Parallel and Distributed Computing
CS3006 (BDS-6A)
Lecture 08

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

21 February, 2023

Recursive Decomposition: Quicksort Example

- Once we have selected a pivot value, partitioning places all the elements less than the pivot in the left part of the array and all elements greater than the pivot in the right part of the array and the pivot is in the slot between them.

- The pivot element ends up in the position it retains in the final sorted order

- After partitioning no element flops to the other side of the pivot in the final sorted order

Thus we can sort the elements to the left of the pivot and the right of the pivot independently

Quicksort Pseudocode

```
Quicksort(A, low, high)
    If (low < high)
    pivotLocation = Partition(A, low, high)
    Quicksort(A, low, pivotLocation - 1)
    Quicksort(A, pivotLocation + 1, high)

Partition(A, low, high)
    Pivot = A[low]
    Leftwall = low
    For (i = low +1 to high)
        if (A[i] < pivot)
            leftwall = leftwall + 1
            Swap (A[i], A[leftwall])
    Swap(A[low], A[leftwall])
```

Decomposition Techniques

Data Decomposition

<u>Partitioning *input* data</u>

- In many algorithms, it is not possible or desirable to *partition the output data*.
  - The output may be a *single unknown value*.
  - Such as in case of *finding sum*, *minimum*, *maximum* or *frequencies of a number*.
- It is sometimes possible to *partition the input data*, and then use this partitioning to *induce concurrency*
- A *task is created for each partition of the input data* and this task performs *as much computation as possible using this local data*
- Then *local solutions are combined* to generate a *global solution*

Decomposition Techniques

Partitioning input data

(a) Partitioning the transactions among the tasks

task 1

Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,

Itemsets	Itemset Frequency
A, B, C	1
D, E	2
C, F, G	0
A, E	1
C, D	0
D, K	1
B, C, F	0
C, D, K	0

task 2

Database Transactions
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Itemsets	Itemset Frequency
A, B, C	0
D, E	1
C, F, G	0
A, E	1
C, D	1
D, K	1
B, C, F	0
C, D, K	0

Decomposition Techniques

Data Decomposition

Partitioning *both input and output* data

- Consider the problems where output data-partitioning is possible
- Here, partitioning the input also, can offer additional concurrency
- The next example shows *4-way decomposition of the previous example* based on both input-output partitioning.

Decomposition Techniques

Partitioning *both input and output* data

(b) Partitioning both transactions and frequencies among the tasks

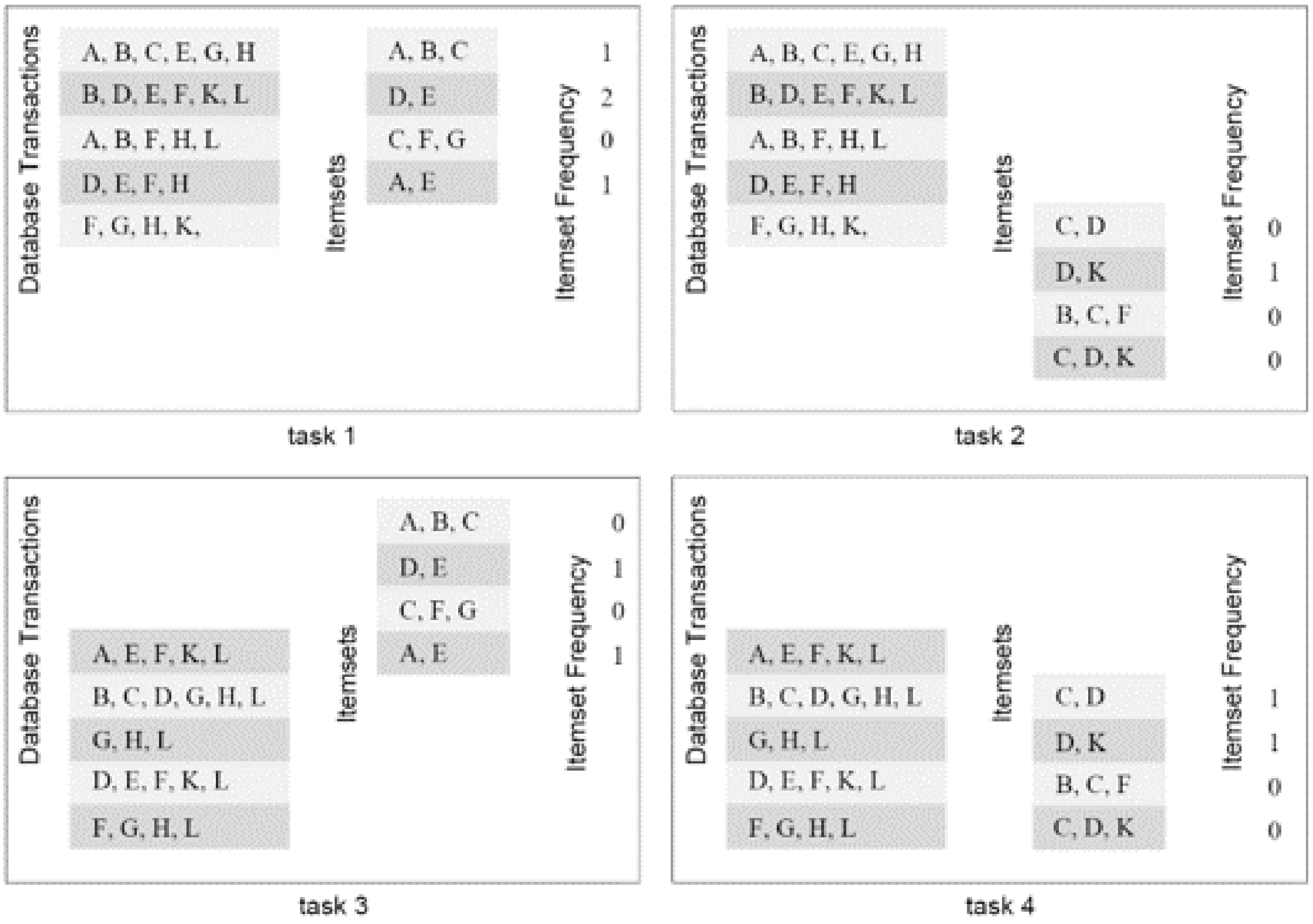


Figure 3.13 Some decompositions for computing itemset frequencies in a transaction database.

Decomposition Techniques

Partitioning *both intermediate data*

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

Task 01: $D_{1,1,1} = A_{1,1}B_{1,1}$
Task 02: $D_{2,1,1} = A_{1,2}B_{2,1}$
Task 03: $D_{1,1,2} = A_{1,1}B_{1,2}$
Task 04: $D_{2,1,2} = A_{1,2}B_{2,2}$
Task 05: $D_{1,2,1} = A_{2,1}B_{1,1}$
Task 06: $D_{2,2,1} = A_{2,2}B_{2,1}$
Task 07: $D_{1,2,2} = A_{2,1}B_{1,2}$
Task 08: $D_{2,2,2} = A_{2,2}B_{2,2}$
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$
Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$
Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$
Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

Figure 3.15 A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.

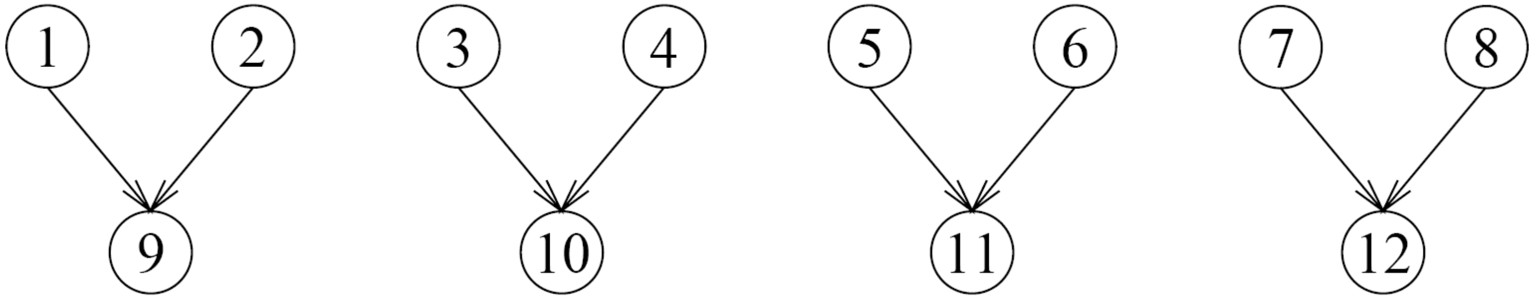


Figure 3.16 The task-dependency graph of the decomposition shown in Figure 3.15.

Decomposition Techniques

3. Exploratory Decomposition

- Specially used to decompose the problems having underlying computation *like search-space exploration*.
- Steps:
  1. Partition the *search space into smaller parts*
  2. Search each one of these parts concurrently, *until the desired solutions* are found.

Decomposition Techniques

3. Exploratory Decomposition

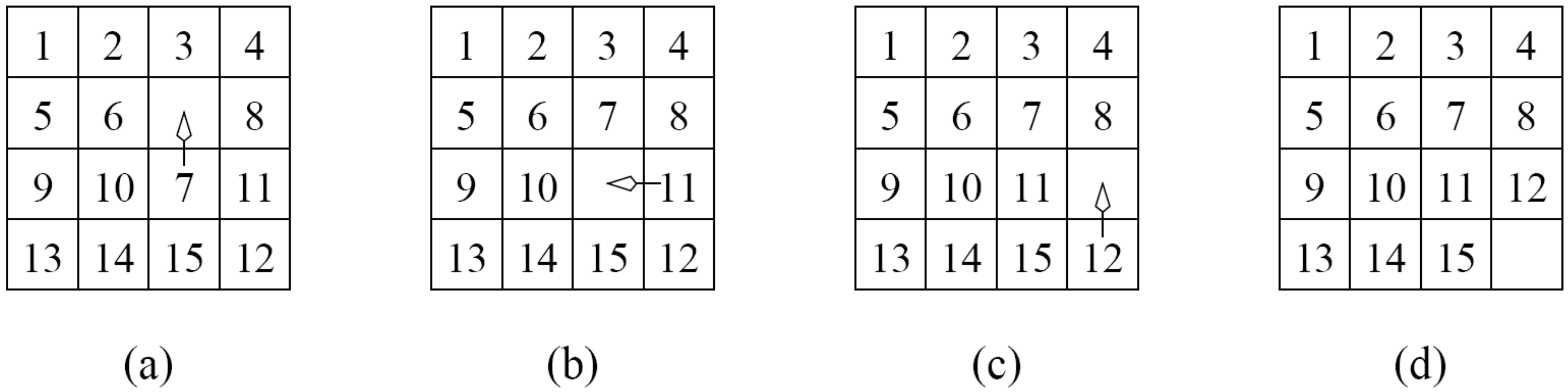


Figure 3.17 A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.

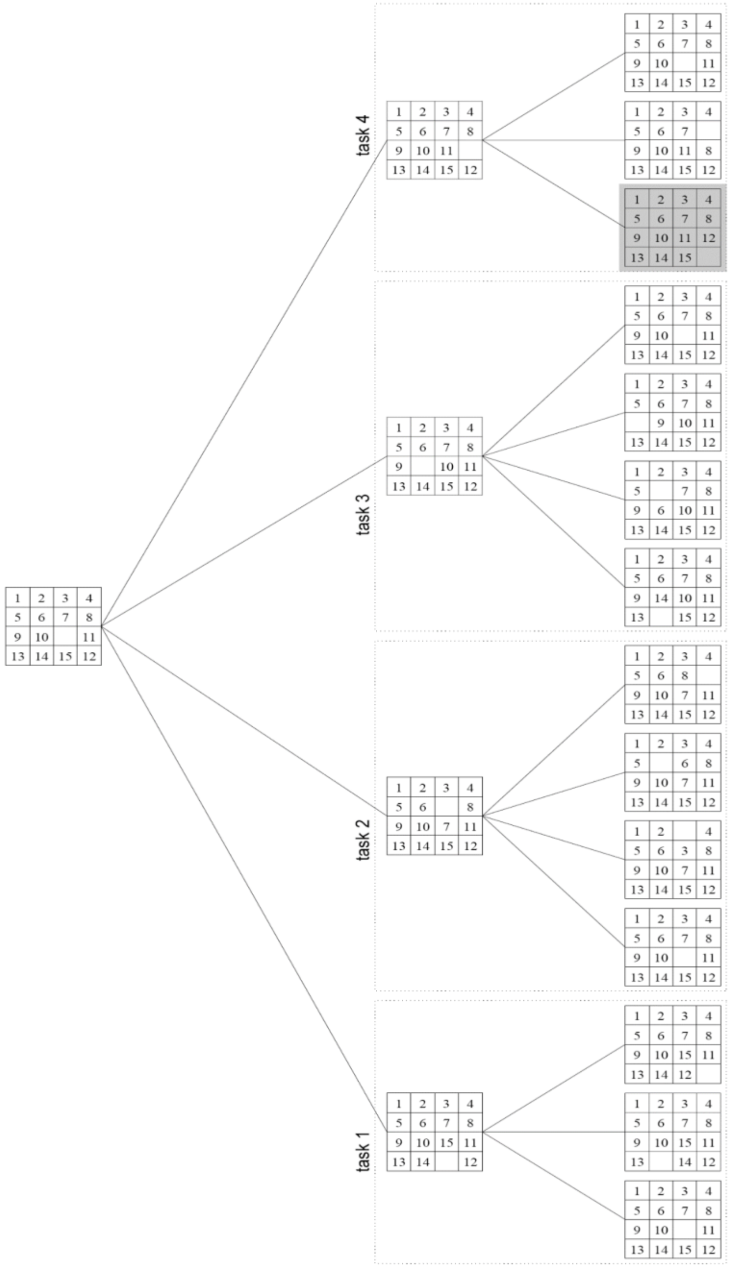
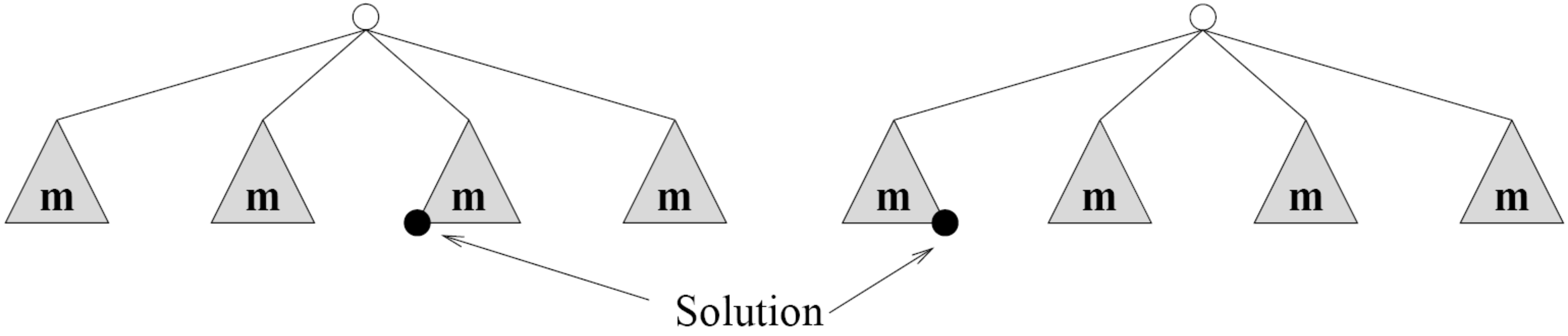


Figure 3.18 The states generated by an instance of the 15-puzzle problem.



Total serial work: 2m+1

Total parallel work: 1

We may also consider this as 4, since each task has done 1 step

(a)

Total serial work: m

Total parallel work: 4m

(b)

Figure 3.19 An illustration of anomalous speedups resulting from exploratory decomposition.

Decomposition Techniques

4. Speculative Decomposition

- Usually used in the problems *where different input values or output of the previous stage* causes many *computationally intensive branches*.
- Speculation is something like *Gamble* or *Risk* or preliminary guess.
- Steps:
  - *Speculate (guess) the output* of previous stage
  - Start performing computations in the *next stage even before the completion of the previous stage*.
  - After the output of the previous stage is available, *if the speculation was correct*, then most of the computation for the next step would have already been done.

Decomposition Techniques

4. Speculative Decomposition

- Switch Example Algorithm:

1: Calculate expression for the switch condition → task 0

2: Case 0: Multiply vector **b** with matrix **A** → task 1

3: Case 1: Multiply vector **c** with matrix **A** → task 2

4: Case 2: Multiply vector **d** with matrix **A** → task 3

5: display result → task 4

Decomposition Techniques

4. Speculative Decomposition

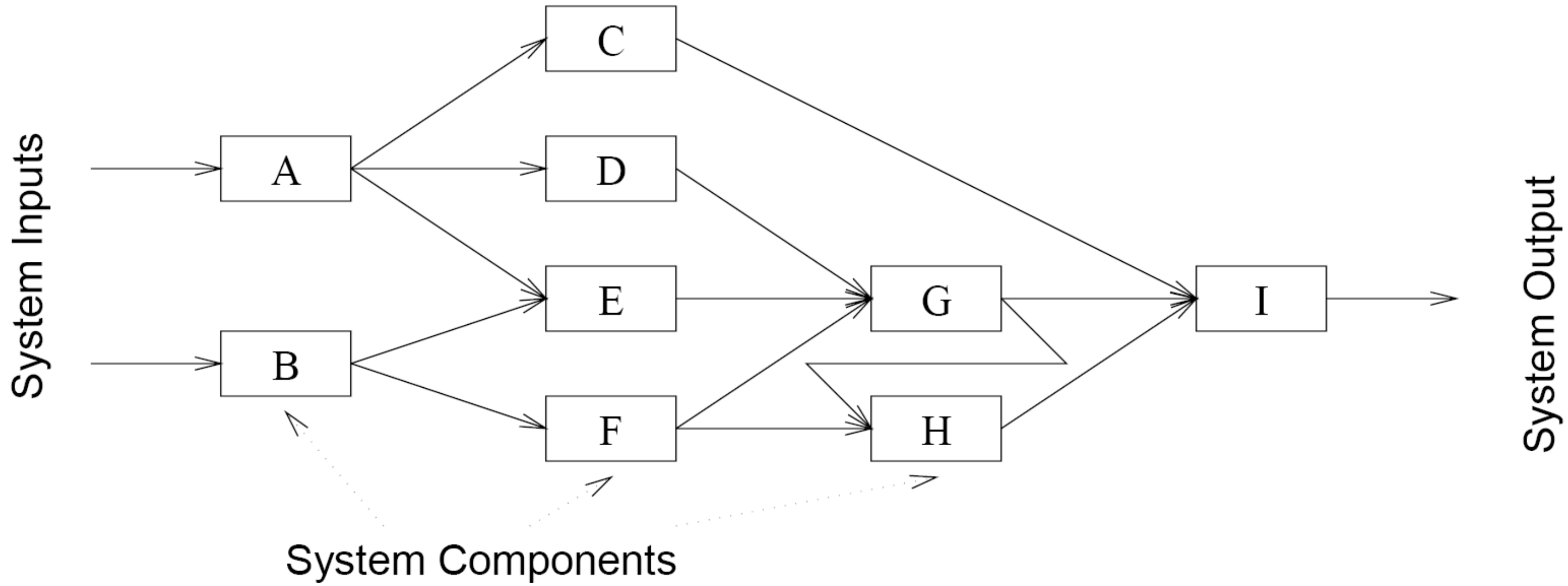


Figure 3.20 A simple network for discrete event simulation.

Decomposition Techniques

5. *Hybrid Decomposition*

- Decomposition technique are not exclusive
  - We often need to combine them together

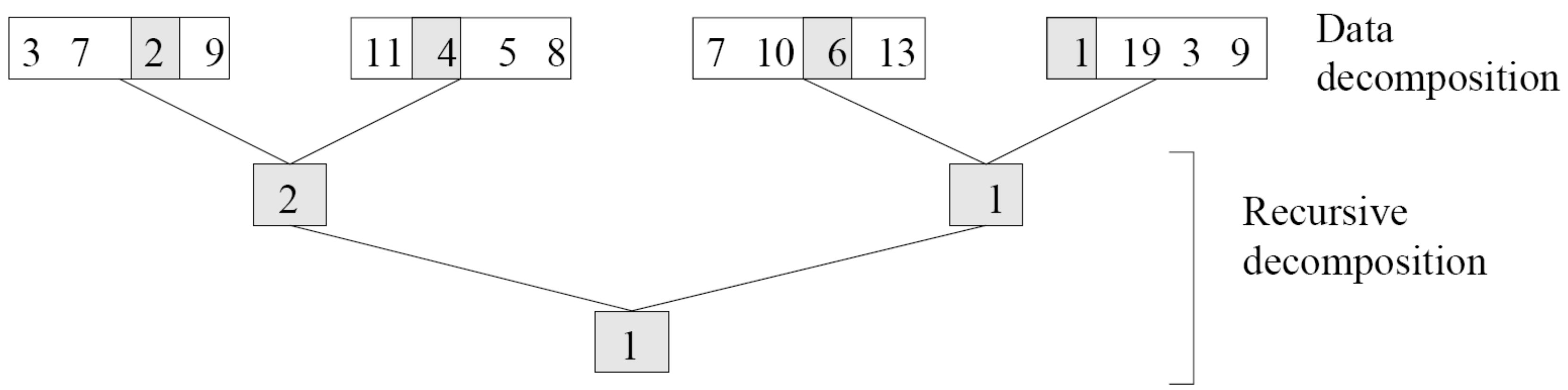


Figure 3.21 Hybrid decomposition for finding the minimum of an array of size 16 using four tasks.

Moving on.....

Mapping Schemes

Static Mapping

- Distributing the tasks among the processes before execution of the program
  - E.g., usually used in situation where total number of tasks and their sizes are known before the execution of the program
- Easy to implement in message passing paradigm

Dynamic Mapping

- When total number of tasks are not known a priori
- (OR) when task sizes are unknown
  - In this case static mapping can lead to serious load-imbalances.

Both static and Dynamic Mappings are equally easy in shared memory paradigm

Schemes for Static Task-Process Mapping

(Mappings based on Data Partitioning)

Array Distribution Schemes

- In a decomposition based on partitioning data, **mapping** the relevant **data** onto the processes is equivalent to **mapping tasks** onto processes *
- Commonly used array mapping schemes:
  - Block distribution
    - 1D and 2D
  - Cyclic and block-cyclic distribution

Schemes for Static Mapping
(Mappings based on Data Partitioning)

Array Distribution Schemes:

- Block distribution (1D)

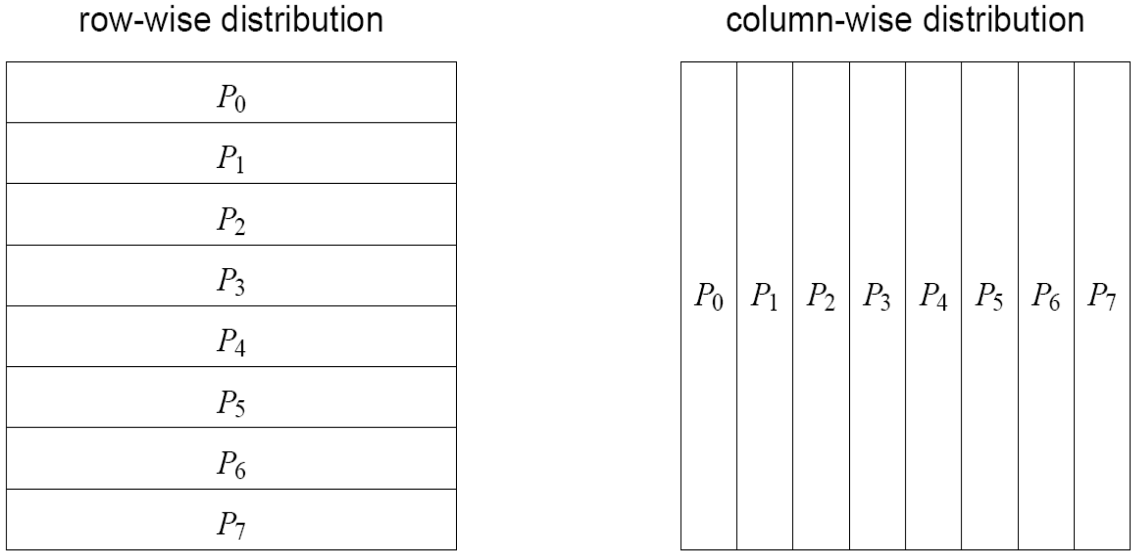
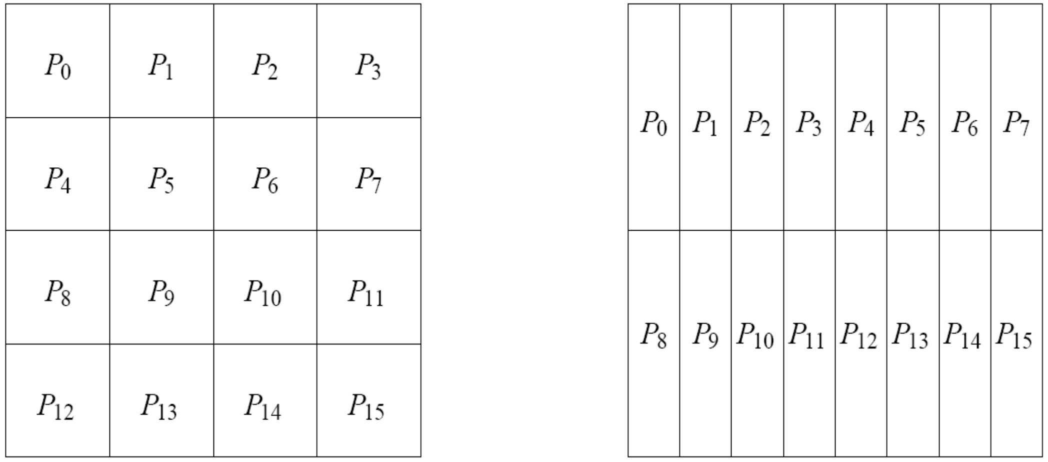


Figure 3.24 Examples of one-dimensional partitioning of an array among eight processes.

Schemes for Static Mapping
(Mappings based on Data Partitioning)

Array Distribution Schemes:

- Block distribution (2D)



(a) (b)

Figure 3.25 Examples of two-dimensional distributions of an array, (a) on a 4×4 process grid, and (b) on a 2×8 process grid.

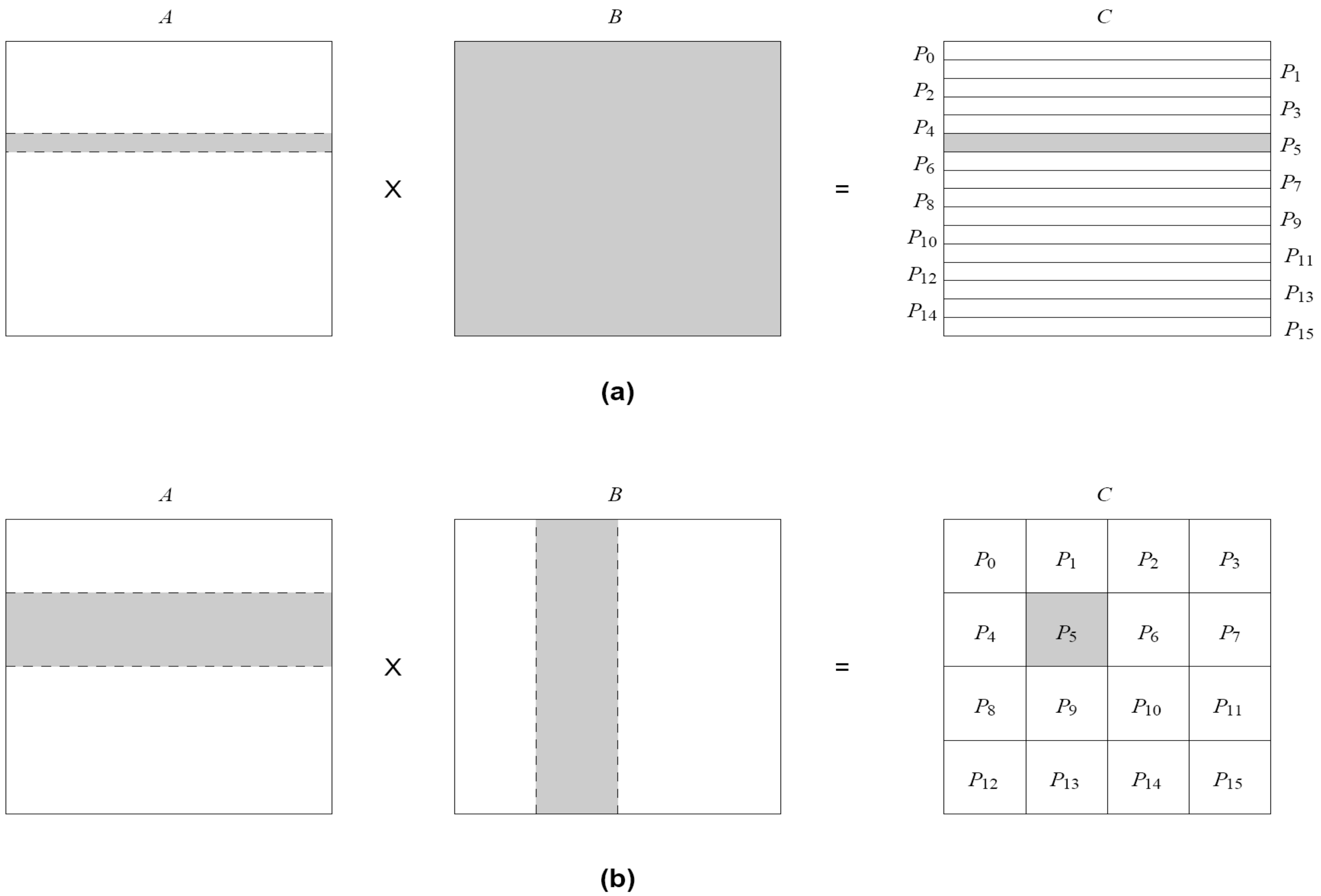


Figure 3.26 Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices A and B are required by the process that computes the shaded portion of the output matrix C.

Schemes for Static Mapping

(Mappings based on Data Partitioning)

Array Distribution Schemes:

- Cyclic distribution (HERE array size=4 x 4 and p=3)

P0	**P1**	**P2**	**P0**
P1	**P2**	**P0**	**P1**
P2	**P0**	**P1**	**P2**
P0	**P1**	**P2**	**P0**

Schemes for Static Mapping

(Mappings based on Data Partitioning)

Array Distribution Schemes:

- Block-Cyclic distribution (1D and 2D)

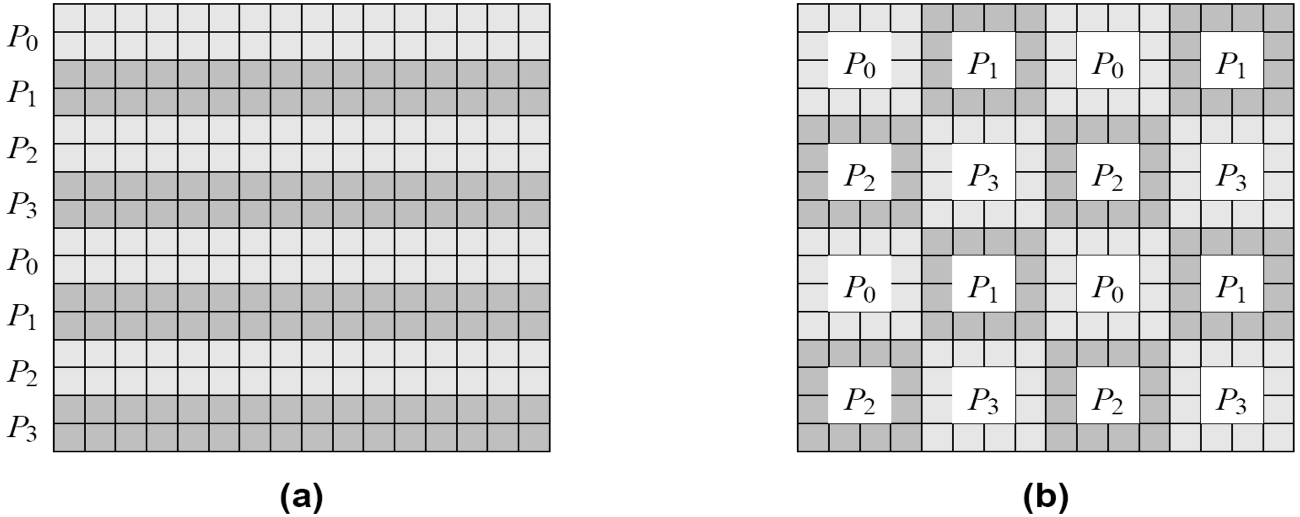


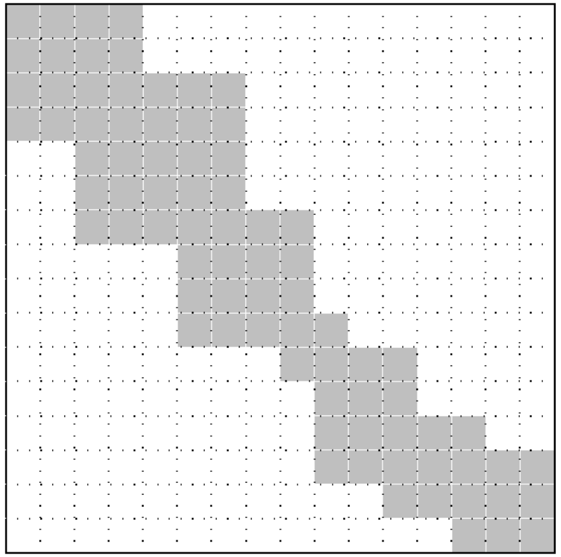
Figure 3.30 Examples of one- and two-dimensional block-cyclic distributions among four processes. (a) The rows of the array are grouped into blocks each consisting of two rows, resulting in eight blocks of rows. These blocks are distributed to four processes in a wraparound fashion. (b) The matrix is blocked into 16 blocks each of size 4×4, and it is mapped onto a 2×2 grid of processes in a wraparound fashion.

Schemes for Static Mapping

(Mappings based on Data Partitioning)

Array Distribution Schemes:

- *Block-Cyclic distribution* (Issue)



(a)

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}

(b)

Figure 3.31 Using the block-cyclic distribution shown in (b) to distribute the computations performed in array (a) will lead to load imbalances.

Schemes for Static Mapping

(Mappings based on Data Partitioning)

Array Distribution Schemes:

- *Randomized-Block distribution* (solution: 1D)

$$V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$$

$$\text{random}(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]$$

$$\text{mapping} = \underbrace{8\ \ 2\ \ 6}_{P_0}\ \underbrace{0\ \ 3\ \ 7}_{P_1}\ \underbrace{11\ \ 1\ \ 9}_{P_2}\ \underbrace{5\ \ 4\ \ 10}_{P_3}$$

Figure 3.32 A one-dimensional randomized block mapping of 12 blocks onto four process (i.e., $\alpha = 3$).

Schemes for Static Mapping

(Mappings based on Data Partitioning)

Array Distribution Schemes:

- *Randomized-Block distribution* (solution: 2D)

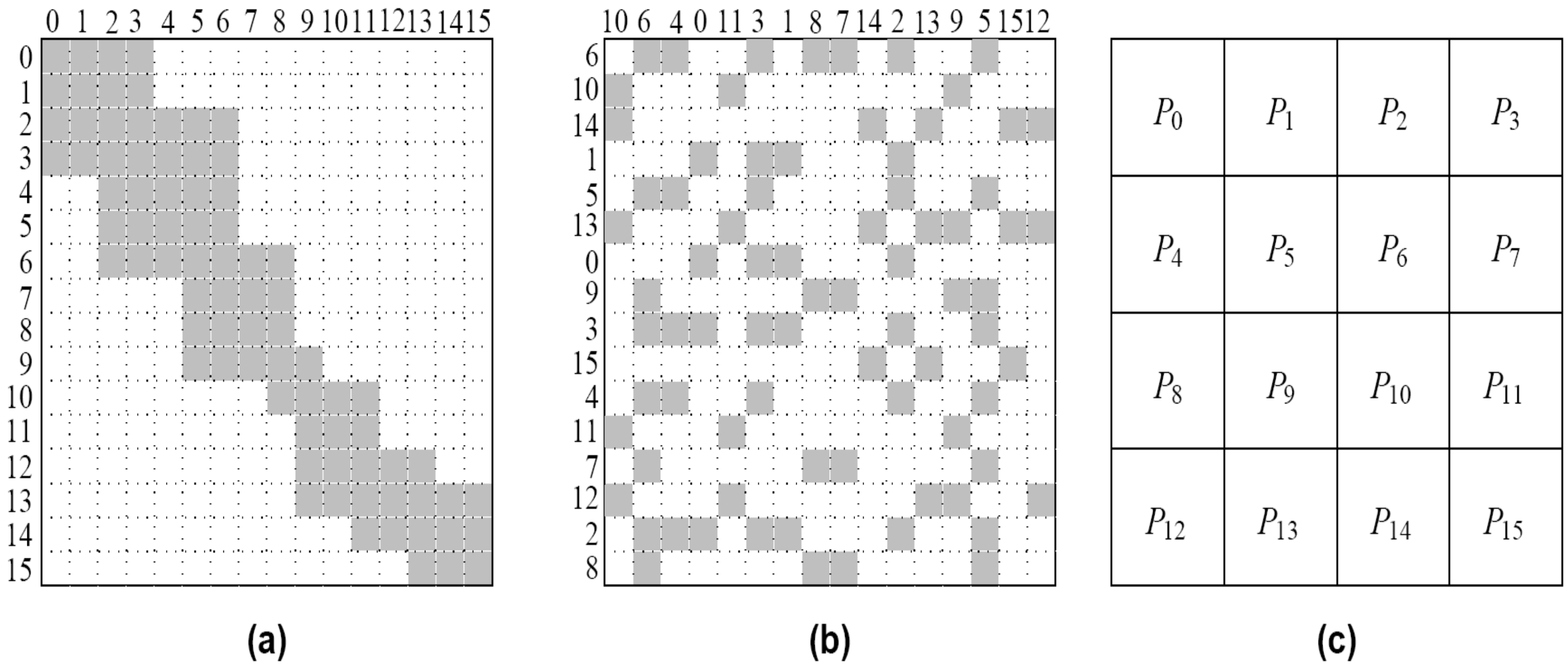


Figure 3.33 Using a two-dimensional random block distribution shown in (b) to distribute the computations performed in array (a), as shown in (c).

Why is *randomized block cyclic distribution* not always used?

A simulation model (*using a mesh of tasks)* for finding dispersion of water contaminant in a lake at different intervals of time.

Random Partitioning

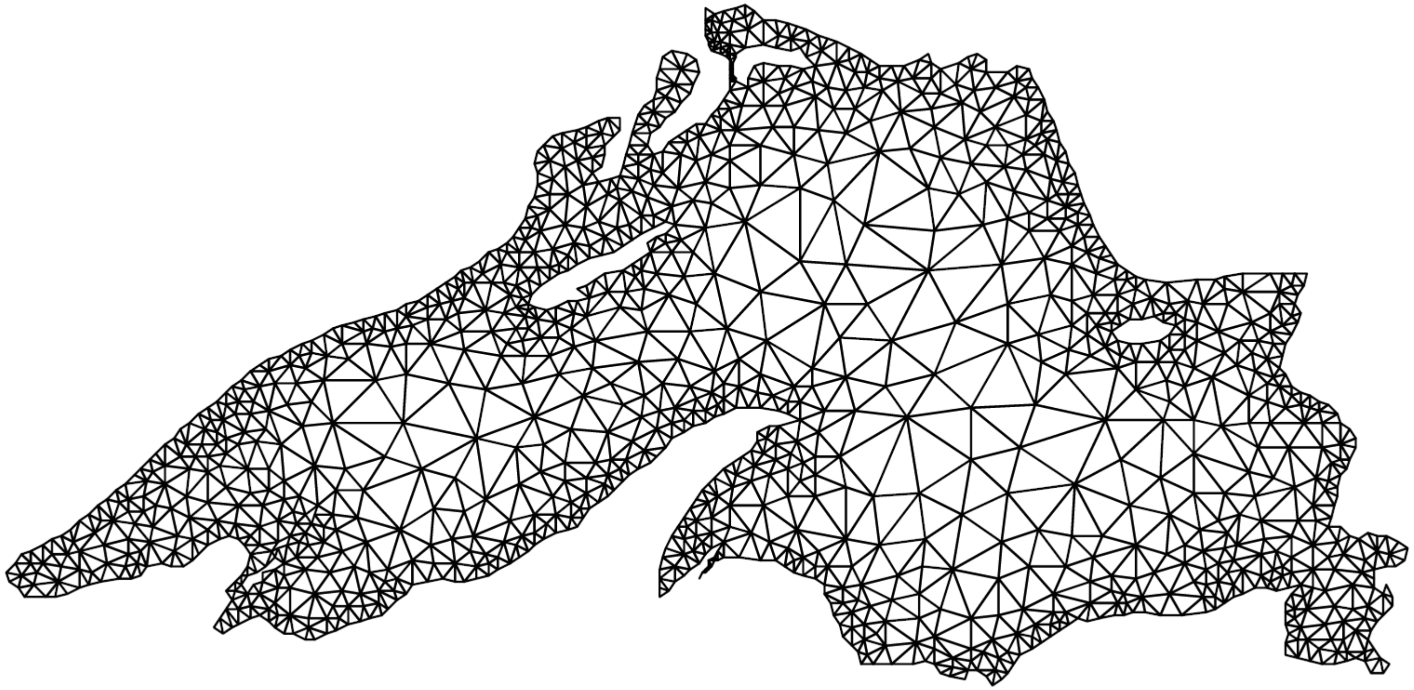


Figure 3.34 A mesh used to model Lake Superior.

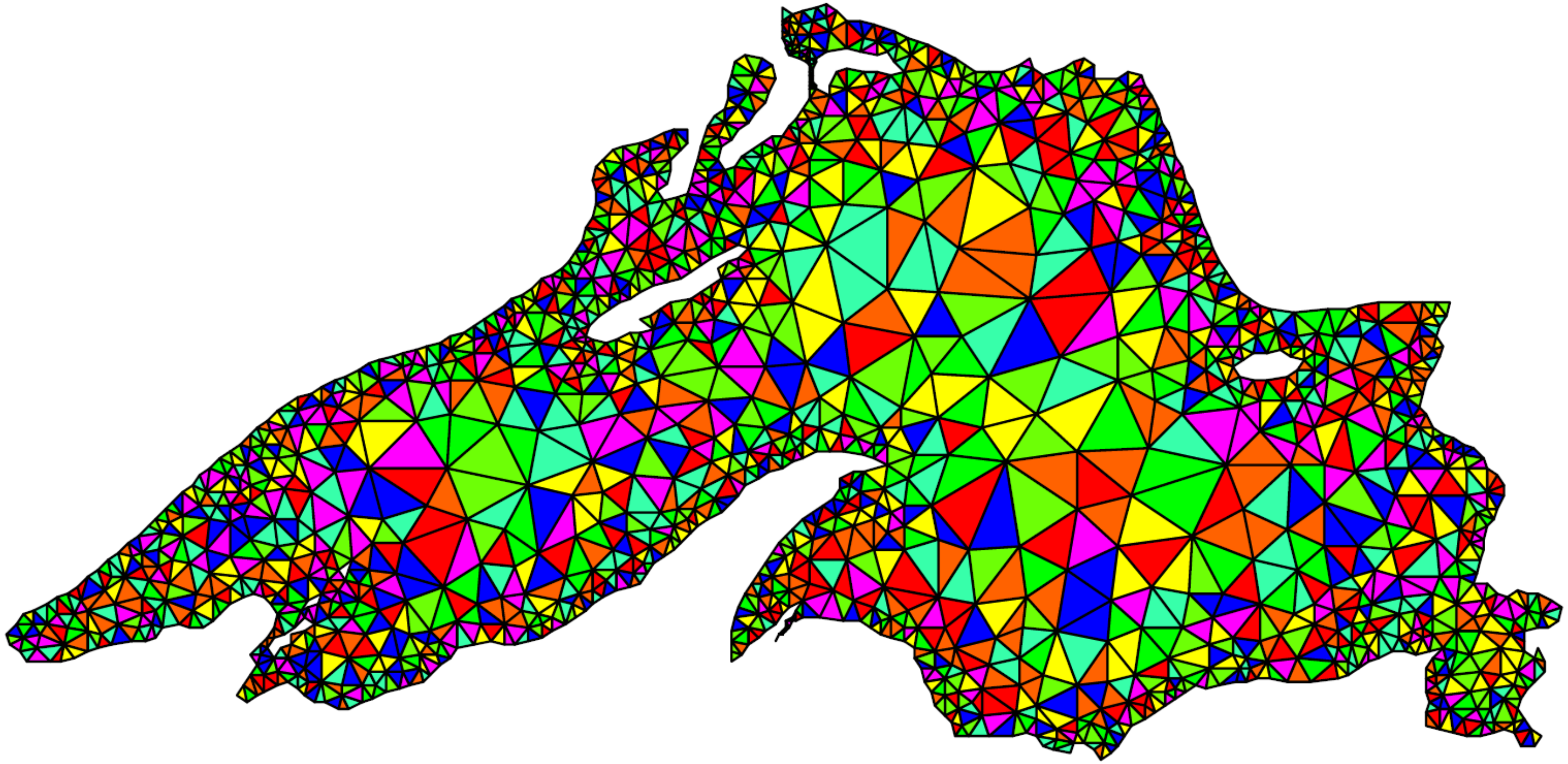


Figure 3.35 A random distribution of the mesh elements to eight processes.

Partitioning for Minimizing Edge-Count

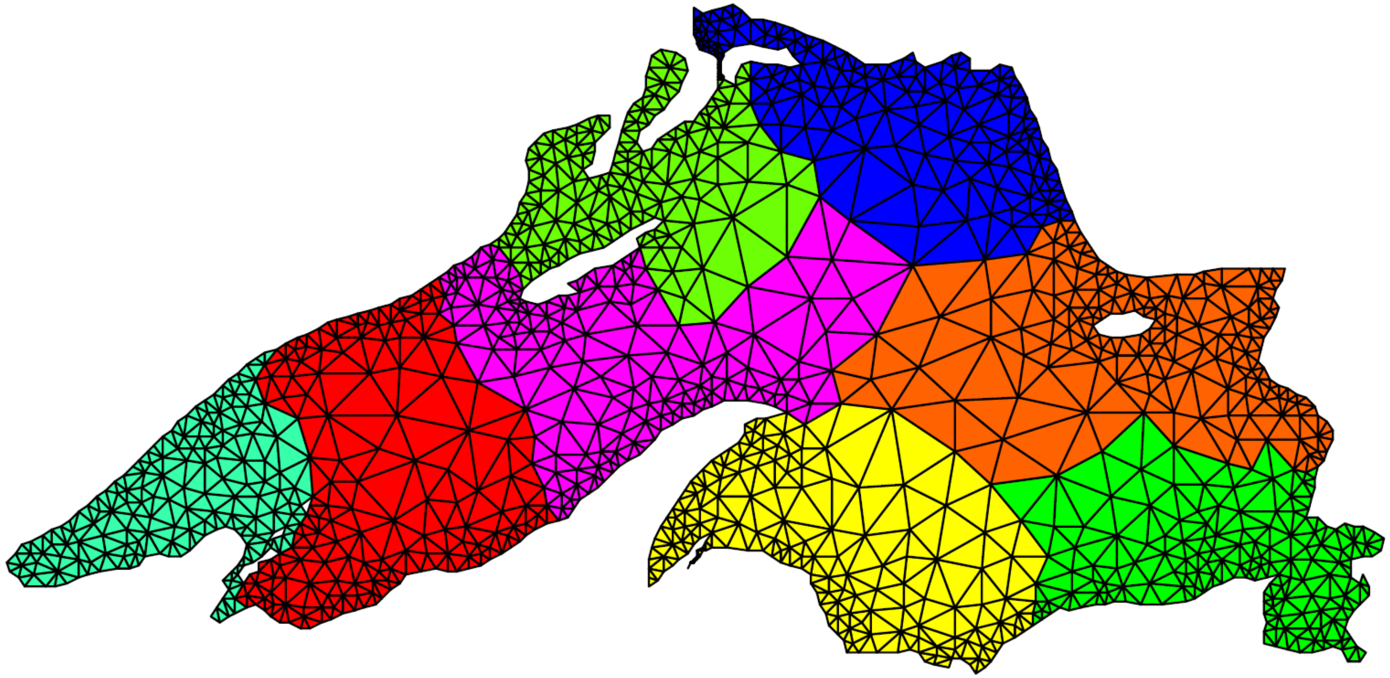


Figure 3.36 A distribution of the mesh elements to eight processes, by using a graph-partitioning algorithm.

Goal of partitioning: Balance work & minimize communication

- *Assign equal number of nodes (or cells) to each process*
  - Random partitioning may lead to high interaction overhead due to data sharing
- *Minimize edge count of the graph partition*
  - Each process should get roughly the same number of elements and the number of edges that cross partition boundaries should be minimized as well.

Thread

- A thread is “an *independent stream of instructions* that can be scheduled to run by the operating system”
- A thread is also a “*procedure that runs independently* from its main program”
- **Pthreads** have been specified (for UNIX) by the IEEE POSIX 1003.1c standard (1995)
- Other threads libraries exist, such as Java threads

Multithreading

- Operating system facility that enables an application to create threads of execution within a process
- Many different users can run programs that appears to be running at the same time
- However with a single processing unit, they are not running at the exact same time
- Operating system switches available resources from one running program to another
- Multiple threads exist within each process and share resources like memory

Pthreads

Posix thread API

- standard threads API, supported by most vendors

- Pthreads are interesting for:
  - Overlapping I/O and CPU work; some threads can block for I/O while others can continue
  - Scheduling and load balancing
  - Ease of programming and widespread use
  - In parallel programming they can be very useful, since communications between threads are much faster (3-5 times)

Threads: Pthreads API

- *Thread management*
  - Create, detach, join, thread attributes
- *Mutexes*
  - Mutual exclusion, create, destroy, lock, unlock, mutex attributes
- *Condition variables*
  - Create, destroy, wait, signal, programmer specified conditions, condition variable attributes

- pthreads.h header file
- Pthreads are defined for C; some FORTRAN compilers also have Pthreads API (e.g. IBM AIX)
- Pthreads API are based on over 60 calls pthread_* ;

Function: `pthread_create`

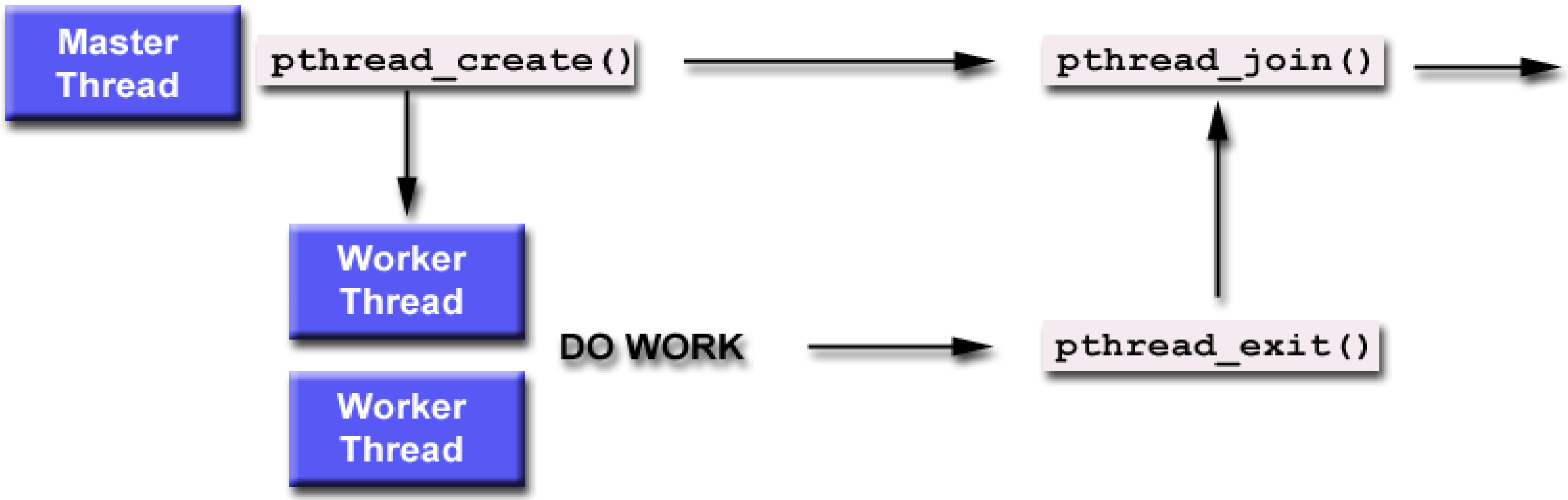
```
int pthread_create (
    pthread_t *thread_handle, const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);
```

- thread_handle unique identifier
- attribute NULL for default attributes
- thread_function C routine executed once thread is created
- arg a single argument that may be passed to thread_function; NULL for no argument

- It can be called any number of times, from anywhere, there is no hierarchy or dependency

Threads: Joining and detaching threads

- `int pthread_join(pthread_t thread_handle, void **value_ptr)`
- It is possible to get return status if specified in `pthread_exit()`
- Only one `pthread_join()` call can be matched
- Thread can be joinable or detached (no possibility to join); it is better to declare it for portability!
- `int pthread_detach(pthread_t thread);`
  - is used to indicate to the implementation that storage for the thread can be reclaimed when the thread terminates. If thread has not terminated, `pthread_detach()` will not cause it to terminate. It works even if thread was created as joinable



- POSIX standard does not specify stack size for a thread; exceeding the limit produces a *segmentation fault*
- Safe and portable programs explicitly allocate enough stack

Example

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
  long tid;
  tid = (long) threadid;
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}
```

```c
int main (int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
      }
  }
}
```

Synchronization problem

- Example: Bank Transactions
  - Current balance = PKR 70,000
  - Check deposited = PKR 10,000
  - ATM withdrawn = PKR 5,000
- Correct Balance after both transactions
  - Balance = 75,000

What if both transactions are initiated at the same time!

- Check Deposit:
  - MOV A, Balance // A = 70,000
  - ADD A, Deposited // A = 80,000
- ATM Withdrawal:
  - MOV B, Balance // B = 70,000
  - SUB B, Withdrawn // B = 65,000

Mutual exclusion

- Mutual exclusion variables (Mutex) work like a lock protecting access to a shared resource
- Only one thread can lock a mutex at a moment; even if more than one thread tries to lock the mutex, only one will be successful; this avoids race
- Sequence:
  - Creation of the mutex
  - More than one thread tries to lock the mutex
  - Only one locks it
  - The owner makes changes
  - The owner unlocks it
  - Another thread gets mutex (it was blocked, unblocking is automatic) and the process repeats
  - At the end mutex is destroyed

Critical Section

```
do
{
        Entry section

          critical section

        Exit section

       remainder section

} while(1)
```

Mutual exclusion

- Mutual exclusion variables (Mutex) work like a lock protecting access to a shared resource
- Only one thread can lock a mutex at a moment; even if more than one thread tries to lock the mutex, only one will be successful; this avoids race

- The Pthreads API provides the following functions for handling mutex-locks:

```
int pthread_mutex_lock ( pthread_mutex_t *mutex_lock);

int pthread_mutex_unlock ( pthread_mutex_t *mutex_lock);

int pthread_mutex_init ( pthread_mutex_t *mutex_lock, const pthread_mutexattr_t *lock_attr);
```

Mutual exclusion

- For example:

```
pthread_mutex_t total_cost_lock;
...
main() {
    ....
    pthread_mutex_init(&total_cost_lock, NULL);
    ....
}
void *add_cost(void *costn) {
    ....
    pthread_mutex_lock(&total_cost_lock);

    total_cost = total_cost + costn;
    /* and unlock the mutex */
    pthread_mutex_unlock(&total_cost_lock);
}
```

Locking overhead

- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can lead to significant performance degradation.
- It is often possible to reduce the idling overhead associated with locks using an alternate function, pthread_mutex_trylock.

```
int pthread_mutex_trylock (pthread_mutex_t *mutex_lock);
```

- pthread_mutex_trylock is typically much faster than pthread_mutex_lock on typical systems since it does not have to deal with queues associated with locks for multiple threads waiting on the lock.

Trylock()

```
pthread_mutex_t total_cost_lock;
int lock_status;
main() {
   pthread_mutex_init(&total_cost_lock, NULL);
   ....
}
void *add_cost(void *costn) {
   ....
   Lock_status pthread_mutex_trylock(&total_cost_lock);
   if (lock_status == EBUSY)
        addlater;
   else
        total_cost = total_cost + costn;
        /* and unlock the mutex */
        pthread_mutex_unlock(&total_cost_lock);
}
```

Moving on to OpenMP

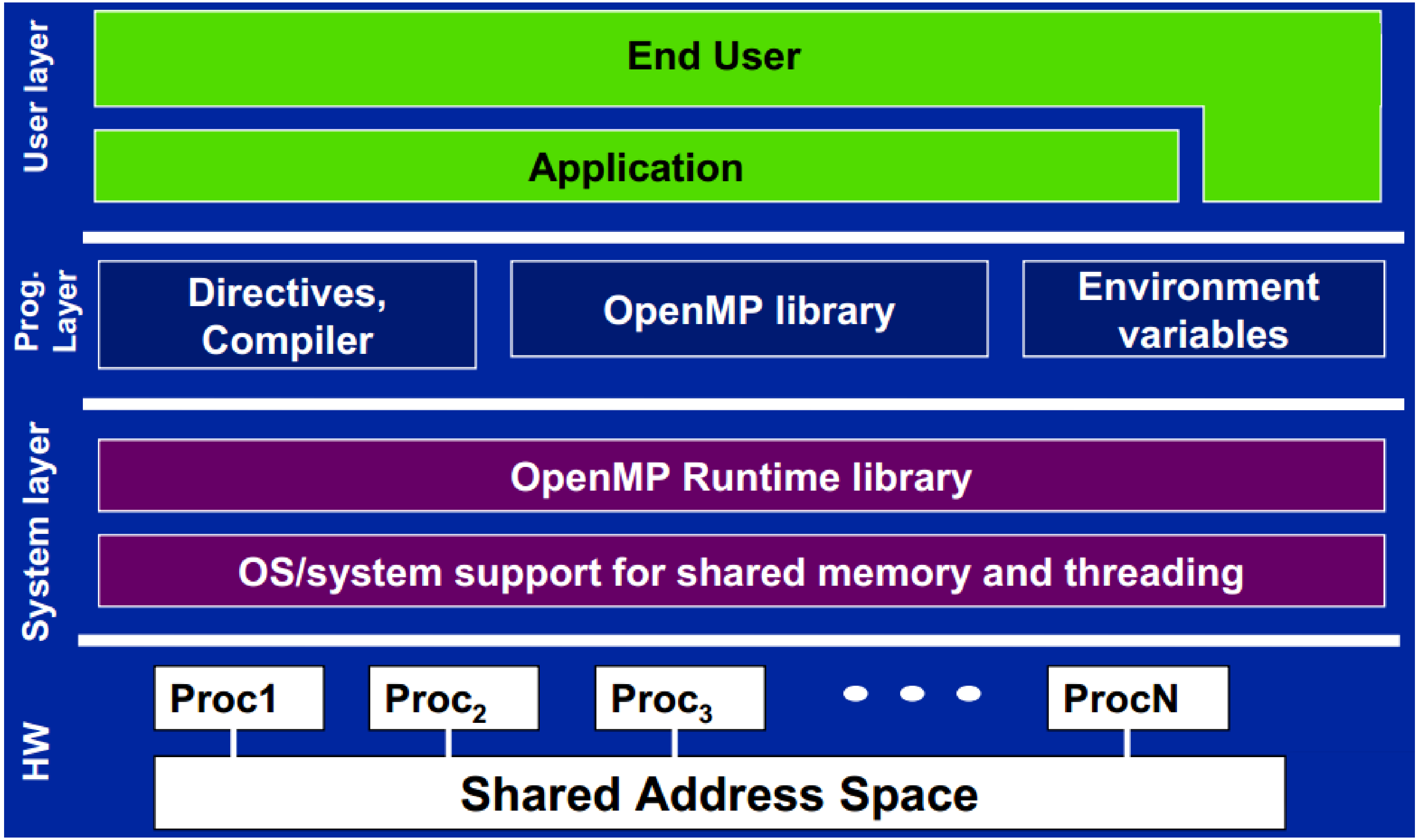
OpenMP

- OpenMP (Open Multi Processing) is an API for writing multithreaded applications
- Provides an implementation model to distribute and decompose the work across multiple processors
- Uses threads to deploy work
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++

OpenMP

- OpenMP is described by the API based on:
  - A set of compiler directives for depicting the parallelism in the source code
  - A library of subroutines
  - A set of Environment Variables
- OpenMP directives in C and C++ are based on the `#pragma` compiler directives.

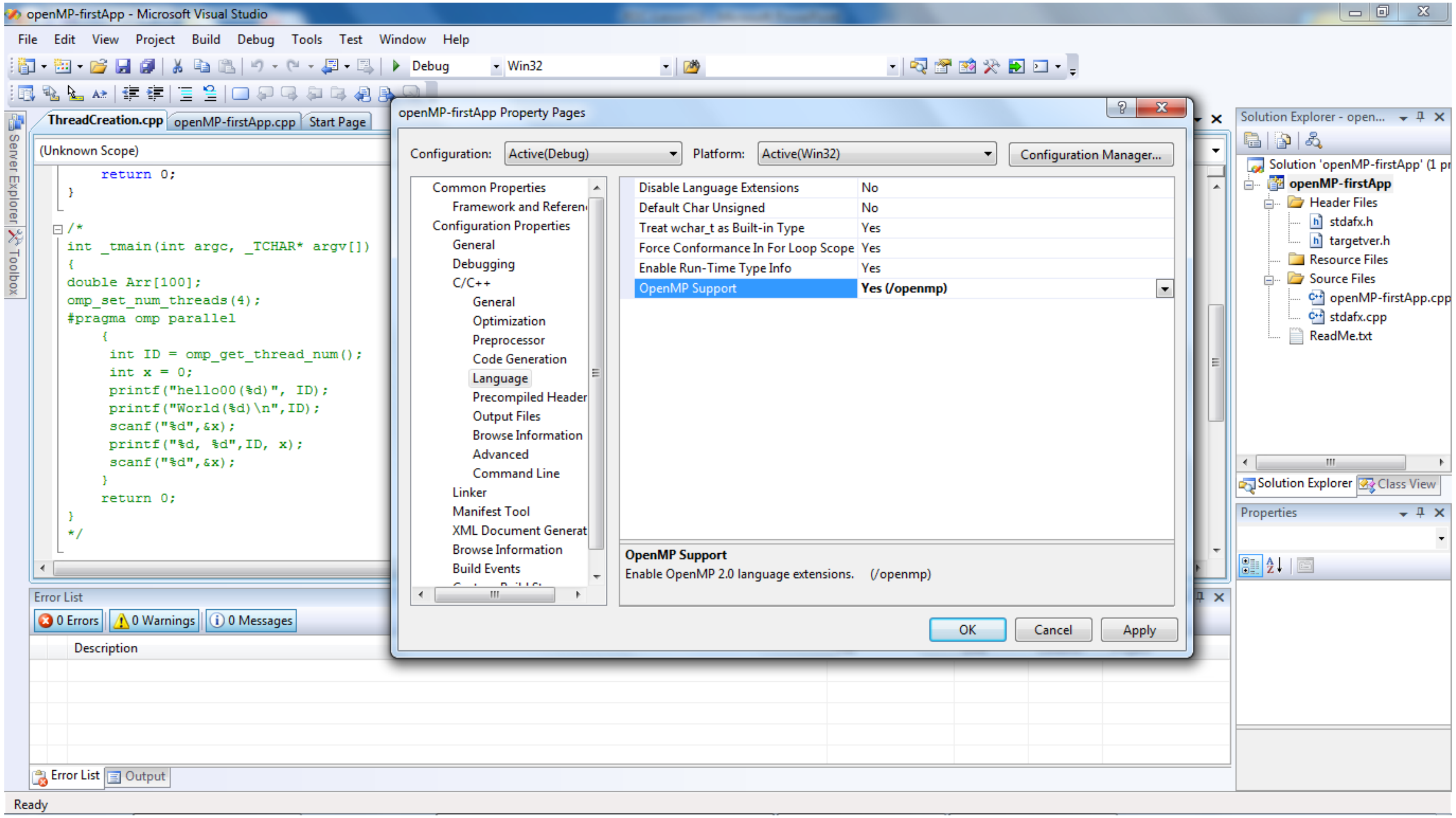
OpenMP solution stack model



Source: http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

Implementation using Visual Studio C++

- Turn on OpenMP support in Visual Studio
- Project properties → Configuration → C/C++/Language



First Program: hello world

```
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
 omp_set_num_threads(4);
 #pragma omp parallel
  {
    int Id = omp_get_thread_num();

    printf ("hello(%d)", Id);
    printf ("world(%d)\n", Id)
  }
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

```c
#include <omp.h>
int numT;
int main()
{
 #pragma omp parallel num_threads(4)
  {
    int Id = omp_get_thread_num();
    numT = omp_get_num_threads();
    printf ("hello(%d)", Id);
    printf ("world(%d)\n", Id)
  }
}
```

Clause to request a certain number of threads

Runtime function returning the num of threads actually created

Sources

- Slides of Dr. Rana Asif Rahman & Dr. Haroon Mahmood, FAST
- (Chapter 2) Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). Introduction to parallel computing (Vol. 110). Redwood City, CA: Benjamin/Cummings.
- Quinn, M. J. Parallel Programming in C with MPI and OpenMP,(2003).