SEND (UOLD(1 :LEN,1)) TO PROC(P1,P2-1)
IF (P2.LT.P) SEND (UOLD(1 :LEN,LEN)) TO PROC(P1,P2+1)

**Compute new values on local data**

```
DO 50 1, LEN
   DO 50 1, LEN
      UNEW(I,J) = 0.25*(UOLD(I-1,J) + UOLD(I+1,J) +
                        UOLD(I,J-1)+UOLD(I,J+1))
   50 CONTINUE
SEND UNEW (1 :LEN,1 :LEN) TO HOST
STOP
END
```
C PROCESSOR STRUCTURE PROC(P,P) = PROC(1 :P,1 :P)

REAL UNEW(1:N,1:N)

C ASSUME P DIVIDES N
C GET BOUNDARIES OF LOCAL BLOCK OF UNEW FOR PROC(P1,P2)
C THESE BLOCKS ARE OWNED BY THE CORRESPONDING NODE PROCESSOR

DO 15 P1=1,P
    LB1(P1)=(P1-1)*LEN+1
    IUB1(P1)=P1*LEN
15  CONTINUE

DO 25 P2=1,P
    LB2(P2)=(P2-1)*LEN+1
    IUB2(P2)=P2*LEN
25  CONTINUE

DO 45 P1=1,P
    DO 45 P2=1,P
        RECEIVE UNEW(LB1(P1) :IUB1(P1),LB2(P2) :IUB2(P2))
            FROM PROC(P1,P2)
45  CONTINUE

Part of Host Code: Jacobi Relaxation
Application Structure: Embarrassingly Parallel

- Multiple processes are spawned at the beginning.
- They execute totally independent of each other.
- Application terminates after all processes terminated.
Application Structure: Master / Slave

- One process executes as a master. It distributes tasks to the slaves and receives the results from the slaves.
- Slaves execute the assigned tasks usually independent of the other slaves.
- Frequently used on heterogeneous systems with external load.
Application Structure: Work Pool

- Processes fetch tasks from a pool and insert new tasks into the pool.
- Pool requires synchronization.
- Large parallel machines require a distributed work pool.
- Can leads to better load balancing than master slave.

Diagram:
- Work Pool
- P0
- P1
- P2
Application Structure: Divide and Conquer

- Recursive partitioning of tasks and collection of results
- Problems:
  - load balancing
  - least granularity
- Used on systems with background load
Application Structure: Pipeline

➢ Examples
  o Different functions are applied to data: functional decomposition
  o Parallel execution of functions for different data.
  o Signal and image processing
  o Groundwater flow, flow of pollutants, visualization
  o Almost no example of high parallelism
We would like to prepare and mail 1000 envelopes each containing a document of 4 pages to members of an association.

• At what intervals, do we see a new envelope prepared for mailing?
  \[ \text{Max}(T_1, T_2, \ldots, T_k) = T_{\text{max}} = 15 \text{ sec.} \]

• What is the total time to get \( N \) envelopes prepared?
  \[ \text{Time} = \text{Cold\_Start\_Time} + T_{\text{max}} \times (N-1) \approx N \times T_{\text{max}} \]

• What is the total time we would have spent if pipelining is not used?
  \[ N \times \sum_{i} T_i \]
Pipelining: Example (contd.)

How much speedup do we get?

\[
\text{Speedup} = \frac{T_{\text{seq}}}{T_{\text{pipe}}} = \left[ \frac{N \sum_i T_i}{N \cdot T_{\text{max}}} \right] = \frac{\sum_i T_i}{T_{\text{max}}}
\]

Speedup = 35/15 = 7/3

What can you do to maximize the speedup?

(i) Create as many pipeline stages as possible and
(ii) Try to balance the load at each stage, i.e. \( T_1 = T_2 = \ldots = T_k \)
One possible configuration to maximize speedup:

- At what intervals, do we see a new envelope prepared for mailing?
  \[
  \text{Max}(T_1, T_2, \ldots, T_k) = T_{\text{max}} = 5 \text{ sec.}
  \]

- What is the speedup now?
  \[
  \text{Speedup} = \frac{30}{5} = 6 = \text{number of stages in the pipeline!}
  \]
Application Structure: Competition

- Evaluation of multiple solution strategies in parallel.
- It might be unknown which strategy is successful or which one is the fastest.
- With $k$ processors, $k$ strategies can be tested. If one of the additional strategies - not tested in the sequential program - is very fast, the speedup can be more than $k$ (Superlinear speedup)
Example: Equation Solver Kernel

- Solver kernel for a simple partial differential equation.
- Finite difference method
- Grid \((n+2) \times (n+2)\)
- Fixed boundaries
- Interior points are recomputed
  - Mean value of five points in stencil
  - Gauss-Seidel method
    - New values of upper and left point
    - Old values of lower and right point
  - Termination if difference between old and new value is below threshold for all points

\[
\]
Sequential Code

```c
int n;                  /*size of matrix: (n + 2-by-n + 2)*/
float: **A, diff = 0;

main ()
begin
read(n);                /*read input parameter: matrix size*/
A = malloc (a 2-d array of size n + 2 by n + 2 doubles);
initialize(A);          /*initialize the matrix A somehow*/
Solve (A);              /*call the routine to solve equation*/
end main
```
Routine SOLVE

procedure Solve (A)       /*solve the equation system*/
float **A;                /*A is an (n + 2) - by -(n + 2) array*/
begin
int i, j, done = 0;
float diff = 0, temp;
while (!done) do          /*outermost loop over sweeps*/
diff = 0;                  /*initialize maximum difference to 0*/
for i=1 to n do            /*sweep over nonborder points of grid*/
    for j=1 to n do
        temp = A[i,j];       /*save old value of element*/
                        A[i,j+1] + A[i+1,j]); /*compute average*/
        diff += abs(A[i,j] - temp);
    end for
end for
if (diff/(n*n) < TOL) then done = 1;
end while
end procedure
Dependences in Gauss-Seidel

- Dependences prohibit row or column wise parallelization
- Point-wise synchronization
- Parallel execution along anti-diagonals
  - Proportional to n
  - Frequent synchronization, once per anti-diagonal
  - Load imbalance for short anti-diagonals
Relaxing Ordering Constraints

- **Jacobi iterations**
  - Full sweep with old values
  - Much slower convergence → more iterations
  - $N^2$ parallel tasks in each iteration

- **Red-Black iterations**:
  - Checkerboard like coloring scheme
  - Two phases
    - Computation of red points with old black values
    - Computation of black points with new red values
  - Faster convergence than Jacobi but more iterations than Gauss-Seidel
  - Each phase with $n^2/2$ parallel tasks
Decomposition

- $n \times n$ matrix and $p$ processors
- point-based decomposition
  - $n^2$ parallel tasks
  - Each iteration can be assigned to a different process
  - Such assignments would reduce reuse of values
- row-based decomposition
  - $n/p$ coarser parallel tasks
  - All iterations of grid points in a row build a task
  - Leads to reuse in a row computation
- 2-dim block-wise decomposition
  - 2-dim block-wise block-wise decomposition
  - All iterations of grid points in a block build a task
  - Leads to reuse in a block
Assignment (1)

- Tasks are assigned to processes
- Row-based decomposition
  - Row-wise (sets of rows) assignment
    \[
    \text{iteration } i \rightarrow \left\lfloor \frac{i}{(n/p)} \right\rfloor
    \]
  - Cyclic assignment
    \[
    \text{iteration } i \rightarrow i \mod p
    \]
- Communication-to-computation ratio
  - Surface-to-volume ratio
  - Row
    \[
    \frac{2 \ast n}{n \ast \frac{n}{p}} = 2 \ast \frac{p}{n}
    \]
  - Cyclic
    \[
    \frac{2 \ast n \ast \frac{n}{p}}{n \ast \frac{n}{p}} = 2
    \]
Assignment (2)

- 2-dim block-wise decomposition
  - Block-Block assignment
    
    \[
    \text{iteration} (i, j) \rightarrow \left( \left\lfloor \frac{i}{(n/\sqrt{p})} \right\rfloor, \left\lfloor \frac{j}{(n/\sqrt{p})} \right\rfloor \right)
    \]

  - Communication-to-computation ratio
    
    \[
    \frac{4 \frac{n}{\sqrt{p}} \frac{n^2}{p}}{\frac{n^2}{p} n} = 4 \frac{\sqrt{p}}{n}
    \]
Summary Parallel Programming

- Steps in the parallelization process
  - Decomposition, assignment, orchestration, mapping

- Variety of programming models for SM and DM systems
  - Different parallel languages and APIs
  - Trend towards standards (OpenMP, MPI, Posix)

- Data vs functional parallelism
  - Data parallelism: potential for scaling performance for large multiprocessor systems
  - Functional parallelism: easier to locate and program but lack of scaling performance