DTS205TC High Performance Computing

Lecture 7 Methodology 2

Di Zhang, Spring 2024

Mapping Techniques

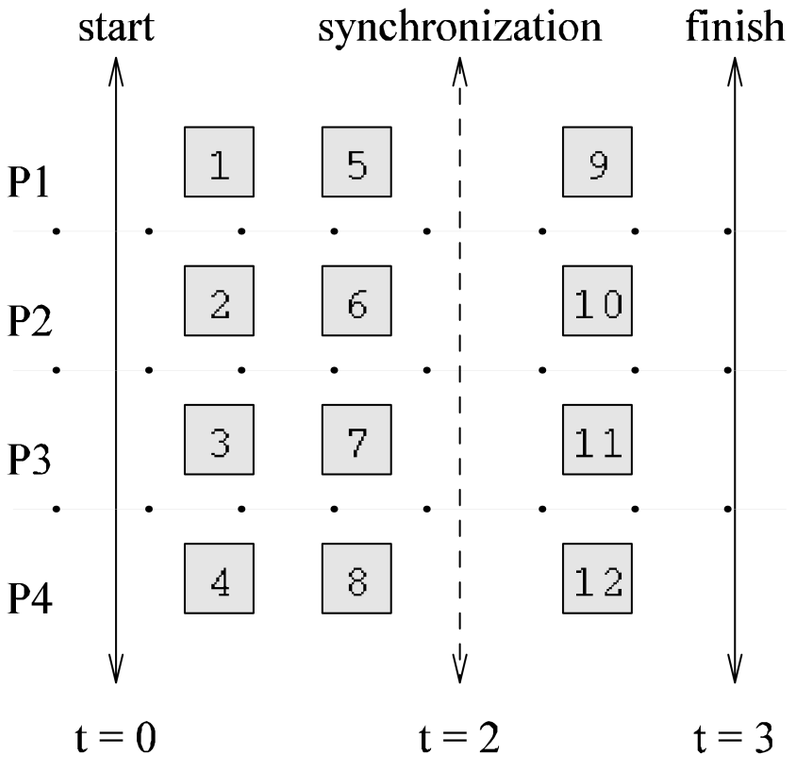
- Once a problem has been decomposed into concurrent tasks, these must be mapped to processes (that can be executed on a parallel platform).
- Mappings must minimize overheads.
- Primary overheads are communication and idling.
- Minimizing these overheads often represents contradicting objectives.
- Assigning all work to one processor trivially minimizes communication at the expense of significant idling.

Mapping Techniques for Minimum Idling

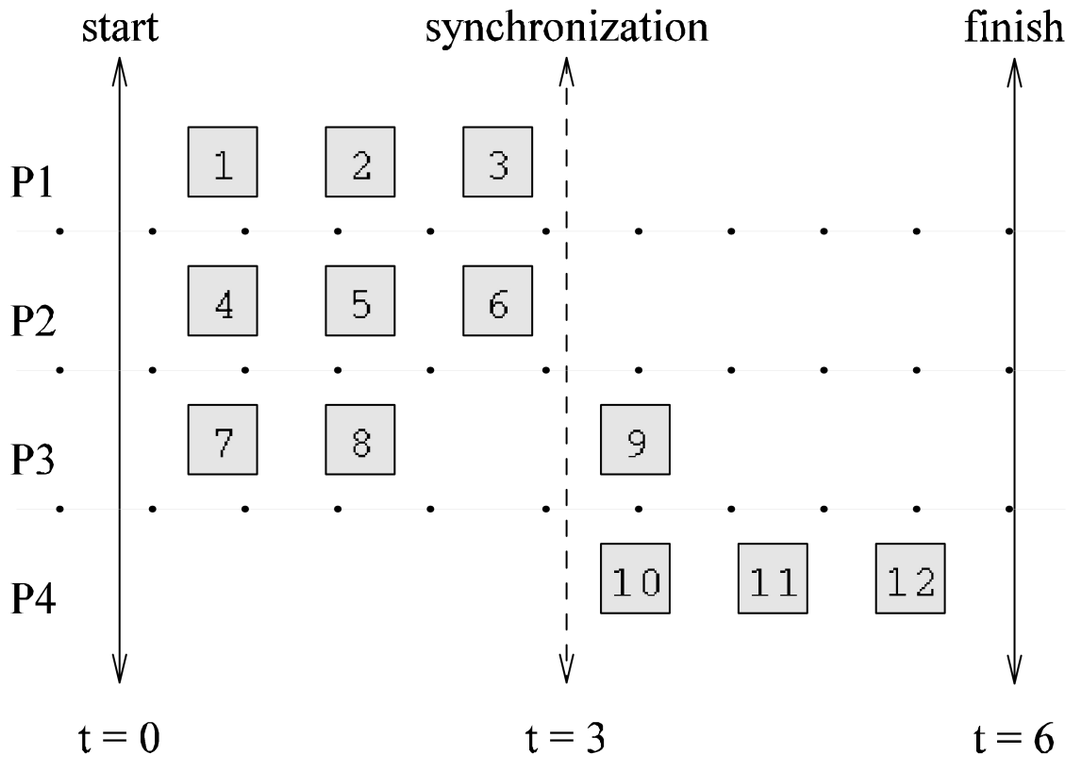
Xi'an Jiaotong-Liverpool University
西交利物浦大學

Mapping must simultaneously minimize idling and load balance.

Merely balancing load does not minimize idling.



(a)



(b)

Mapping Techniques for Minimum Idling

Mapping techniques can be static or dynamic.

- Static Mapping: Tasks are mapped to processes a-priori. For this to work, we must have a good estimate of the size of each task. Even in these cases, the problem may be NP complete.
- Dynamic Mapping: Tasks are mapped to processes at runtime. This may be because the tasks are generated at runtime, or that their sizes are not known.

Other factors that determine the choice of techniques include the size of data associated with a task and the nature of underlying domain.

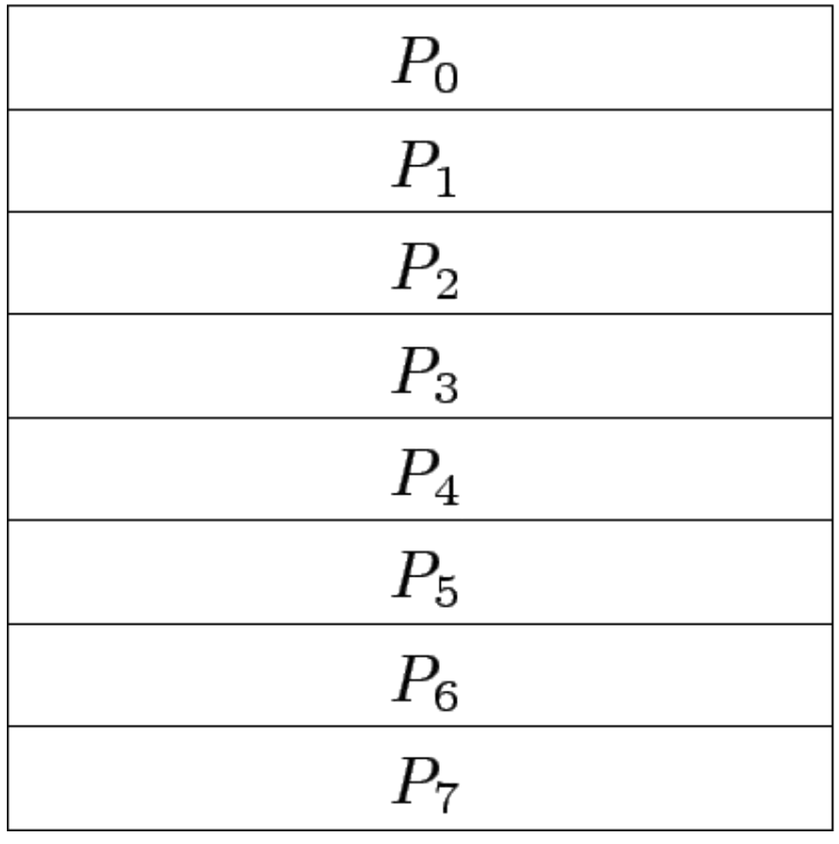
Schemes for Static Mapping

- Mappings based on data partitioning.
- Mappings based on task graph partitioning.
- Hybrid mappings.

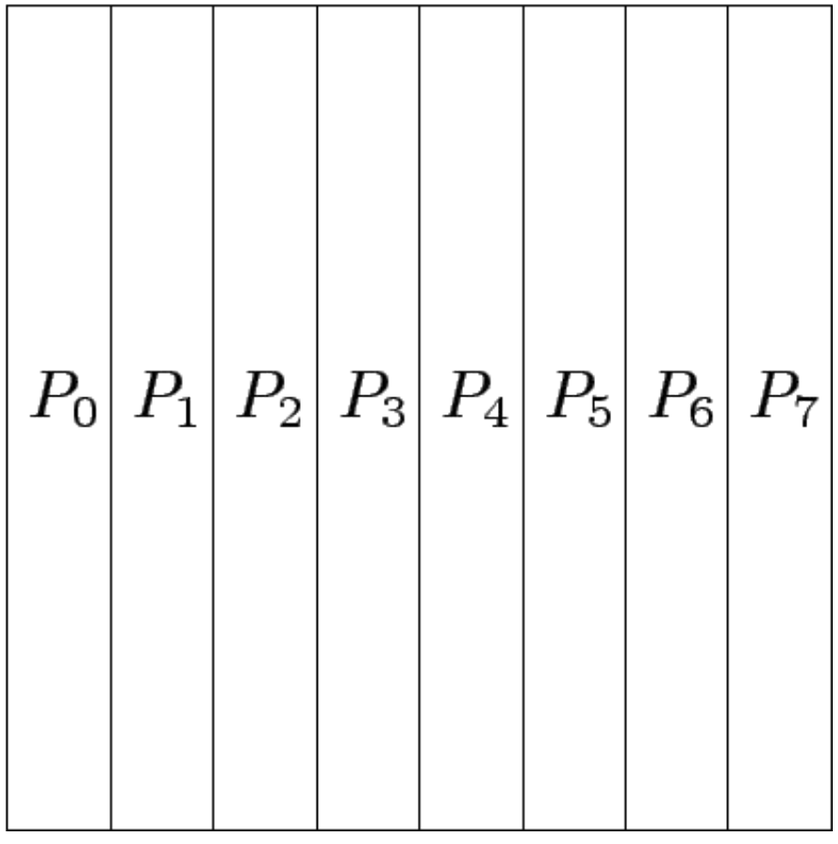
Mappings Based on Data Partitioning

We can combine data partitioning with the ``owner-computes'' rule to partition the computation into subtasks. The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.

row-wise distribution

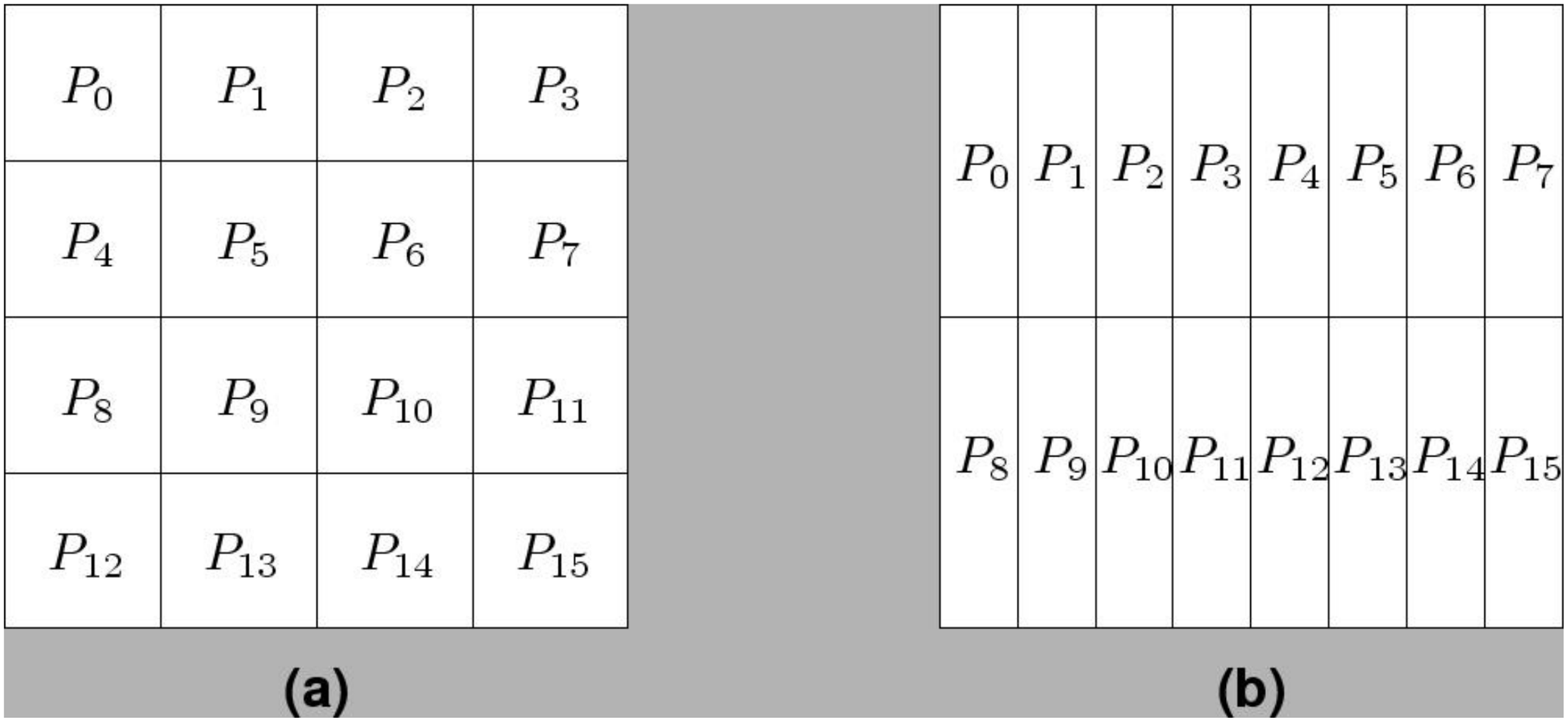


column-wise distribution



Block Array Distribution Schemes

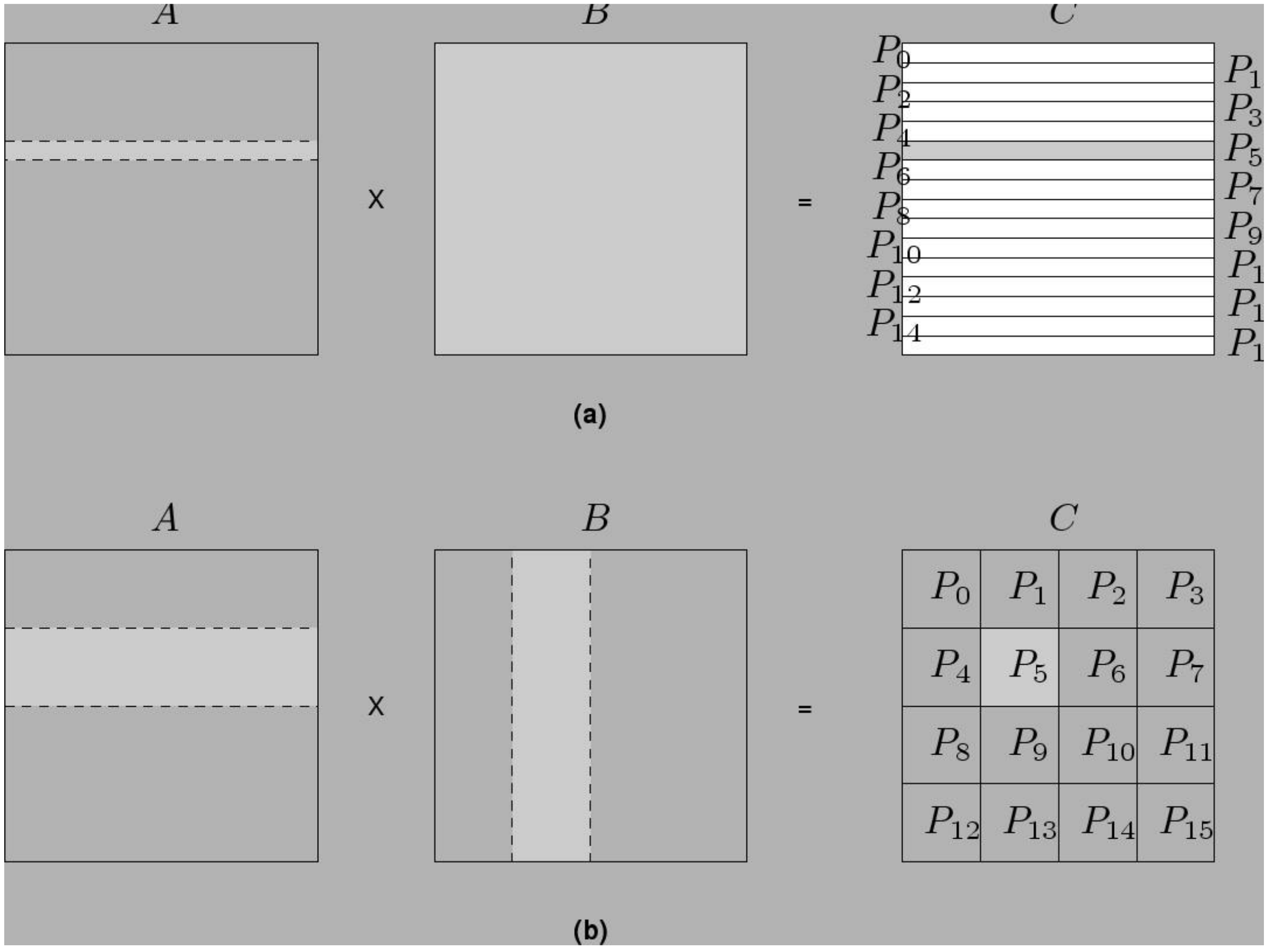
Block distribution schemes can be generalized to higher dimensions as well.



Block Array Distribution Schemes: Examples

- For multiplying two dense matrices ***A*** and ***B***, we can partition the output matrix ***C*** using a block decomposition.
- For load balance, we give each task the same number of elements of ***C***. (Note that each element of ***C*** corresponds to a single dot product.)
- The choice of precise decomposition (1-D or 2-D) is determined by the associated communication overhead.
- In general, higher dimension decomposition allows the use of larger number of processes.

Data Sharing in Dense Matrix Multiplication



Cyclic and Block Cyclic Distributions

- If the amount of computation associated with data items varies, a block decomposition may lead to significant load imbalances.
- A simple example of this is in LU decomposition (or Gaussian Elimination) of dense matrices.

LU Factorization of a Dense Matrix

A decomposition of LU factorization into 14 tasks - notice the significant load imbalance.

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} . \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

1: $A_{1,1} \rightarrow L_{1,1}U_{1,1}$

2: $L_{2,1} = A_{2,1}U_{1,1}^{-1}$

3: $L_{3,1} = A_{3,1}U_{1,1}^{-1}$

4: $U_{1,2} = L_{1,1}^{-1}A_{1,2}$

5: $U_{1,3} = L_{1,1}^{-1}A_{1,3}$

6: $A_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$

7: $A_{3,2} = A_{3,2} - L_{3,1}U_{1,2}$

8: $A_{2,3} = A_{2,3} - L_{2,1}U_{1,3}$

9: $A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}$

10: $A_{2,2} \rightarrow L_{2,2}U_{2,2}$

11: $L_{3,2} = A_{3,2}U_{2,2}^{-1}$

12: $U_{2,3} = L_{2,2}^{-1}A_{2,3}$

13: $A_{3,3} = A_{3,3} - L_{3,2}U_{2,3}$

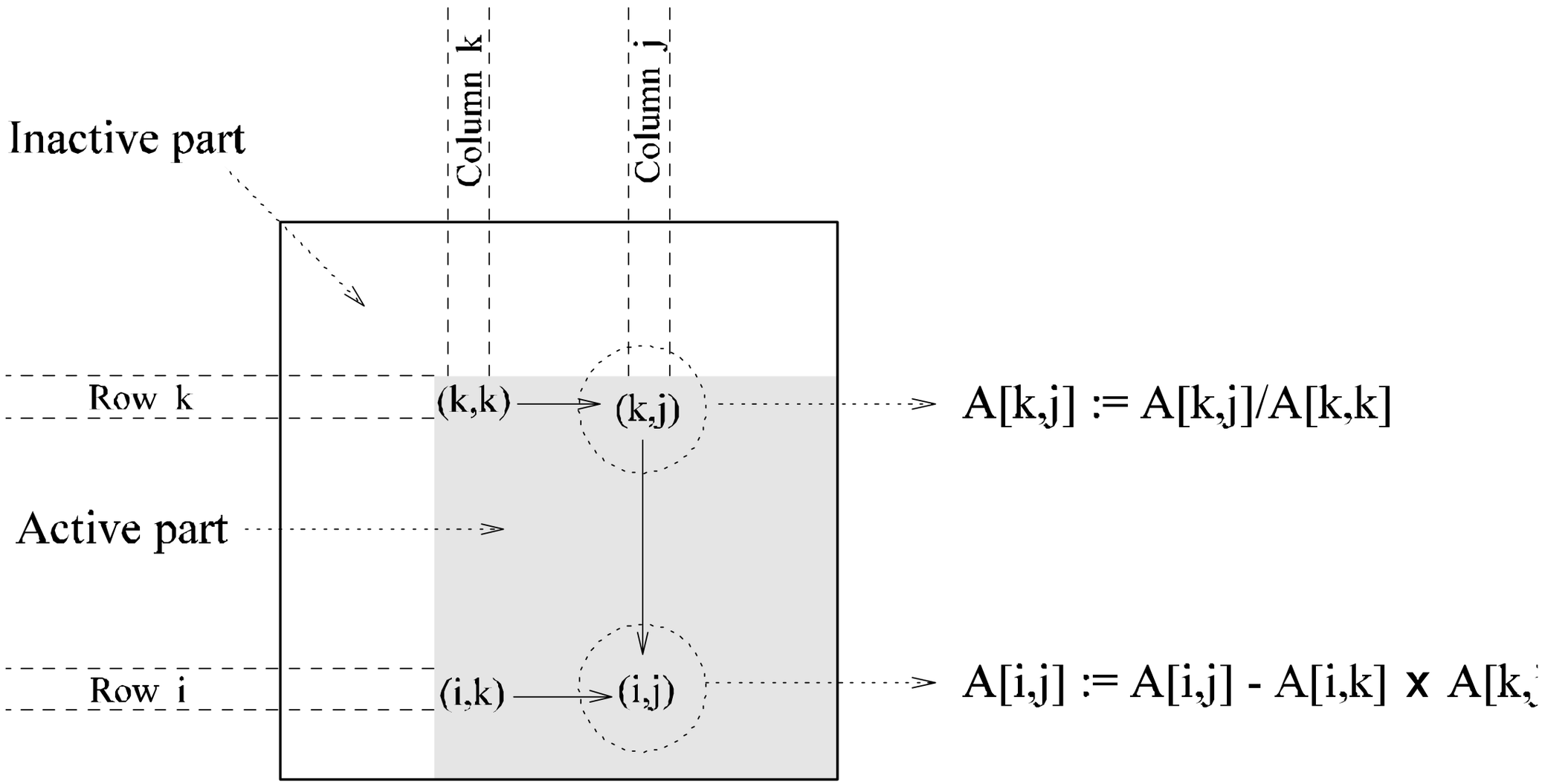
14: $A_{3,3} \rightarrow L_{3,3}U_{3,3}$

Block Cyclic Distributions

- Variation of the block distribution scheme that can be used to alleviate the load-imbalance and idling problems.
- Partition an array into many more blocks than the number of available processes.
- Blocks are assigned to processes in a round-robin manner so that each process gets several non-adjacent blocks.

Block-Cyclic Distribution for Gaussian Elimination

The active part of the matrix in Gaussian Elimination changes. By assigning blocks in a block-cyclic fashion, each processor receives blocks from different parts of the matrix.



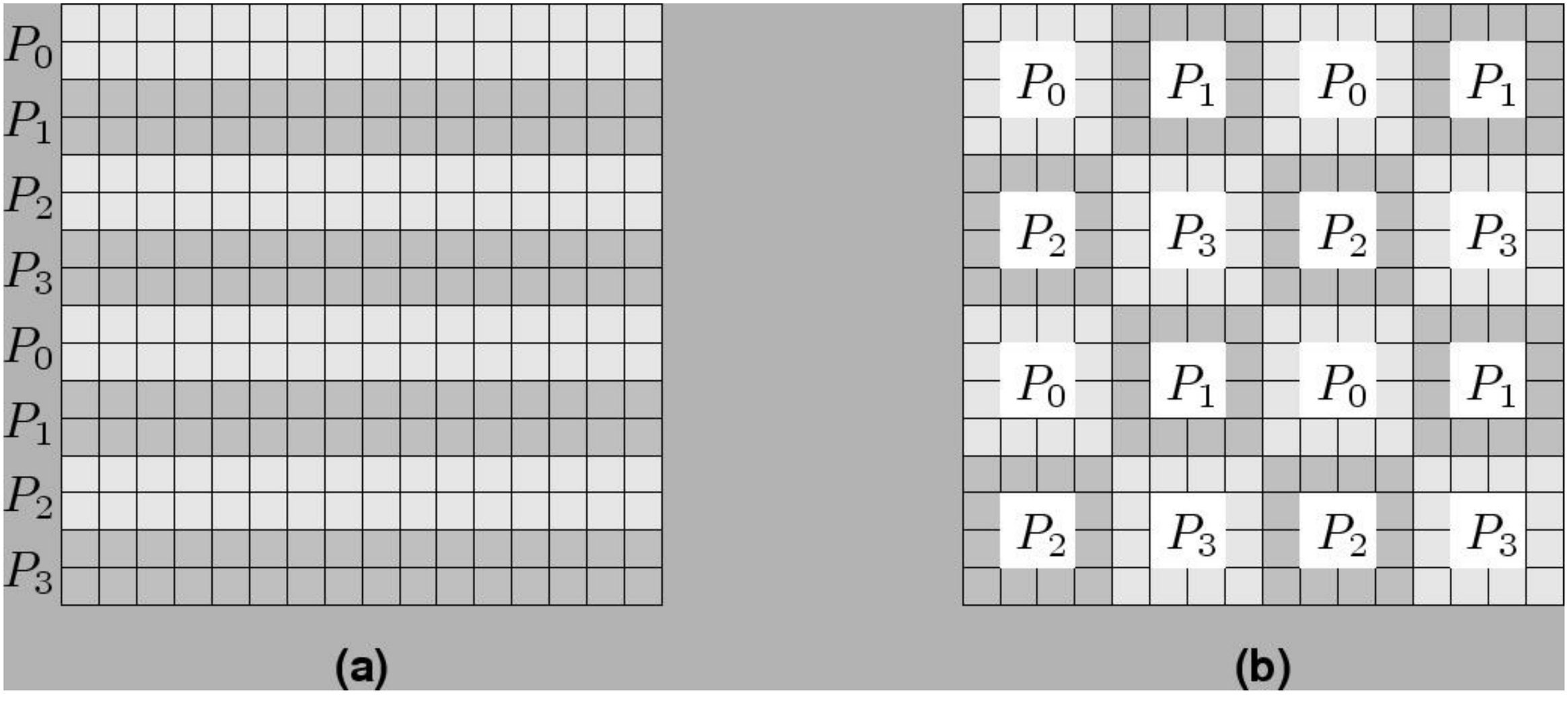
Block-Cyclic Distribution: Examples

One- and two-dimensional block-cyclic distributions among 4 processes.

P_0 T_1	**P_3** T_4	**P_6** T_5
P_1 T_2	**P_4** T_6 T_{10}	**P_7** T_8 T_{12}
P_2 T_3	**P_5** T_7 T_{11}	**P_8** T_9 T_{13} T_{14}

Block-Cyclic Distribution

- A cyclic distribution is a special case in which block size is one.
- A block distribution is a special case in which block size is ***n/p*** , where ***n*** is the dimension of the matrix and ***p*** is the number of processes.

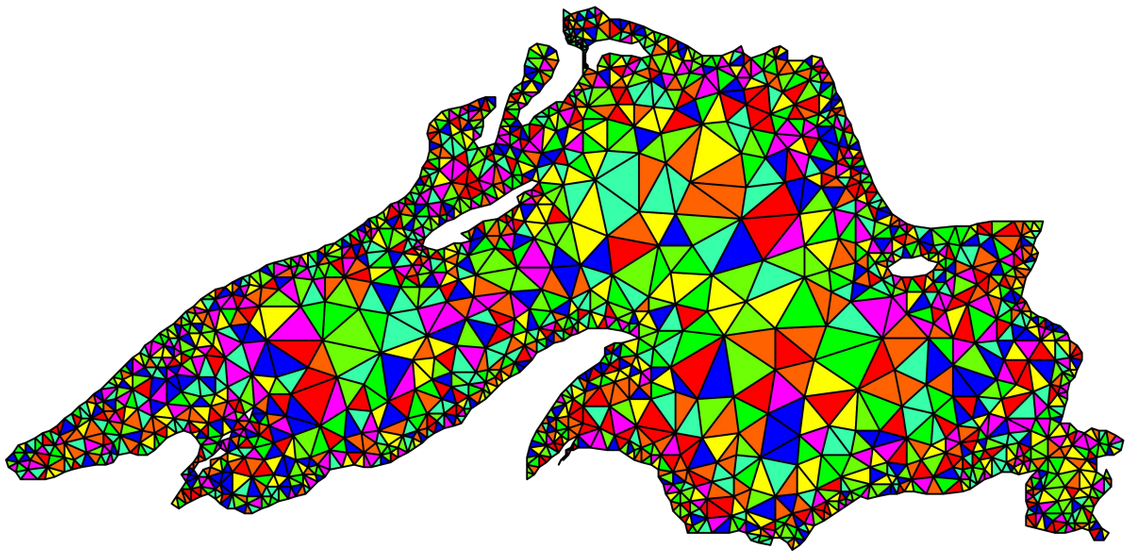


Graph Partitioning Based Data Decomposition

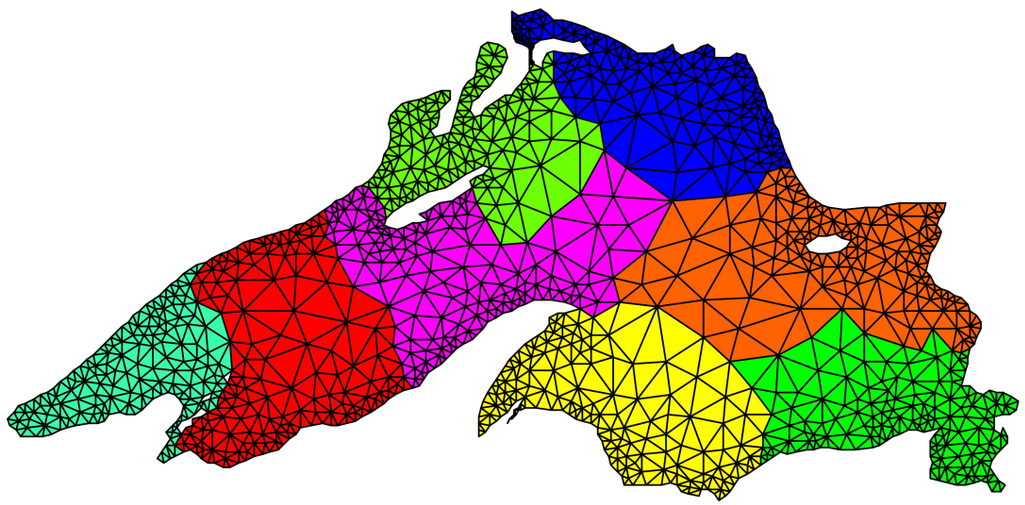
- In case of sparse matrices, block decompositions are more complex.
- Consider the problem of multiplying a sparse matrix with a vector.
- The graph of the matrix is a useful indicator of the work (number of nodes) and communication (the degree of each node).
- In this case, we would like to partition the graph so as to assign equal number of nodes to each process, while minimizing edge count of the graph partition.

Partitioning the Graph of Lake Superior

Xi'an Jiaotong-Liverpool University
西交利物浦大學



Random Partitioning



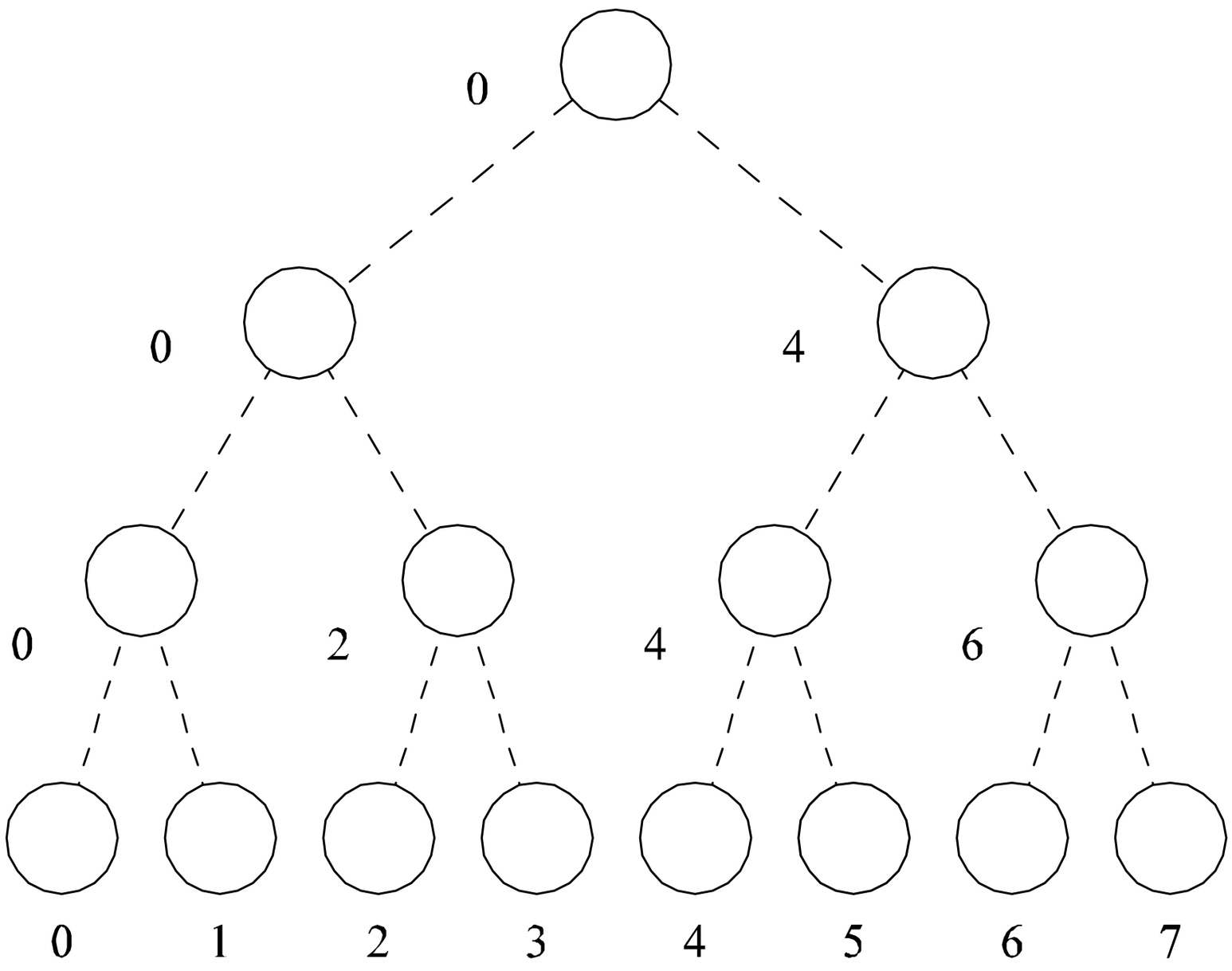
Partitioning for minimum edge-cut.

Mappings Based on Task Paritioning

- Partitioning a given task-dependency graph across processes.
- Determining an optimal mapping for a general task-dependency graph is an NP-complete problem.
- Excellent heuristics exist for structured graphs.

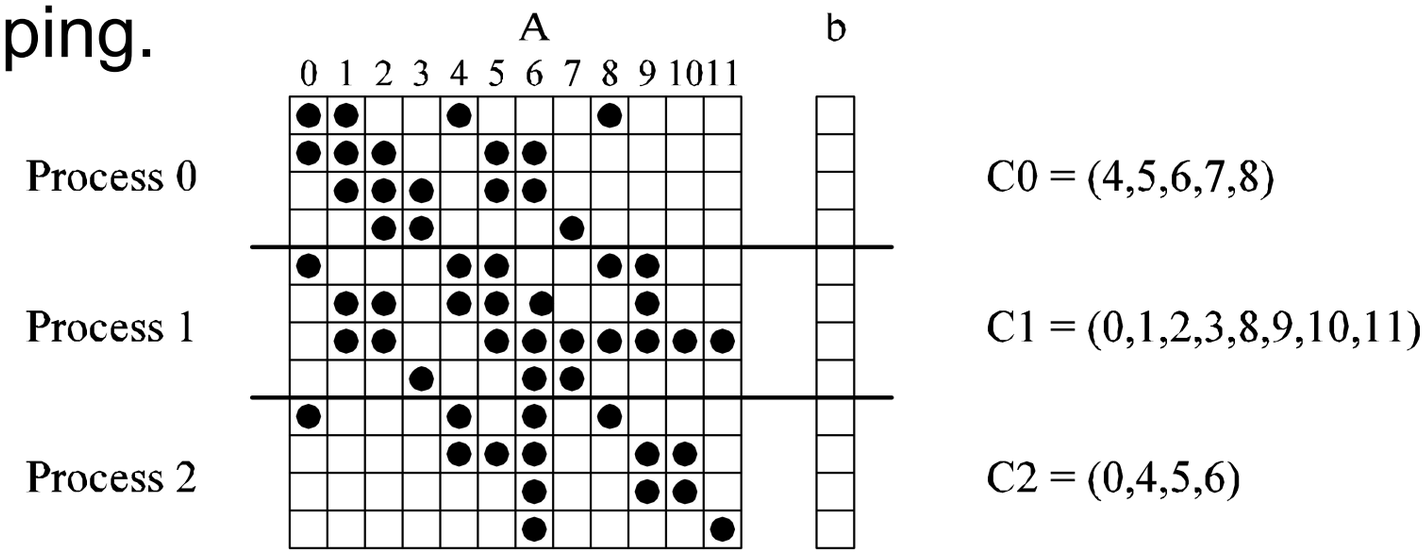
Task Partitioning: Mapping a Binary Tree Dependency Graph

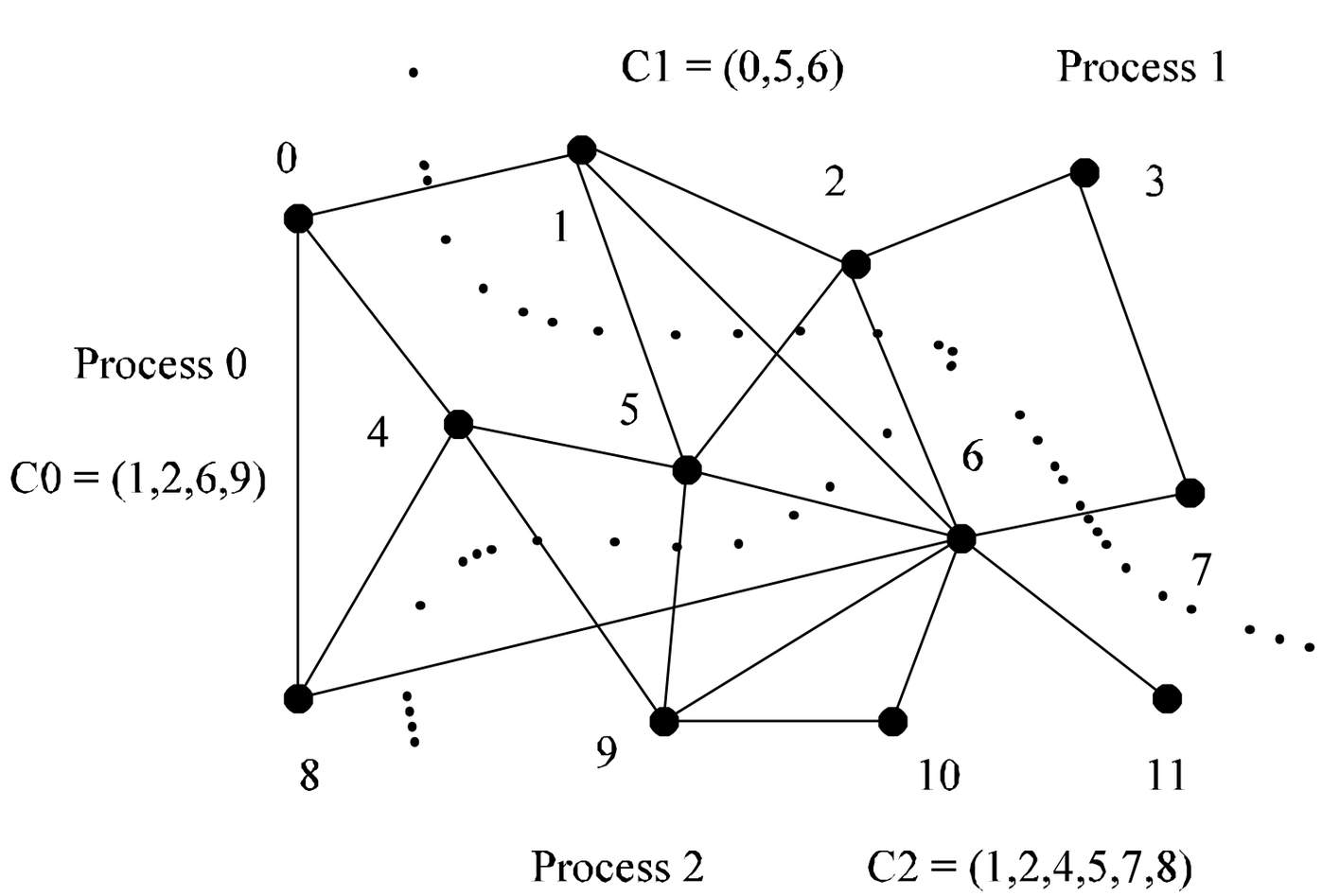
Example illustrates the dependency graph of one view of quick-sort and how it can be assigned to processes in a hypercube.



Sparse Graph

Sparse graph for computing a sparse matrix-vector product and its mapping.

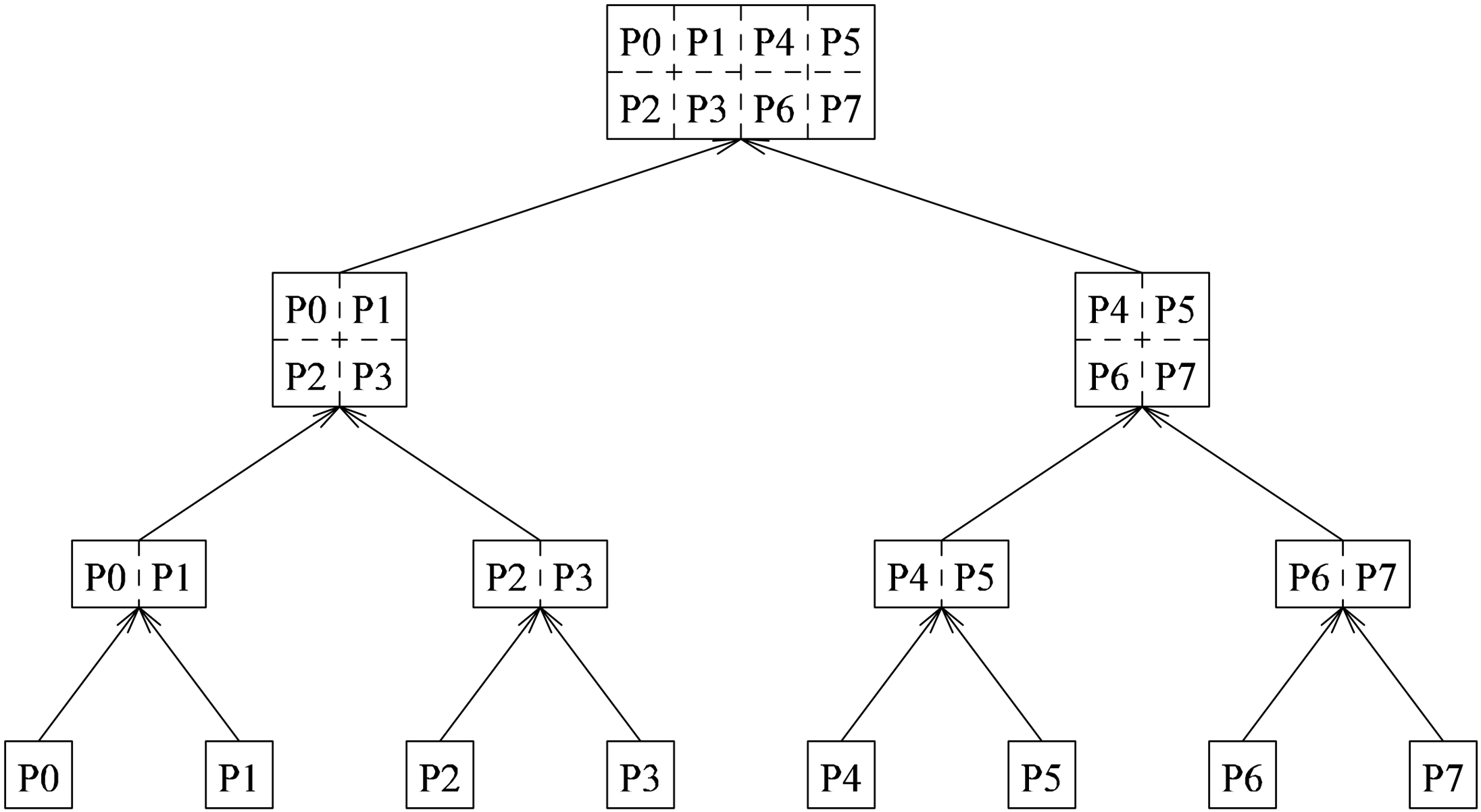




Hierarchical Mappings

- Sometimes a single mapping technique is inadequate.
- For example, the task mapping of the binary tree (quicksort) cannot use a large number of processors.
- For this reason, task mapping can be used at the top level and data partitioning within each level.

An example of task partitioning at top level with data partitioning at the lower level.



Schemes for Dynamic Mapping

- Dynamic mapping is sometimes also referred to as dynamic load balancing, since load balancing is the primary motivation for dynamic mapping.
- Dynamic mapping schemes can be centralized or distributed.

Centralized Dynamic Mapping

- Processes are designated as masters or slaves.
- When a process runs out of work, it requests the master for more work.
- When the number of processes increases, the master may become the bottleneck.
- To alleviate this, a process may pick up a number of tasks (a chunk) at one time. This is called Chunk scheduling.
- Selecting large chunk sizes may lead to significant load imbalances as well.
- A number of schemes have been used to gradually decrease chunk size as the computation progresses.

Distributed Dynamic Mapping

- Each process can send or receive work from other processes.
- This alleviates the bottleneck in centralized schemes.
- There are four critical questions: how are sensing and receiving processes paired together, who initiates work transfer, how much work is transferred, and when is a transfer triggered?
- Answers to these questions are generally application specific. We will look at some of these techniques later in this class.

Minimizing Interaction Overheads

Xi'an Jiaotong-Liverpool University
西交利物浦大學

- Maximize data locality: Where possible, reuse intermediate data. Restructure computation so that data can be reused in smaller time windows.
- Minimize volume of data exchange: There is a cost associated with each word that is communicated. For this reason, we must minimize the volume of data communicated.
- Minimize frequency of interactions: There is a startup cost associated with each interaction. Therefore, try to merge multiple interactions to one, where possible.
- Minimize contention and hot-spots: Use decentralized techniques, replicate data where necessary.

Minimizing Interaction Overheads (continued)

- Overlapping computations with interactions: Use non-blocking communications, multithreading, and prefetching to hide latencies.
- Replicating data or computations.
- Using group communications instead of point-to-point primitives.
- Overlap interactions with other interactions.

Parallel Algorithm Models

An algorithm model is a way of structuring a parallel algorithm by selecting a decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

- Data Parallel Model: Tasks are statically (or semi-statically) mapped to processes and each task performs similar operations on different data.
- Task Graph Model: Starting from a task dependency graph, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs.

Parallel Algorithm Models (continued)

- Master-Slave Model: One or more processes generate work and allocate it to worker processes. This allocation may be static or dynamic.
- Pipeline / Producer-Comsumer Model: A stream of data is passed through a succession of processes, each of which perform some task on it.
- Hybrid Models: A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.

Gallery of patterns and models

Xi'an Jiaotong-Liverpool University 西交利物浦大學

- Decomposition Patterns
  - Data (DD)
  - Recursive (RD)
  - *Exploratory
  - *Speculative
  - *Hybrid
- Mapping Patterns
  - Static
    - Block distribution (BD)
    - *Others
  - Dynamic
    - Centralized Scheme (CS)
      - Worker pool (WP)
      - Master-slave (MS)
    - *Distributed Scheme

- Models
  - Embarrassingly parallel

    = DD + BD/CS
  - Fork–join

    = RD + BD/CS
  - *Pipeline
  - Geometric

    = DD + BD

* Will not be discussed in this course

Embarrassingly parallel

Xi'an Jiaotong-Liverpool University
西交利物浦大學

- There is little or no dependency or need for communication between its parallel tasks.
- Example:
  - Monte Carlo sampling
    Relational database queries
    Brute force search, grid search
    Predict many instance by one model
    Random forest

Data parallelism

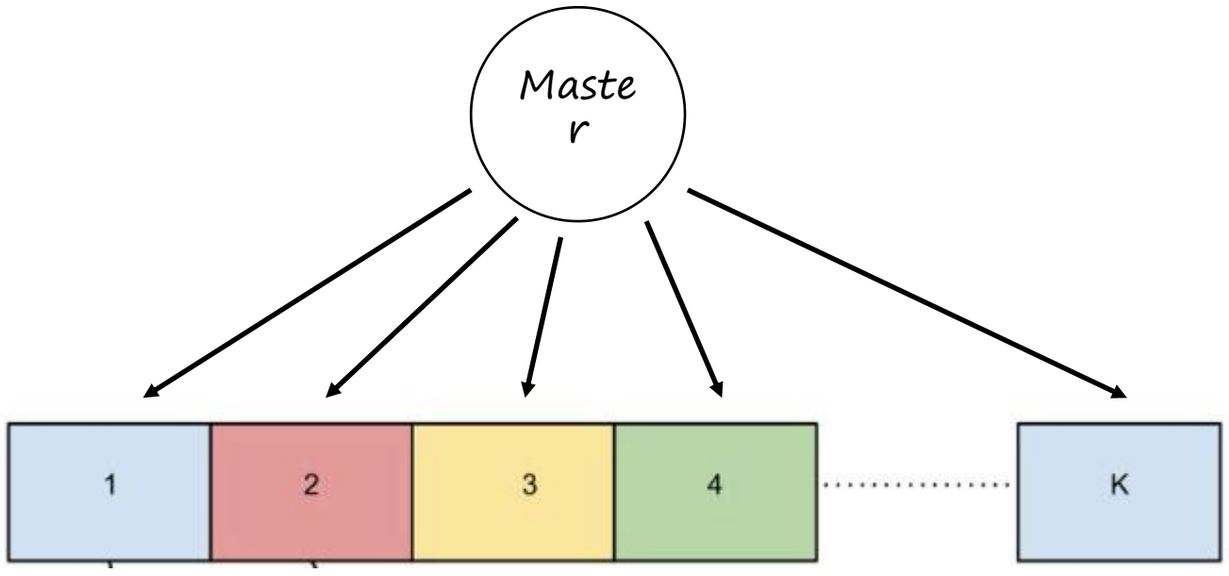
Xi'an Jiaotong-Liverpool University
西交利物浦大學

```
A = [1, 2, 3, 4, 5]

print(list(map(f, A)))
```

Any function to apply

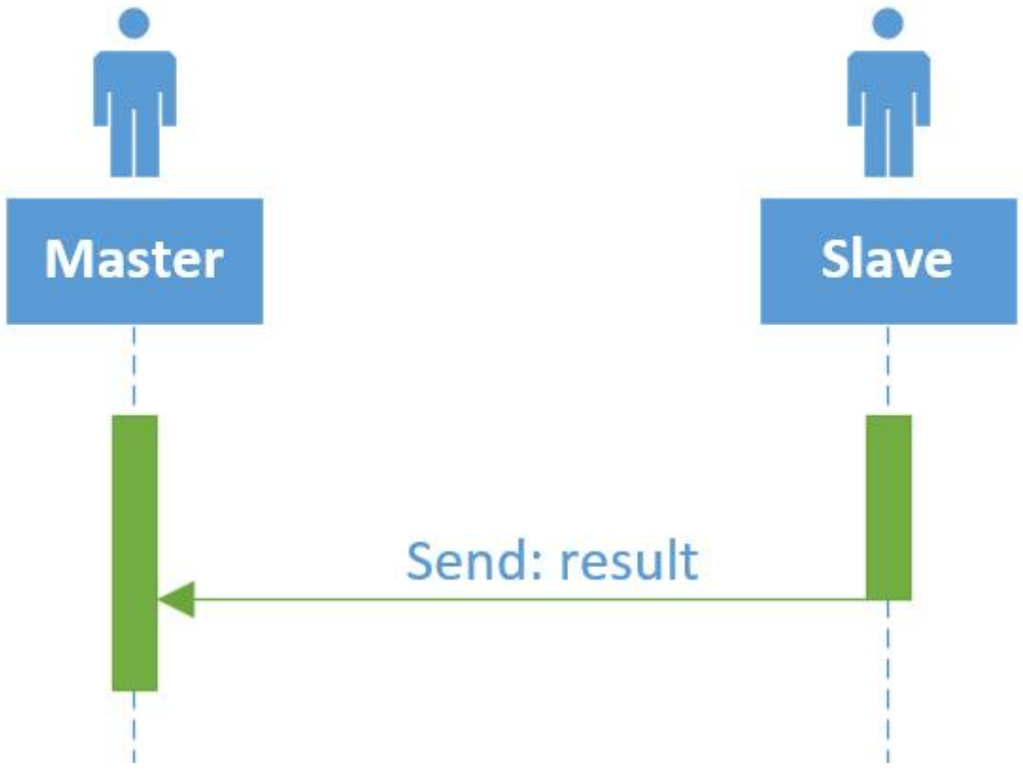
- Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array.
- In data parallel operations, the source collection is partitioned so that multiple threads can operate on different segments concurrently.

Master-slave Model



- There are 1 master and K slaves
- Master splits the data into K portions, and assigns them to K slaves
  - This can also be the input parameter of processing function
- After the calculation is completed, the result is returned to the master for the reduce operation

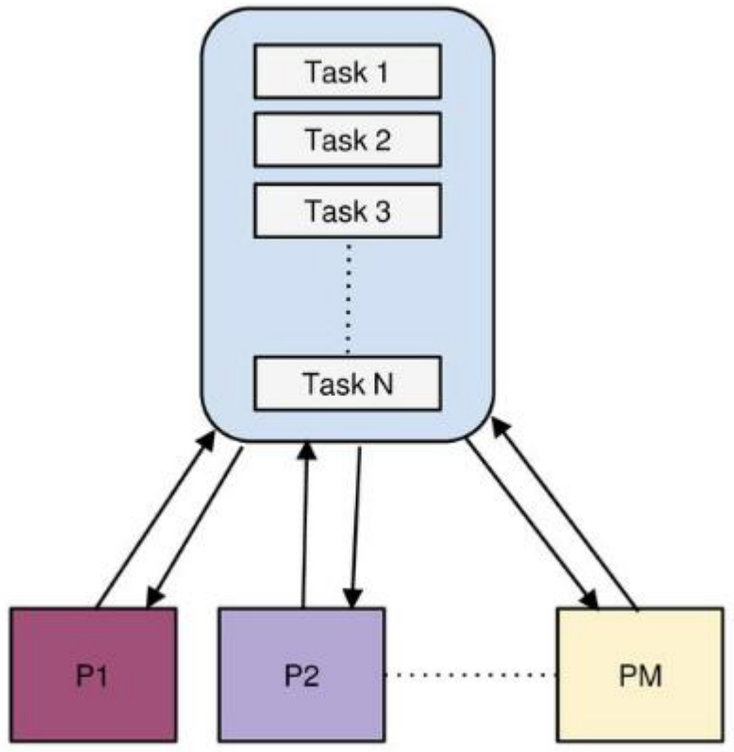
Interactive diagram and code



- Please extend the program to more processes (task1)

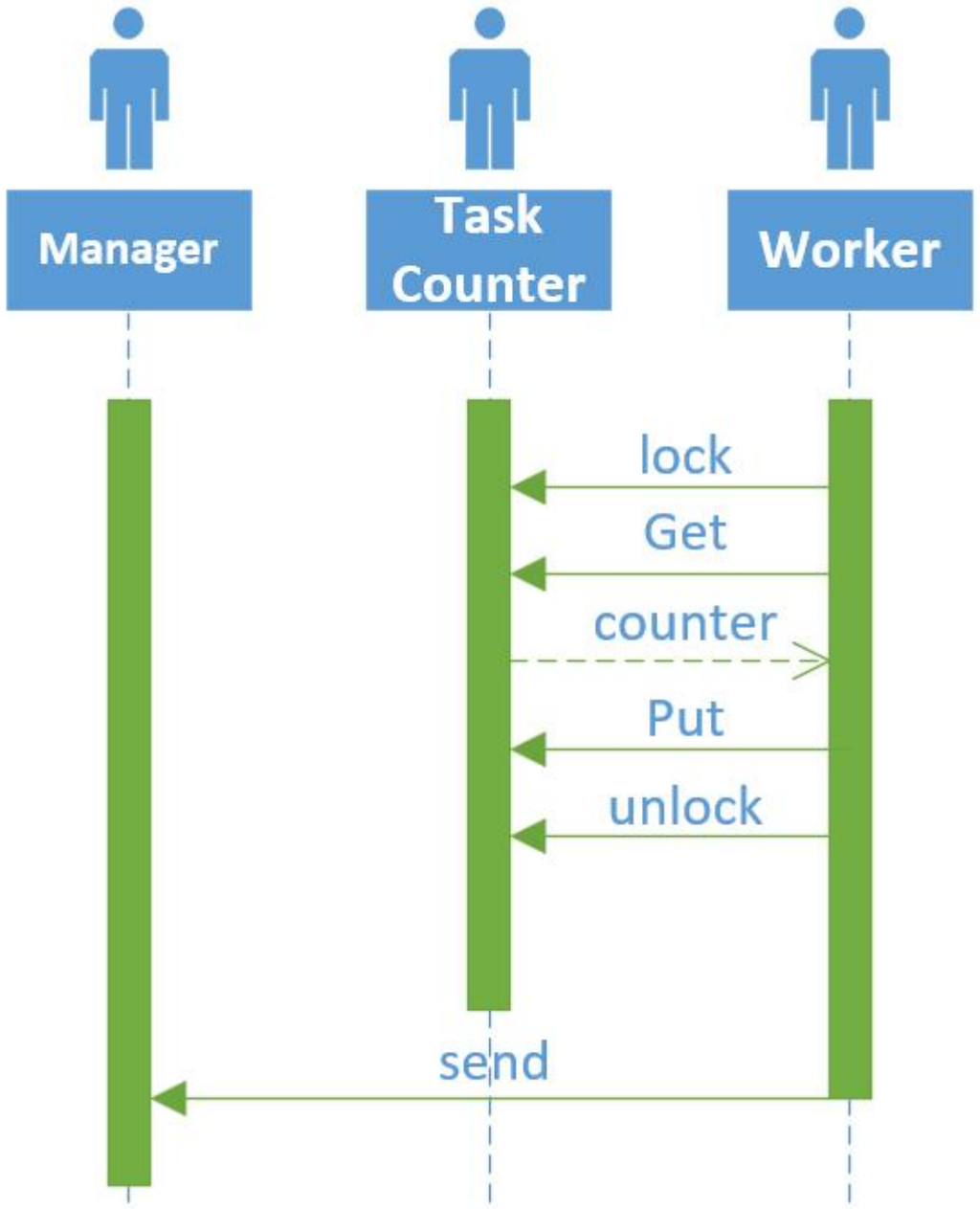
```python
import numpy as np
from mpi4py import MPI

from util import calc_pi, hit_circle

# environment info
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nproc = comm.Get_size()

# number of tasks
T = nproc - 1

if rank == 0:  # master

    assert nproc == 2

    # recv result
    result = comm.recv()

    print(f'proc {rank}:result={result}')

else:  # slave

    # send result back
    comm.send(rank, dest=0)
    print(f'proc {rank}:sent!')
```

Workpool Model



- There are 1 manager and M worker
- Workers fetch their own task from a task pool. It must determine by itself whether there are still tasks and update the status of the tasks it has processed.
- After the calculation is completed, it is returned to the manager for the reduce operation

Interactive diagram and code




```python
import sys

import numpy as np
from mpi4py import MPI

from util import hit_circle, calc_pi

# environment info
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nproc = comm.Get_size()

# allocate window for completed tasks
datatype = MPI.INT
itemsize = datatype.Get_size()  # get size of datatype

num_tasks_done = np.array(0, dtype='i')  # buffer
N = num_tasks_done.size

win_size = N * itemsize if rank == 0 else 0
win = MPI.Win.Allocate(win_size, comm=comm)  # allocate window

if rank == 0:  # manager

    assert nproc == 2

    # recv result
    result = comm.recv()

    print(f'proc {rank}:result={result}')

else:  # worker

    win.Lock(rank=0)
    win.Get(num_tasks_done, target_rank=0)
    win.Put(num_tasks_done, target_rank=0)
    win.Unlock(rank=0)

    # send result back
    comm.send(num_tasks_done, dest=0)  # 同步
    print(f'proc {rank}:task {num_tasks_done} done!')

win.Free()
```

Explain later...

- Please extend the program to more processes and add logic of judging whether all tasks are completed (task3)

Section

- MPI One-side Communication

One-Sided Communications

- One-sided communications
  - Allows the communication process to be invisible to one of the parties, that is, it's ok to write related code on one side
- Advantage:
  - Simplified procedures, just like shared storage, can be read and written directly, eliminating the need to transfer data back and forth
    Leverage underlying hardware support, if available
- Notes
  - It is necessary to be mutually exclusive, especially write. Thus, you need to set a lock.

Main steps


```
10    rank = comm.Get_rank()
11    nproc = comm.Get_size()
12
13    # allocate window for completed tasks
14    datatype = MPI.INT
15    itemsize = datatype.Get_size()  # get size of datatype
16
17    num_tasks_done = np.array(0, dtype='i')  # buffer
18    N = num_tasks_done.size
19
20    win_size = N * itemsize if rank == 0 else 0
21    win = MPI.Win.Allocate(win_size, comm=comm)  # allocate window
22
23    if rank == 0:  # manager
```

- Create a window
  - The window length here can only be calculated in bytes, so the size of the data structure must be estimated
- The key data structure (num_tasks_done) here is stored in the rank=0 process, so it only needs to be allocated in this process; But other processes also declare the window, with the size set to 0

Main steps


```
31
32      else:  # worker
33
34          win.Lock(rank=0)
35          win.Get(num_tasks_done, target_rank=0)
36          win.Put(num_tasks_done, target_rank=0)
37          win.Unlock(rank=0)
38
39          # send result back
40          comm.send(num_tasks_done, dest=0)  # 同步
41          print(f'proc {rank}:task {num_tasks_done} done!')
42
43      win.Free()
44
```

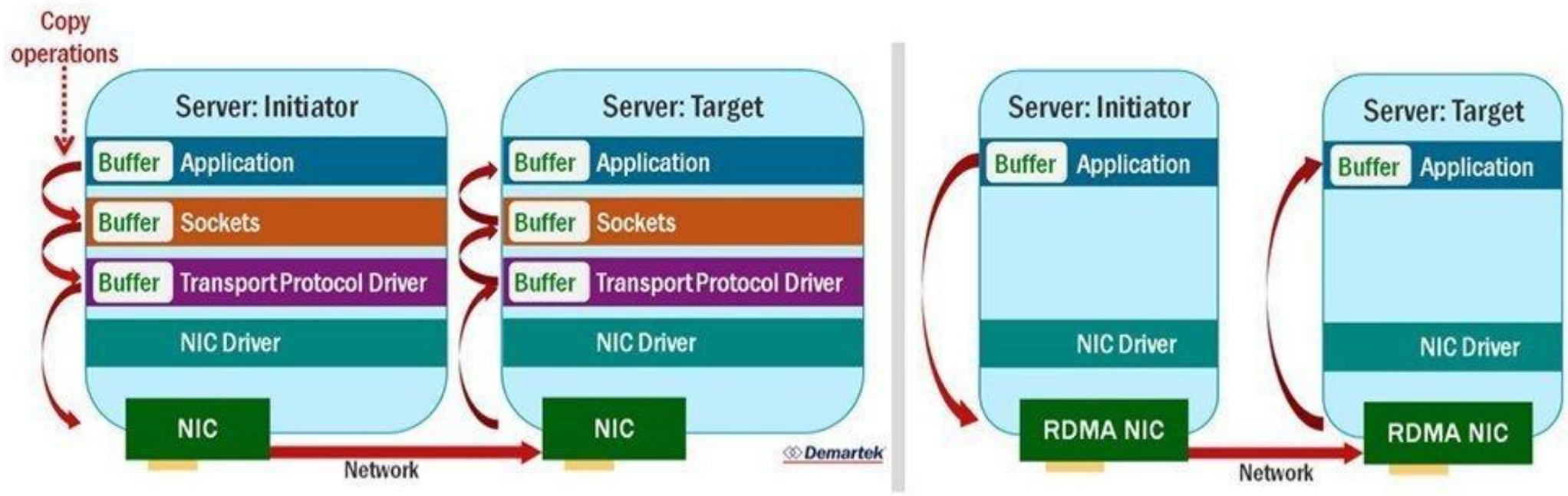
- Get exclusive access
  - only the origin process calls the Win.Lock and Win.Unlock methods. Locks are used to protect remote accesses to the locked remote window and to protect local load/store accesses to a locked local window.

Main steps


```
31
32  else:  # worker
33
34      win.Lock(rank=0)
35      win.Get(num_tasks_done, target_rank=0)
36      win.Put(num_tasks_done, target_rank=0)
37      win.Unlock(rank=0)
38
39      # send result back
40      comm.send(num_tasks_done, dest=0)  #
41      print(f'proc {rank}:task {num_tasks_done} done!')
42
43  win.Free()
44
```

- Remote write, read and reduction are available through calling the methods Win.Put, Win.Get, and Win.Accumulate respectively within a Win instance.
- These methods need an integer rank identifying the target process and an integer offset relative the base address of the remote memory block being accessed.
- Note: After calling Win.Pet, the num_tasks_done will not be updated immediately! The num_tasks_done will be updated only after Win.Unlock is called.

*Hardware support



- Bypasses network protocol-related computation
  The burden on the CPU is lightened because the network card takes over the transfer process