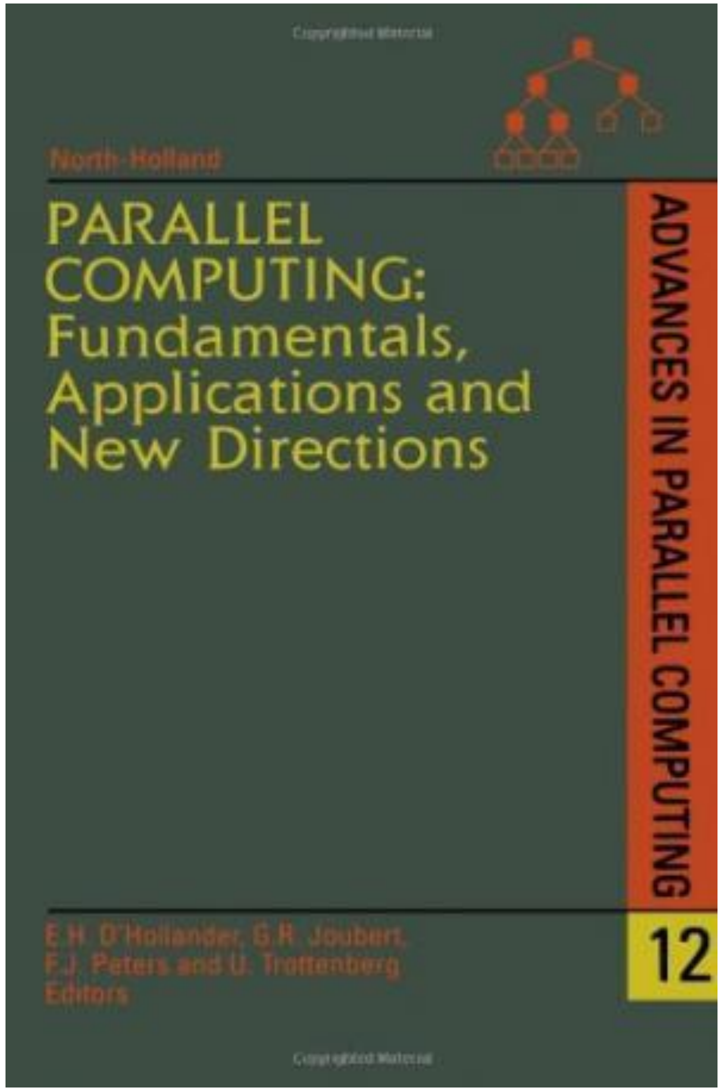
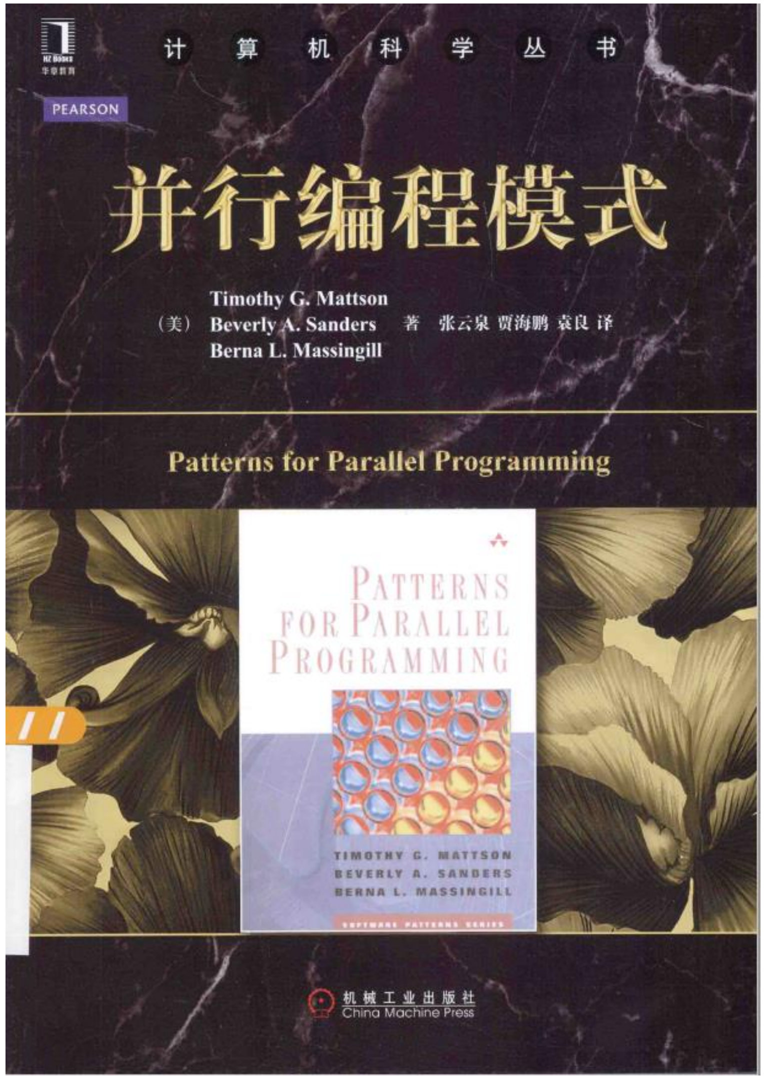
大纲

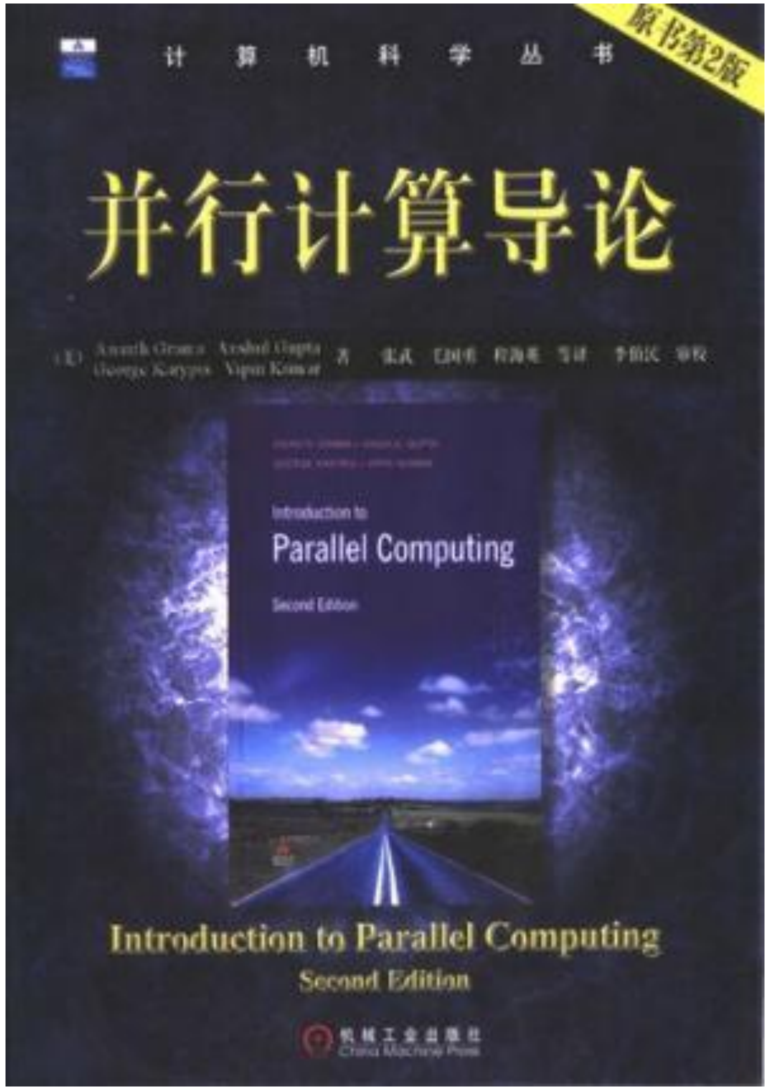
- **前言**
  - 为什么需要“大规模计算” [HPC, DL, Business platform system, Cloud已经合流]
    - 导入 – 科学计算(天气预报)，DL, 互联网平台(Google, Amazon, Alibaba, MeiTuan, …)
- **基础篇**
  - 并发程序的样子 – Divide & Conquer, Model & Challenges, PCAM, Data/Task, …
    - 天气预报的计算
  - 运行环境
    - 硬件 – 自己梳理的3个方案 – Shared/Unshared Memory, Hybrid
    - 系统软件 – 协议栈, Modern OS, Distributed Job Scheduler, GTM等
- **算法级篇**
  - OpenMP, MPI, CUDA (DL的实现), Big Data 中的MR/Spark等 (只涉及在Big Data SDK之上的编程；大数据本身的介绍放到后一部分)
- **系统级篇 – 互联网平台的实现**
  - “秒杀”的技术架构
  - 计算广告
  - 系统架构 (HTAP等)
    - Flink, ClickHouse, MaxCompute, ELK …


Copyrighted Material
North-Holland
PARALLEL COMPUTING: Fundamentals, Applications and New Directions
ADVANCES IN PARALLEL COMPUTING
12
E.H. D'Hollander, G.R. Joubert, F.J. Peters and U. Trottenberg
Editors
Copyrighted Material

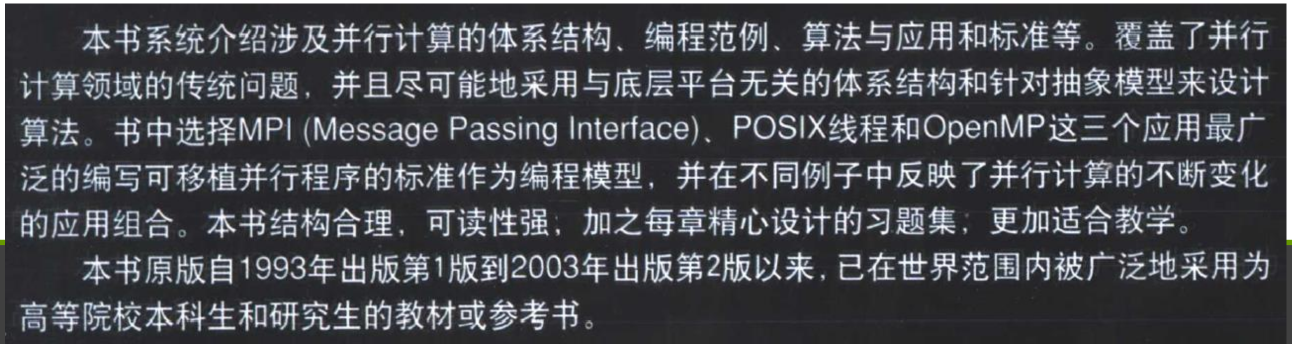
- Parallel Computing: Fundamentals, Applications and New Directions
- *E.H. D'Hollander*, *F.J. Peters*, *G.R. Joubert*, *U. Trottenberg and R. Völpel (Eds.)*
- North Holland
- 1998



- 并行编程模式
- *Timothy G. Mattson*, *Beverly A. Sanders*, *Berna L. Massingill*
- 本书介绍了并行编程模式的相关概念和技术，主要内容包括并行编程模式语言、并行计算的背景、软件开发中的并发性、并行算法结构设计、支持结构、设计的实现机制以及OpenMP、MPI等。本书可供软件专业的本科生或研究生使用，同时也可供从事软件开发工作的广大技术人员参考。
- 2015



- 并行计算导论
- *Ananth Grama*, *George Karypis*, *张武*, *毛国勇*, *Anshul Gupta*, *Vipin Kumar*, *程海英*
  - 《并行计算导论》(原书第2版)全面介绍并行计算的各个方面，包括体系结构、编程范例、算法与应用和标准等，涉及并行计算的新技术，也覆盖了较传统的算法，如排序、搜索、图和动态编程等。《并行计算导论》(原书第2版)尽可能采用与底层平台无关的体系结构并且针对抽象模型来设计处落地。书中选择MPI、POSIX线程和OpenMP作为编程模型，并在不同例子中反映了并行计算的不断变化的应用组合
- **2005**

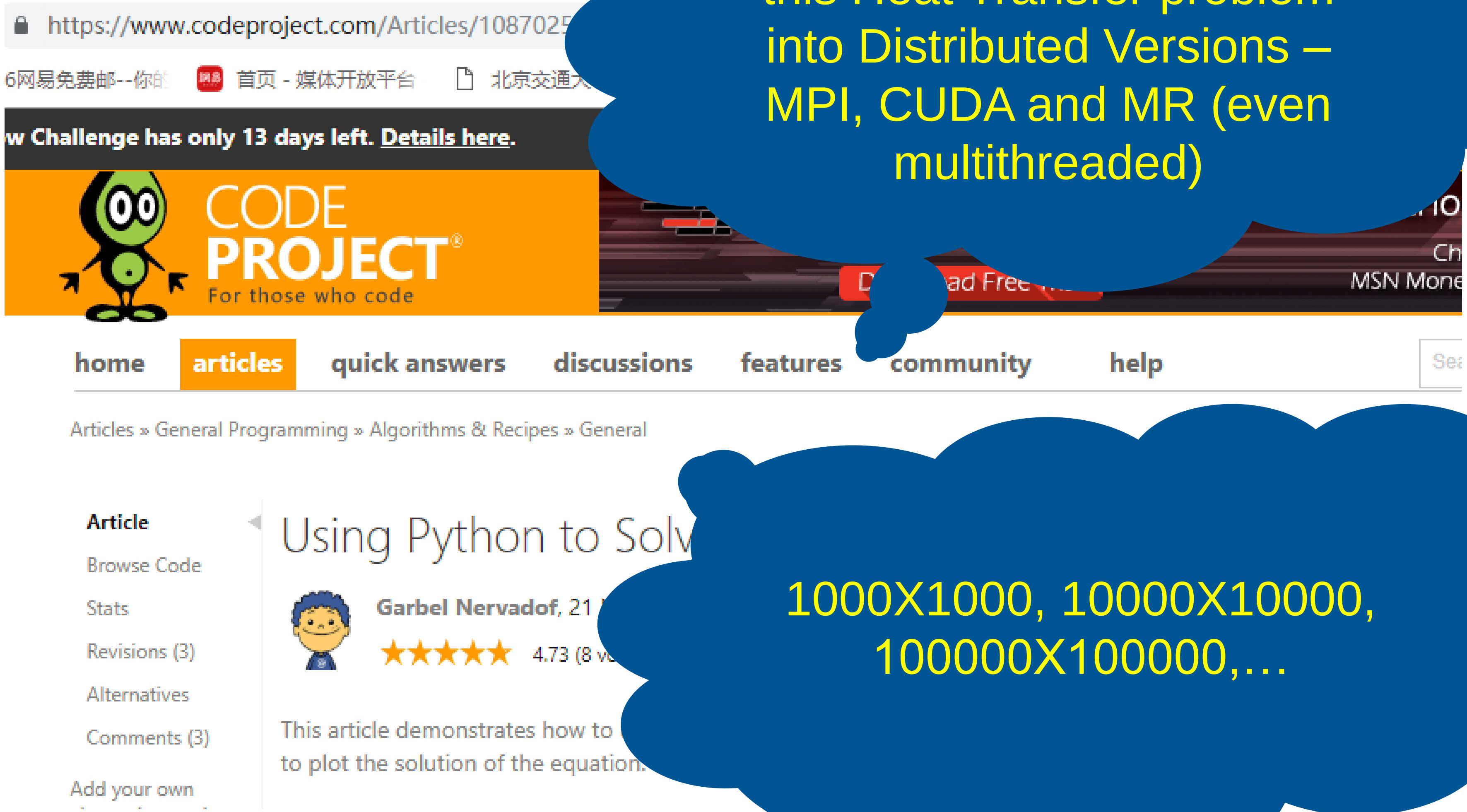

本书系统介绍涉及并行计算的体系结构、编程范例、算法与应用和标准等。覆盖了并行计算领域的传统问题，并且尽可能地采用与底层平台无关的体系结构和针对抽象模型来设计算法。书中选择MPI (Message Passing Interface)、POSIX线程和OpenMP这三个应用最广泛的编写可移植并行程序的标准作为编程模型，并在不同例子中反映了并行计算的不断变化的应用组合。本书结构合理，可读性强，加之每章精心设计的习题集，更加适合教学。

本书原版自1993年出版第1版到2003年出版第2版以来，已在世界范围内被广泛地采用为高等院校本科生和研究生的教材或参考书。

Chapter 02: HPC with example

- **Faster for larger data**
  - Sequential implementation with Python
    - "Using Python to Solve Computational Physics Problems"
  - Ideas to convert Sequential to Parallel
    - Shared Memory programming, Distributed Memory programming
    - Hint to get the **EU**s for the Heat Equation
  - Measure the performance

Hints


https://www.codeproject.com/Articles/1087025

Articles » General Programming » Algorithms & Recipes » General

Using Python to Solv

Garbel Nervadof, 21

★★★★★ 4.73 (8 v

This article demonstrates how to
to plot the solution of the equation.

You're required to implement this Heat Transfer problem into Distributed Versions – MPI, CUDA and MR (even multithreaded)

1000X1000, 10000X10000, 100000X100000,…

We have know Solution Technique

- **A grid is used to divide the region of interest.**
  - Since the PDE is satisfied at each point in the area, it must be satisfied at each point of the grid.
- **A finite difference approximation is obtained at each grid point.**

$$\frac{\partial^2\ T(x,y)}{\partial x^2} \approx \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2}, \quad \frac{\partial^2\ T(x,y)}{\partial y^2} \approx \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2}$$

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

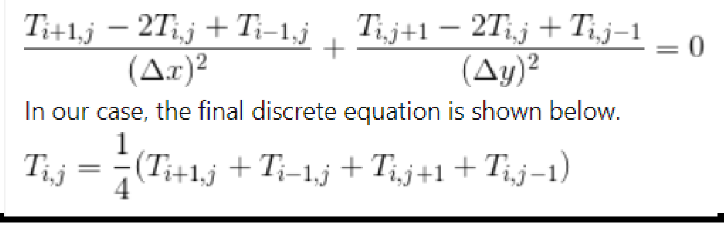
In our case, the final discrete equation is shown below.

$$T_{i,j} = \frac{1}{4}(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1})$$

□ The code demonstration of “Using Python to Solve Computational Physics Problems”

1. **Configure the parameters**
   - GRID
     - ➢ With Initial values [初始值]
     - ➢ Boundary conditions
       - ✓ [边界条件]
   - Termination condition
     - ➢ Iteration number or Epsilon

```python
import numpy as np
# Set Dimension and delta
lenX = lenY = 100 #we set it rectangular
delta = 1
# Initial guess of interior grid
Tguess = 0

# Set meshgrid
X, Y = np.meshgrid(np.arange(0, lenX),
np.arange(0, lenY))

# Set array size and set the interior value with Tguess
T = np.empty((lenX, lenY))
T.fill(Tguess)
```



$$\frac{T_{i+1,j}-2T_{i,j}+T_{i-1,j}}{(\Delta x)^2}+\frac{T_{i,j+1}-2T_{i,j}+T_{i,j-1}}{(\Delta y)^2}=0$$

In our case, the final discrete equation is shown below.

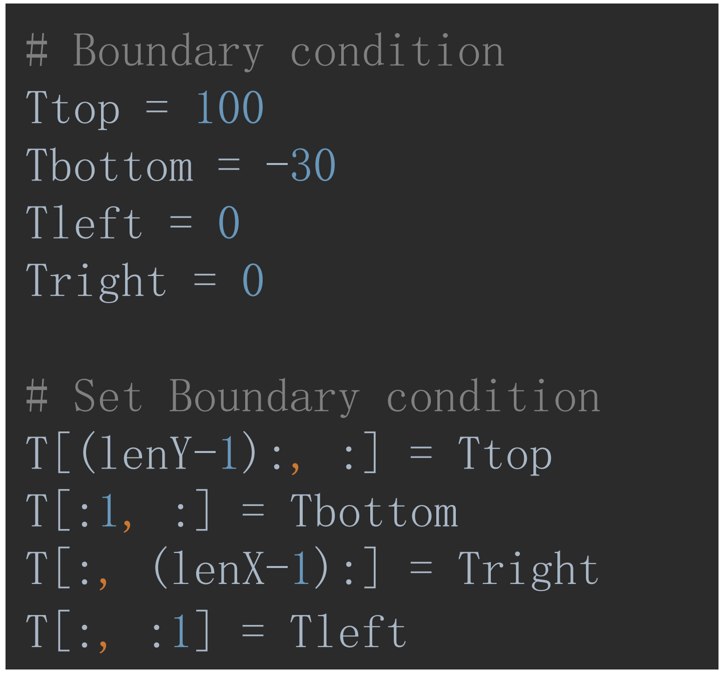
$$T_{i,j}=\frac{1}{4}(T_{i+1,j}+T_{i-1,j}+T_{i,j+1}+T_{i,j-1})$$

☐ The code demonstration of "Using Python to Solve Computational Physics Problems"

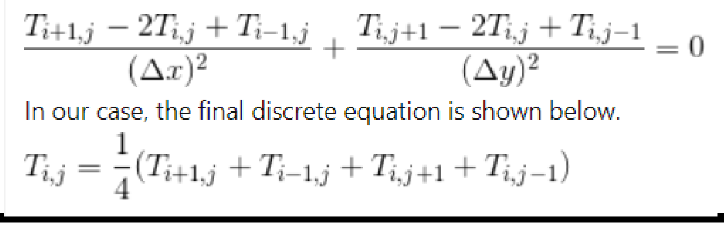
1. Configure the parameters

- GRID
  - ➢ With Initial values [初始值]
  - ➢ With Boundary conditions
    - ✓ [边界条件]

- Termination condition
  - ➢ Iteration number or Epsilon



```
# Boundary condition
Ttop = 100
Tbottom = -30
Tleft = 0
Tright = 0

# Set Boundary condition
T[(lenY-1):, :] = Ttop
T[:1, :] = Tbottom
T[:, (lenX-1):] = Tright
T[:, :1] = Tleft
```



$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

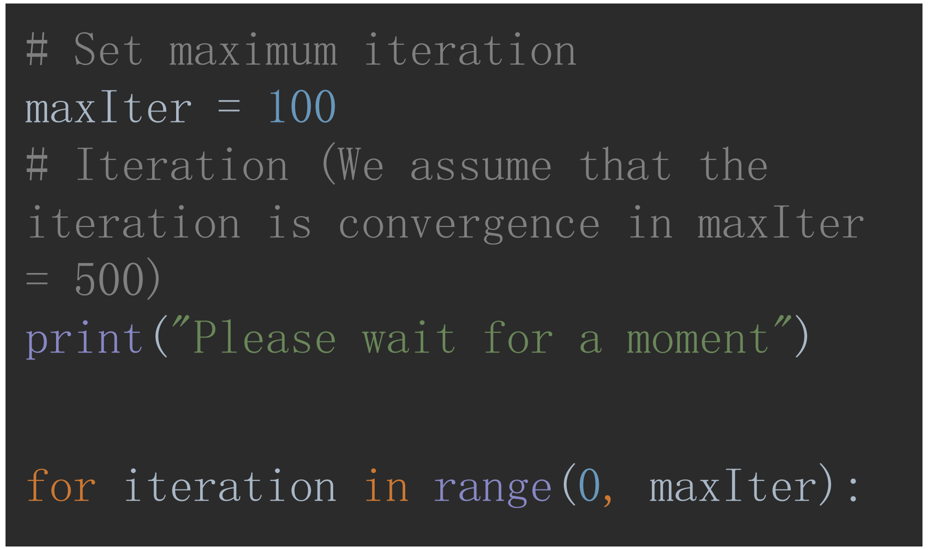
In our case, the final discrete equation is shown below.

$$T_{i,j} = \frac{1}{4}(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1})$$

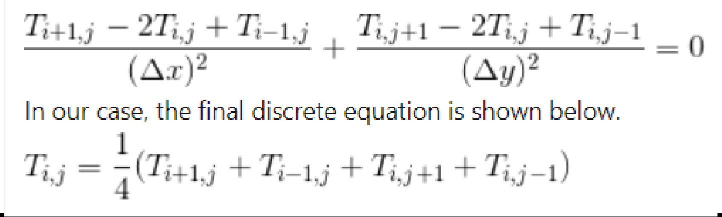
☐ The code demonstration of "Using Python to Solve Computational Physics Problems"

1. Configure the parameters

- GRID
  - ➢With Initial values [初始值]
  - ➢With Boundary conditions
    - ✓[边界条件]
- Termination condition
  - ➢Iteration number or Epsilon



```
# Set maximum iteration
maxIter = 100
# Iteration (We assume that the
iteration is convergence in maxIter
= 500)
print("Please wait for a moment")


for iteration in range(0, maxIter):
```



$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

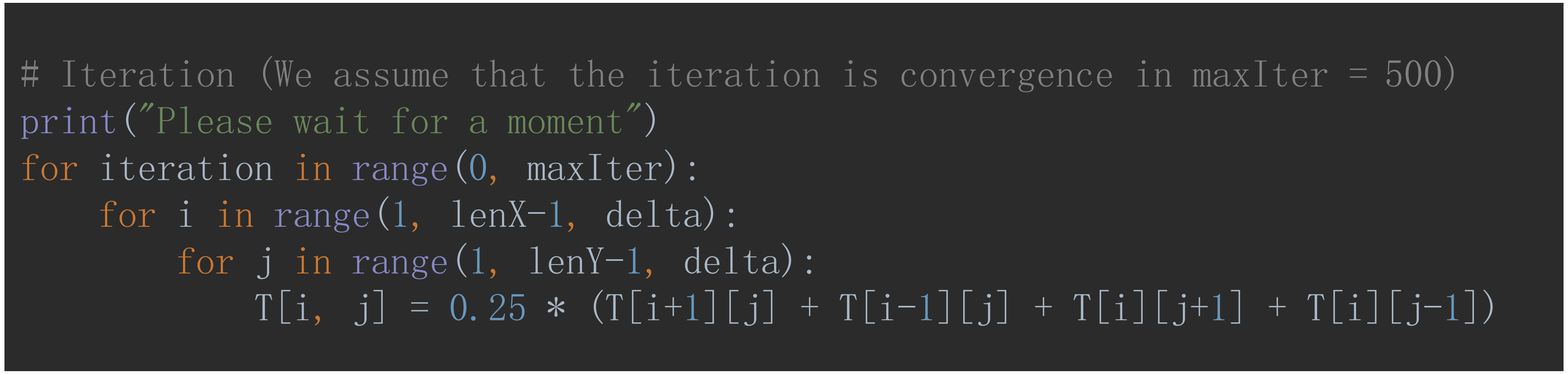
In our case, the final discrete equation is shown below.

$$T_{i,j} = \frac{1}{4}(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1})$$

□ The code demonstration of "Using Python to Solve Computational Physics Problems"

2. Iterative updating

- Use "Termination condition" to control the updating of the internal vertices



```
# Iteration (We assume that the iteration is convergence in maxIter = 500)
print("Please wait for a moment")
for iteration in range(0, maxIter):
    for i in range(1, lenX-1, delta):
        for j in range(1, lenY-1, delta):
            T[i, j] = 0.25 * (T[i+1][j] + T[i-1][j] + T[i][j+1] + T[i][j-1])
```

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

In our case, the final discrete equation is shown below.

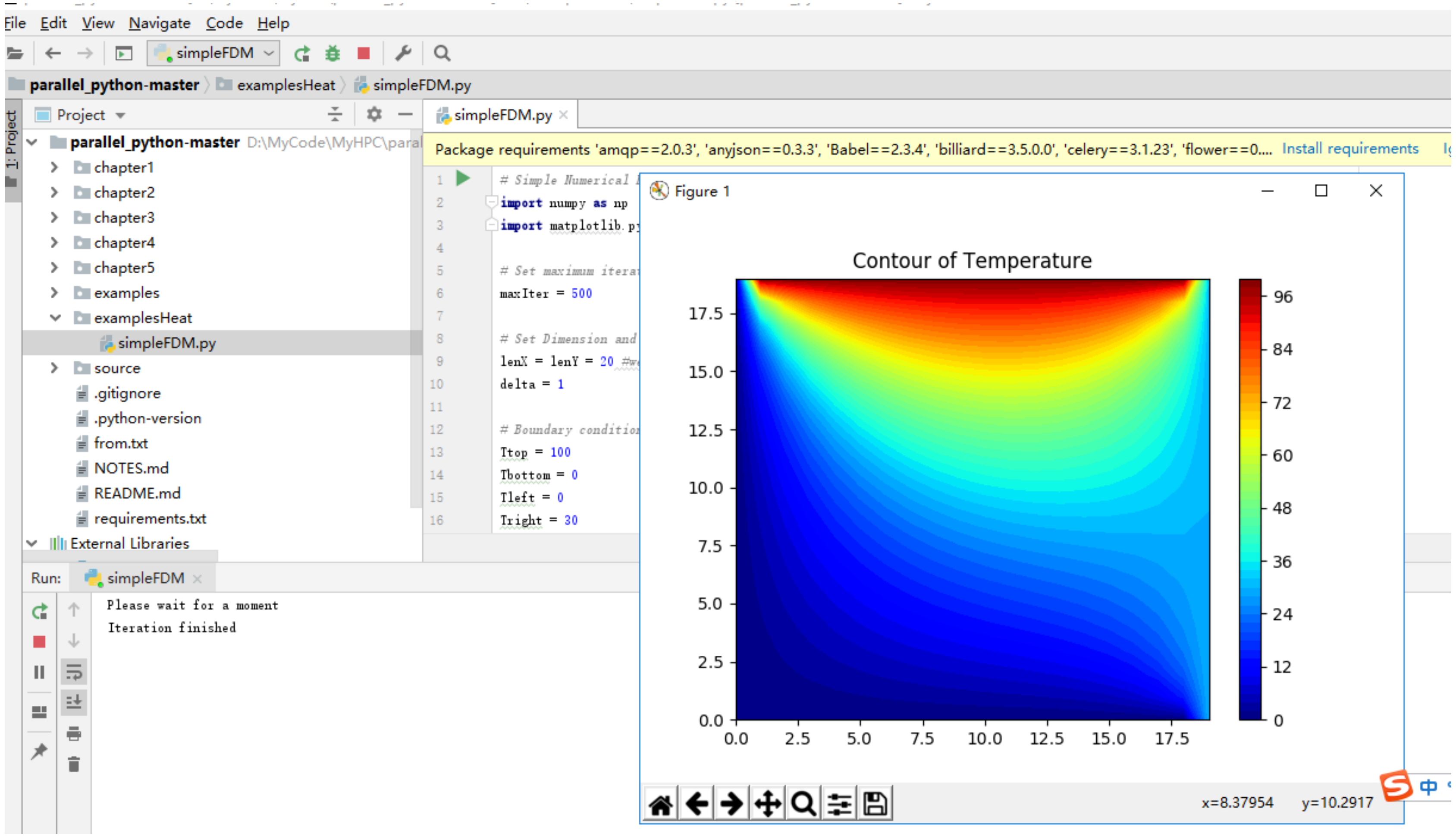
$$T_{i,j} = \frac{1}{4}(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1})$$

☐ The code demonstration of "Using Python to Solve Computational Physics Problems"

3. Visualize the dynamics

```
# Set colour interpolation and colour map
colorinterpolation = 100
colourMap = plt.cm.jet #you can try: colourMap = plt.cm.coolwarm

<<Repeated updating>>

# Configure the contour
plt.title("Contour of Temperature")
plt.contourf(X, Y, T, colorinterpolation, cmap=colourMap)

# Set Colorbar
plt.colorbar()

# Show the result in the plot window
plt.show()
```

Copy the code into PyCharm project



Package requirements 'amqp==2.0.3', 'anyjson==0.3.3', 'Babel==2.3.4', 'billiard==3.5.0.0', 'celery==3.1.23', 'flower==0....

```
# Simple Numerical
import numpy as np
import matplotlib.p

# Set maximum itera
maxIter = 500

# Set Dimension and
lenX = lenY = 20 #w
delta = 1

# Boundary conditio
Ttop = 100
Tbottom = 0
Tleft = 0
Tright = 30
```

```
Please wait for a moment
Iteration finished
```

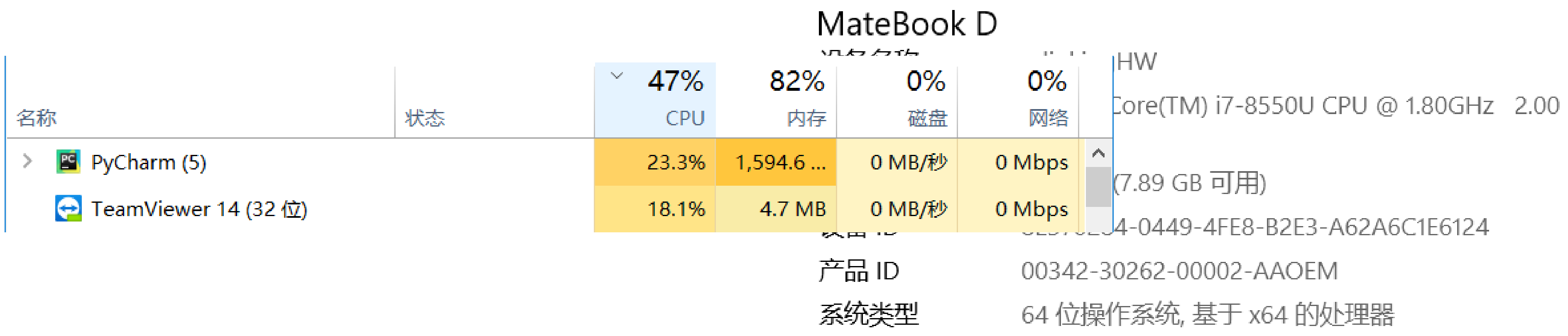
Small challenge

- **Define and use Epsilon to control the repetition?**

- **Hint:**
  - Use the **matrix norm**

Run the program with different scales

- When the “maxIter = 100000”, the program takes almost 40 minutes!
- When “lenX = lenY = 10000” + “maxIter = 1000”, it takes 64832 secs = **18 hours**!



MateBook D

名称	状态	47% CPU	82% 内存	0% 磁盘	0% 网络
PyCharm (5)		23.3%	1,594.6 ...	0 MB/秒	0 Mbps
TeamViewer 14 (32 位)		18.1%	4.7 MB	0 MB/秒	0 Mbps

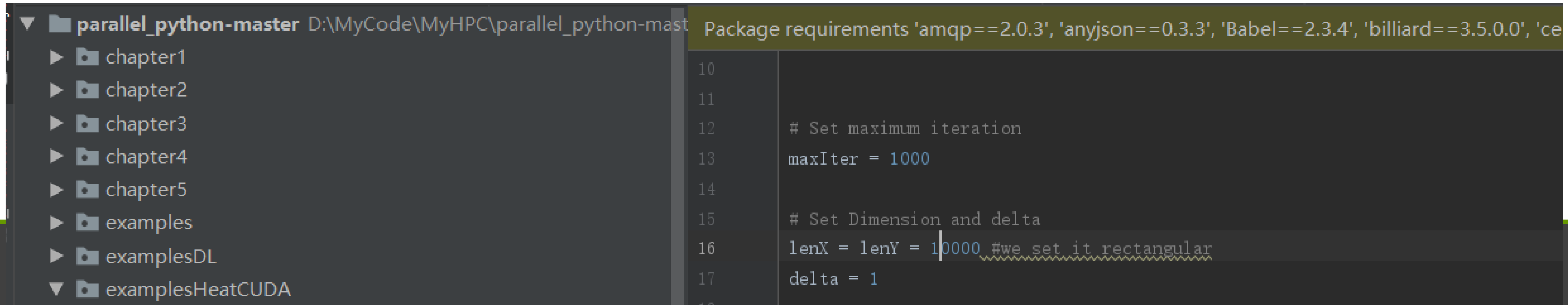
HW

Core(TM) i7-8550U CPU @ 1.80GHz 2.00

(7.89 GB 可用)

4-0449-4FE8-B2E3-A62A6C1E6124

产品 ID	00342-30262-00002-AAOEM
系统类型	64 位操作系统, 基于 x64 的处理器



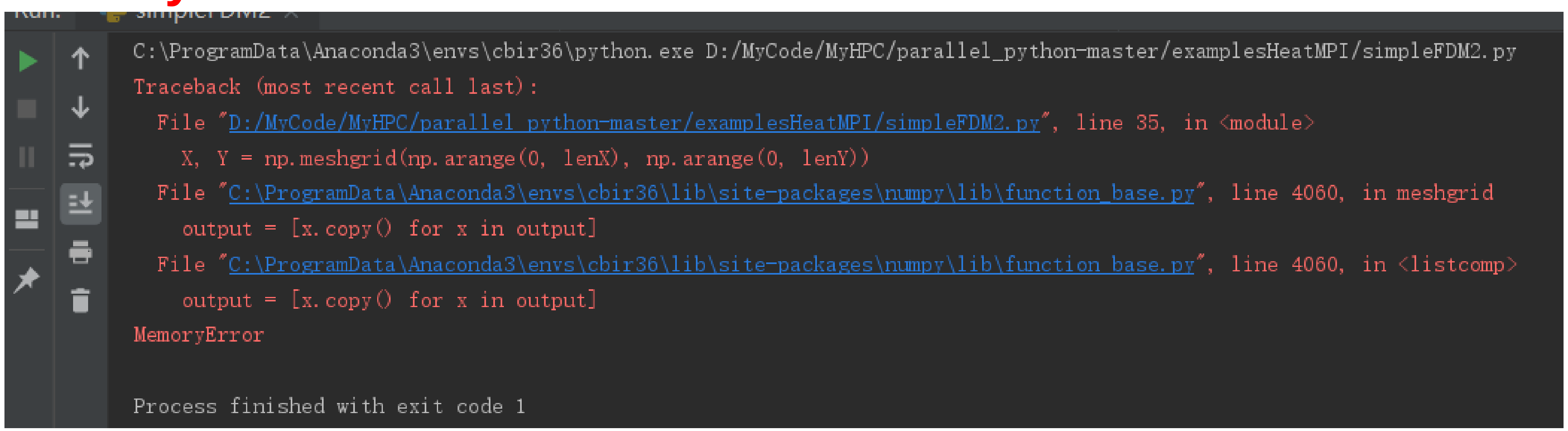
parallel_python-master D:\MyCode\MyHPC\parallel_python-mast
- chapter1
- chapter2
- chapter3
- chapter4
- chapter5
- examples
- examplesDL
- examplesHeatCUDA

Package requirements 'amqp==2.0.3', 'anyjson==0.3.3', 'Babel==2.3.4', 'billiard==3.5.0.0', 'ce

```python


# Set maximum iteration
maxIter = 1000

# Set Dimension and delta
lenX = lenY = 10000 #we set it rectangular
delta = 1
```

You can try

- **Run the program with different scales**
  - When “lenX = lenY = 100000” + “maxIter = 1000”, the error of “**MemoryError**”!!



```
C:\ProgramData\Anaconda3\envs\cbir36\python.exe D:/MyCode/MyHPC/parallel_python-master/examplesHeatMPI/simpleFDM2.py
Traceback (most recent call last):
  File "D:/MyCode/MyHPC/parallel_python-master/examplesHeatMPI/simpleFDM2.py", line 35, in <module>
    X, Y = np.meshgrid(np.arange(0, lenX), np.arange(0, lenY))
  File "C:\ProgramData\Anaconda3\envs\cbir36\lib\site-packages\numpy\lib\function_base.py", line 4060, in meshgrid
    output = [x.copy() for x in output]
  File "C:\ProgramData\Anaconda3\envs\cbir36\lib\site-packages\numpy\lib\function_base.py", line 4060, in <listcomp>
    output = [x.copy() for x in output]
MemoryError

Process finished with exit code 1
```

  - How to finish the computation of the weather-forecasting for BeiJing, China, Globe?

Chapter 02: HPC with example

Faster for larger data

- Sequential implementation with Python
  - “Using Python to Solve Computational Physics Problems”
- Ideas to convert Sequential to Parallel
  - Shared Memory programming, Distributed Memory programming
  - Hint to get the **EU**s for the Heat Equation
- Measure the performance

迟学斌 中国科学院计算机网络信息中心
chi@sccas.cn, chi@sc.cnic.cn

http://lssc.cc.ac.cn/
http://www.sccas.cn/
http://www.scgrid.cn/
http://www.cngrid.org/

2005 年 4 月 6 日

并行程序设计是并行软件开发的基础之一，针对不同的并行计算机，以及不同的并行实现平台，其实现方式是不同的。在这一部分，简单介绍并行实现过程所需的基本技术。

§4.1 并行编程模式的主要类型

并行编程模式主要有如下的三种类型:

- 主从模式 *(Master-slave)*: 有一个主进程，其它为从进程。在这种模式中，主进程一般负责整个并行程序的数据控制，从进程负责对数据的处理和计算任务，当然，主进程也可以参与对数据的处理和计算。一般情况下，从进程之间不发生数据交换，数据的交换过程是通过主进程来完成的。
- 对称模式 *(SPMD)*: 在这种编程模式中，没有哪个进程是主进程，每个进程的地位是相同的。然而，在并行实现过程中，我们总是要在这些进程中选择一个进行输入输出的进程，它扮演的角色和主进程类似。
- 多程序模式 *(MPMD)*: 在每个处理机上执行的程序可能是不同的，在某些处理机上可能执行相同的程序。

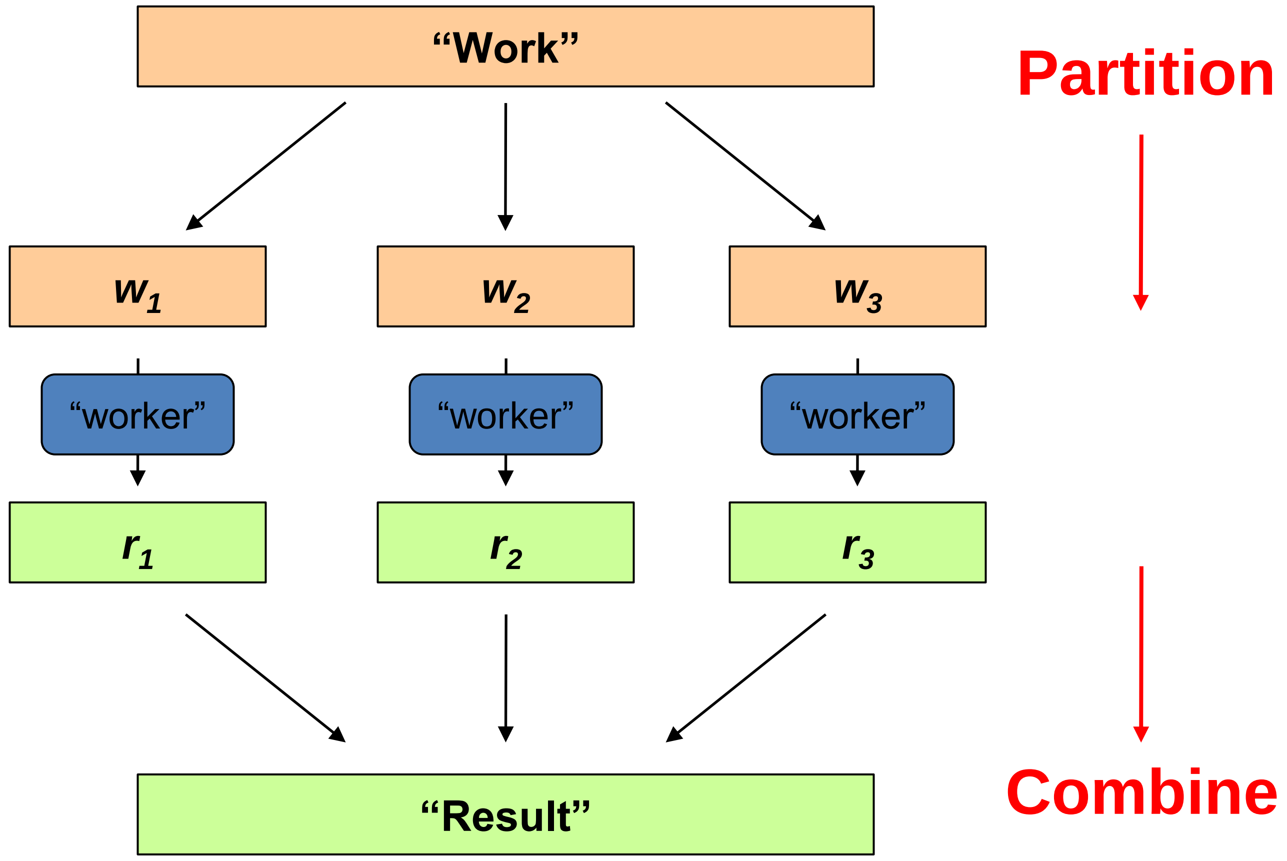
迟学斌 中国科学院计算机网络信息中心
chi@sccas.cn, chi@sc.cnic.cn

http://lssc.cc.ac.cn/
http://www.sccas.cn/
http://www.scgrid.cn/
http://www.cngrid.org/

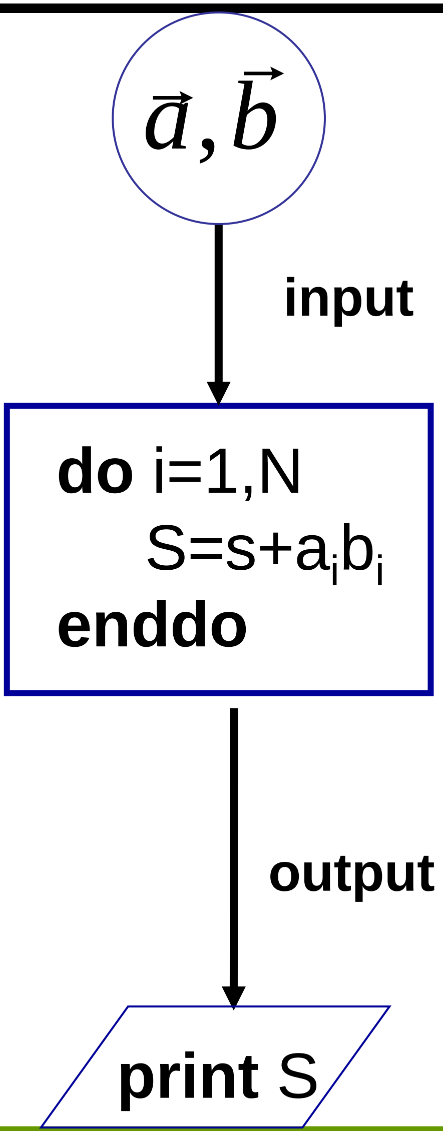
2005 年 4 月 6 日

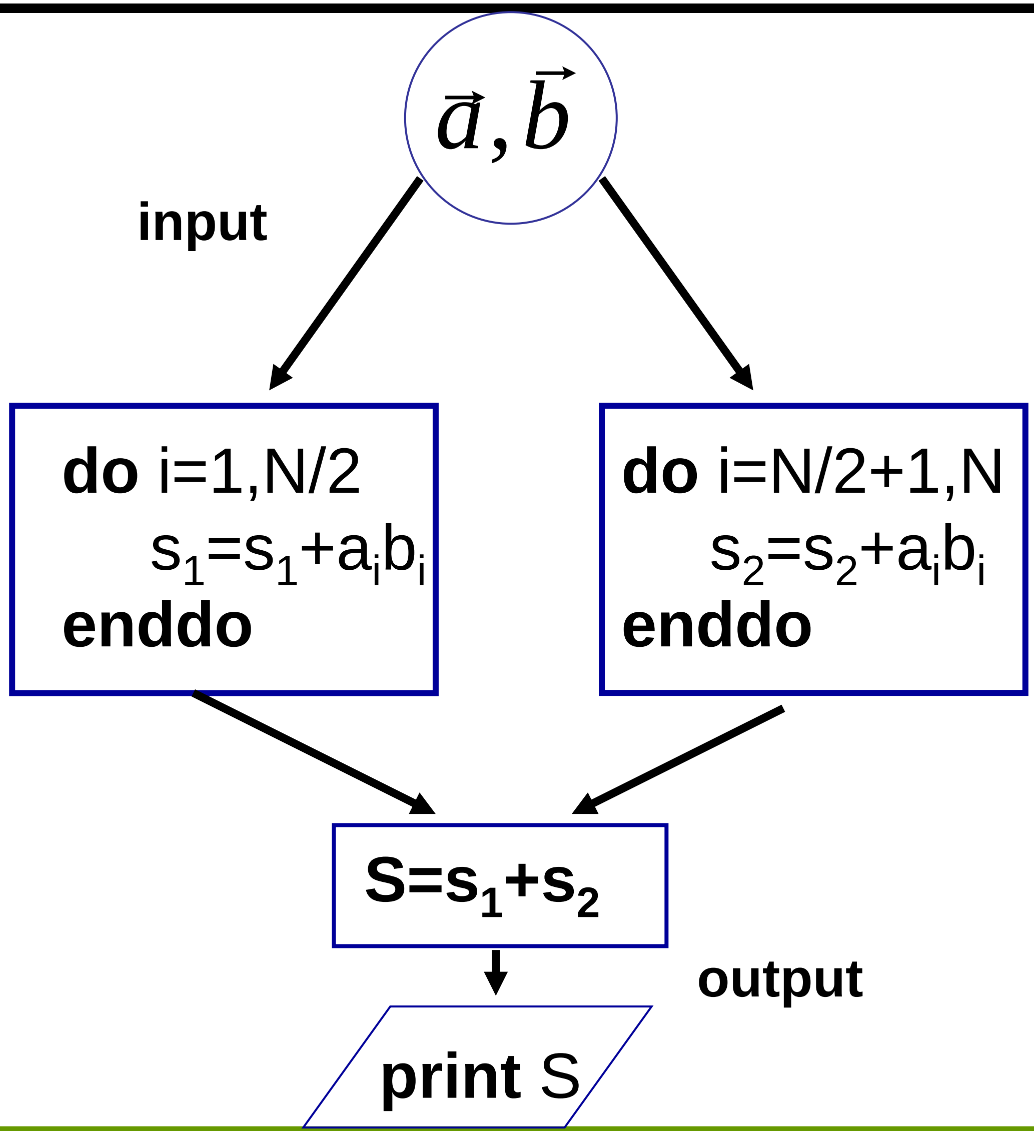
MPI 是英文 *Message Passing Interface* 的缩写，是基于消息传递编写并行程序的一种用户界面. 消息传递是目前并行计算机上广泛使用的一种程序设计模式，特别是对分布式存储的可扩展的并行计算机 *SPCs (Scalable Parallel Computers)* 和工作站机群 *NOWs (Networks of Workstations)* 或 *COWs (Clusters of Workstations)*. 尽管还有很多其它的程序实现方式，但是过程之间的通信采用消息传递已经是一种共识. 在 *MPI* 和 *PVM* 问世以前，并行程序设计与并行计算机系统是密切相关的，对不同的并行计算机就要编写不同的并行程序，给并行程序设计和应用带来了许多麻烦，广大并行计算机的用户迫切需要一些通用的消息传递用户界面，使并行程序具有和串行程序一样的可移植性.

HPComputing: How to get cooperated units? – Divide & Conquer



Example: scalar product of vectors





Sequential (serial) algorithm **Parallel algorithm**

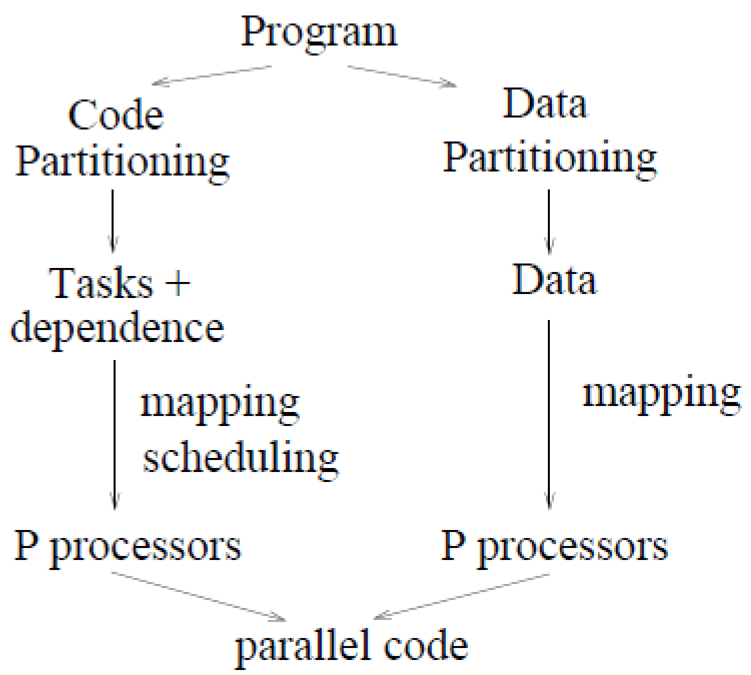
Transformation based parallel programming

Program parallelization techniques.

1. **Program Mapping**
   - Program Partitioning. Dependence Analysis.
   - Scheduling & Load balancing.
   - Code distribution.
2. **Data Mapping.**
   - Data partitioning.
   - Communication between processors.
   - Data distribution. Indexing of local data.

Program and data mapping should be **consistent**.

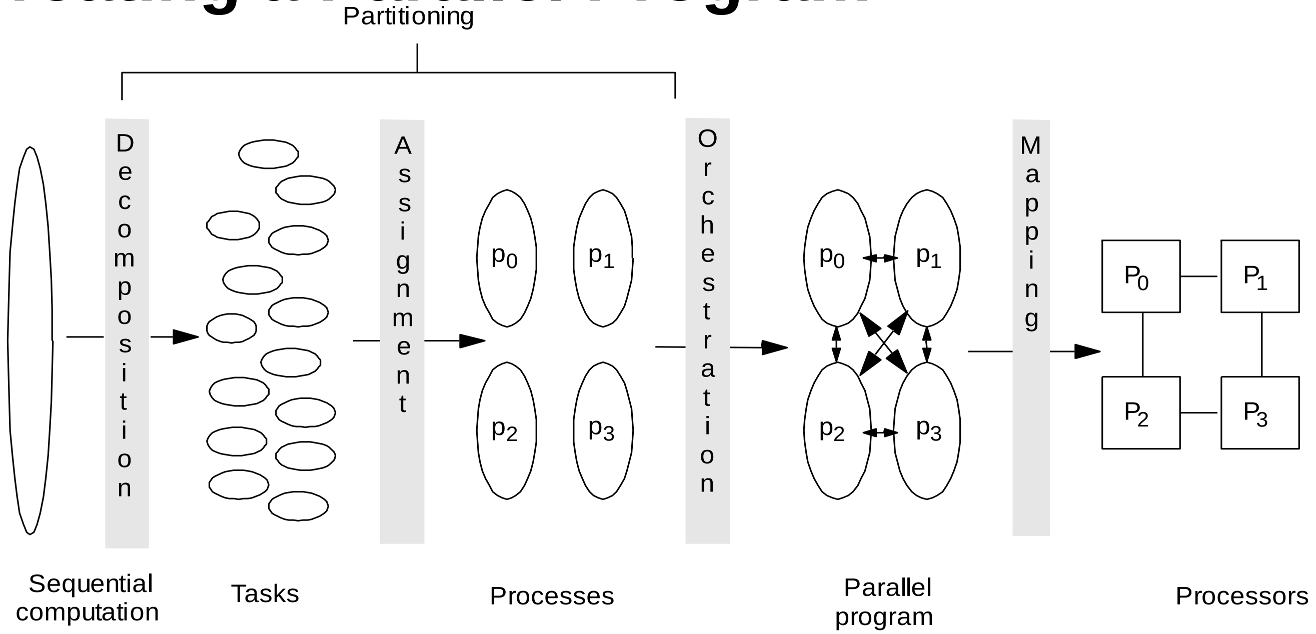
Program Parallelization



DAOM (1991)

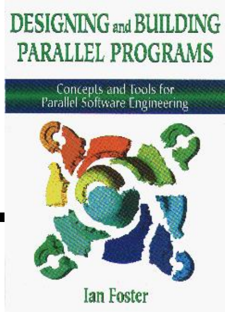
K. Mani Chandy, Stephen Taylor. An Introduction to Parallel Programming. Jones and Bartl
Burlington. 1991.

4 Steps in Creating a Parallel Program

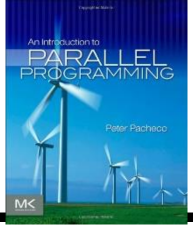


- **D**ecomposition of computation in tasks
- **A**ssignment of tasks to processes
- **O**rchestration of data access, comm, synch.
- **M**apping processes to processors

Foster's model – PCAM (1995)


DESIGNING and BUILDING PARALLEL PROGRAMS
Concepts and Tools for Parallel Software Engineering
Ian Foster

- **In "*Designing and Building Parallel Programs*" Ian Foster proposes a model with tasks that interact with each other by communicating through channels.**
  - A **task** is a program, its local memory, and its communication in-ports and out-ports.
  - A **channel** connects a task's in-port to another task's out-port.
  - Channels are buffered. Sending is **asynchronous** while receiving is **synchronous** (receiving task is blocked until expected message arrives).



However, Ian Foster provides an outline of steps in his online book *Designing and Building Parallel Programs* [19]:

1. *Partitioning*. Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
2. *Communication*. Determine what communication needs to be carried out among the tasks identified in the previous step.
3. *Agglomeration or aggregation*. Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. *Mapping*. Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

This is sometimes called **Foster's methodology**.

PCAM设计方法学

任务划分(**Partitioning**)

- 将整个计算分解为一些小任务，其目的是尽量开拓并行执行的机会

通信(**Communication**)

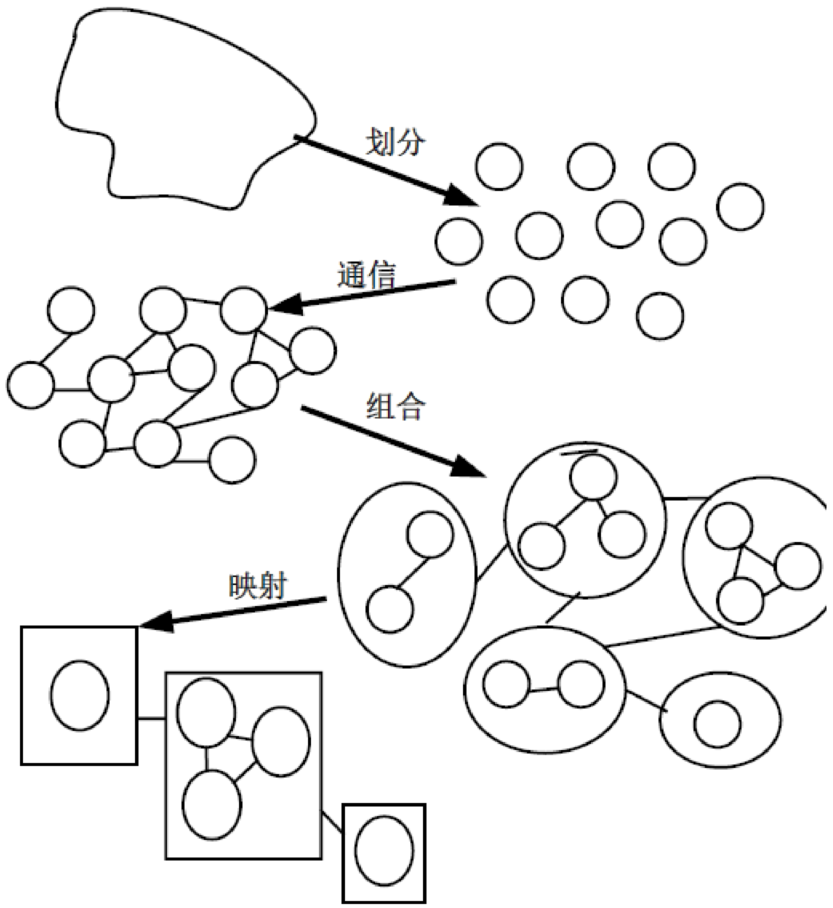
- 确定诸任务执行中所需要交换的数据和协调诸任务的执行，由此检测上述划分的合理性

任务组合(**Agglomeration**)

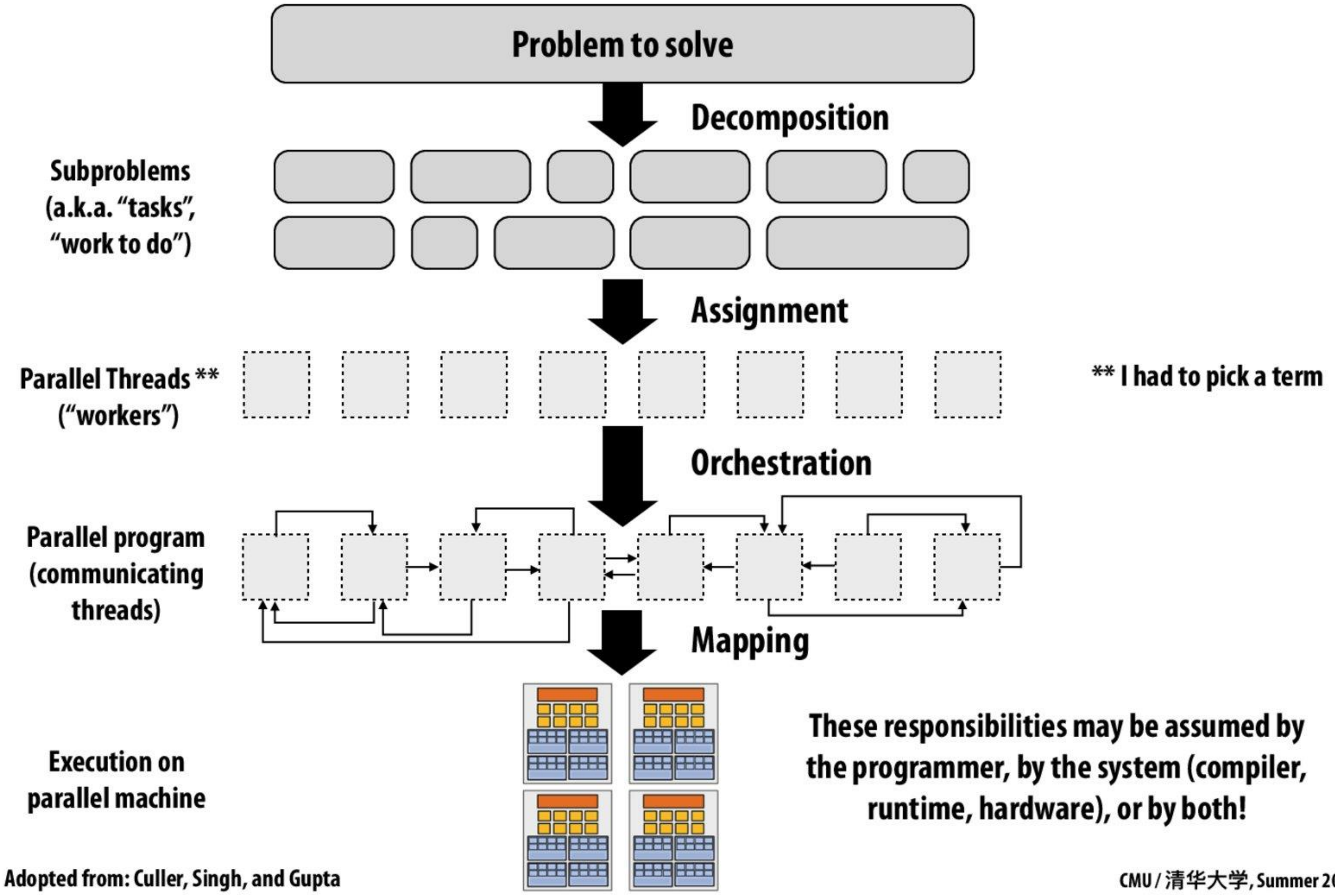
- 按性能要求和实现的代价来考察前两阶段的结果，必要时可将一些小任务组合成更大的任务以提高性能和减少通信开销

处理器映射(**Mapping**)

- 将每个任务分配到一个处理器上，其目的是最小化全局执行时间和通信成本以及最大化处理器的利用率



Creating a parallel program



Problem to solve
Decomposition
Subproblems (a.k.a. "tasks", "work to do")
Assignment
Parallel Threads ** ("workers")
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine

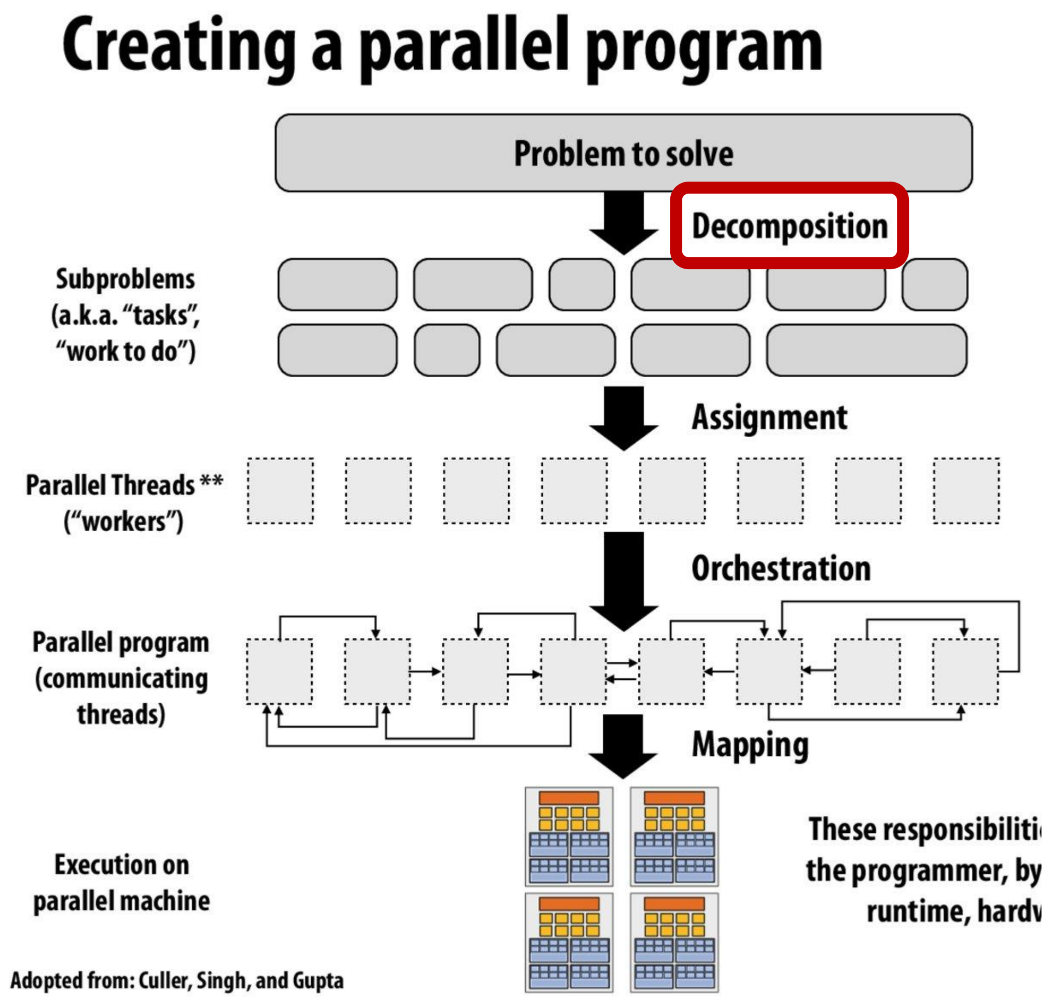
** I had to pick a term

These responsibilities may be assumed by the programmer, by the system (compiler, runtime, hardware), or by both!

Adopted from: Culler, Singh, and Gupta

◻ 划分 (Partition/Decompose)

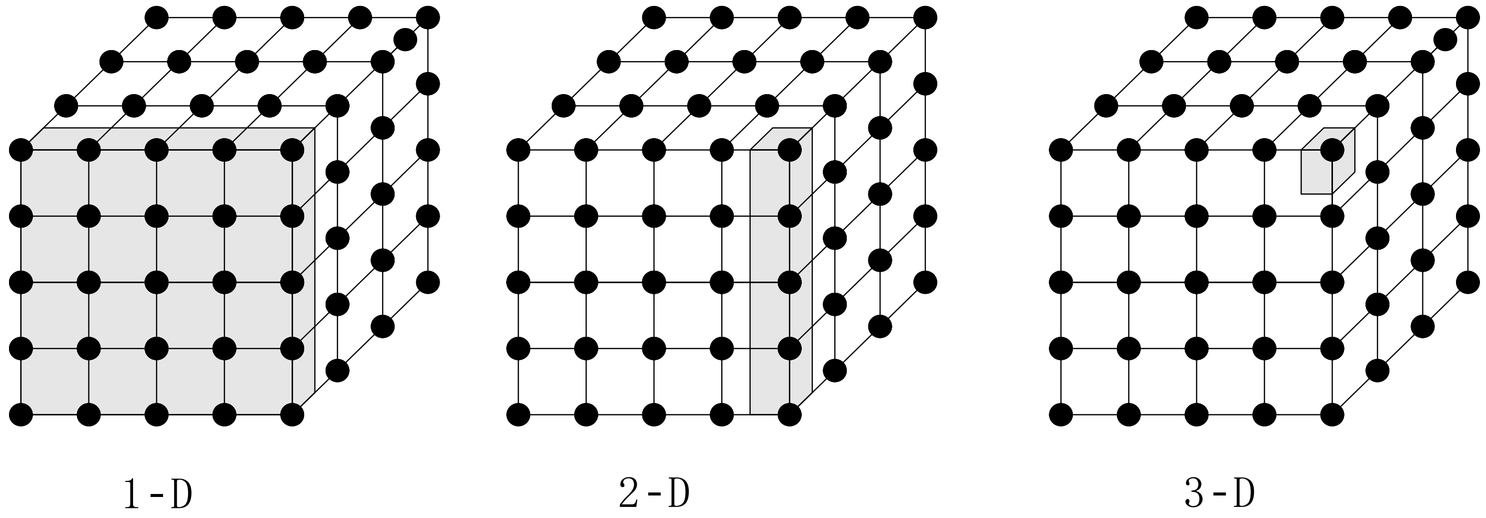
- 充分开拓算法的并发性和可扩放性；
- 先进行数据分解(称域分解)，再进行计算功能的分解(称功能分解)；
- 使数据集和计算集互不相交；
- 划分阶段忽略处理器数目和目标机器的体系结构；
- 能分为两类划分：
  - ➢域分解(Domain Decomposition)
  - ➢功能分解(Functional Decomposition)


Creating a parallel program
Problem to solve
Decomposition
Subproblems (a.k.a. "tasks", "work to do")
Assignment
Parallel Threads ** ("workers")
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine
These responsibiliti
the programmer, by
runtime, hardv
Adopted from: Culler, Singh, and Gupta

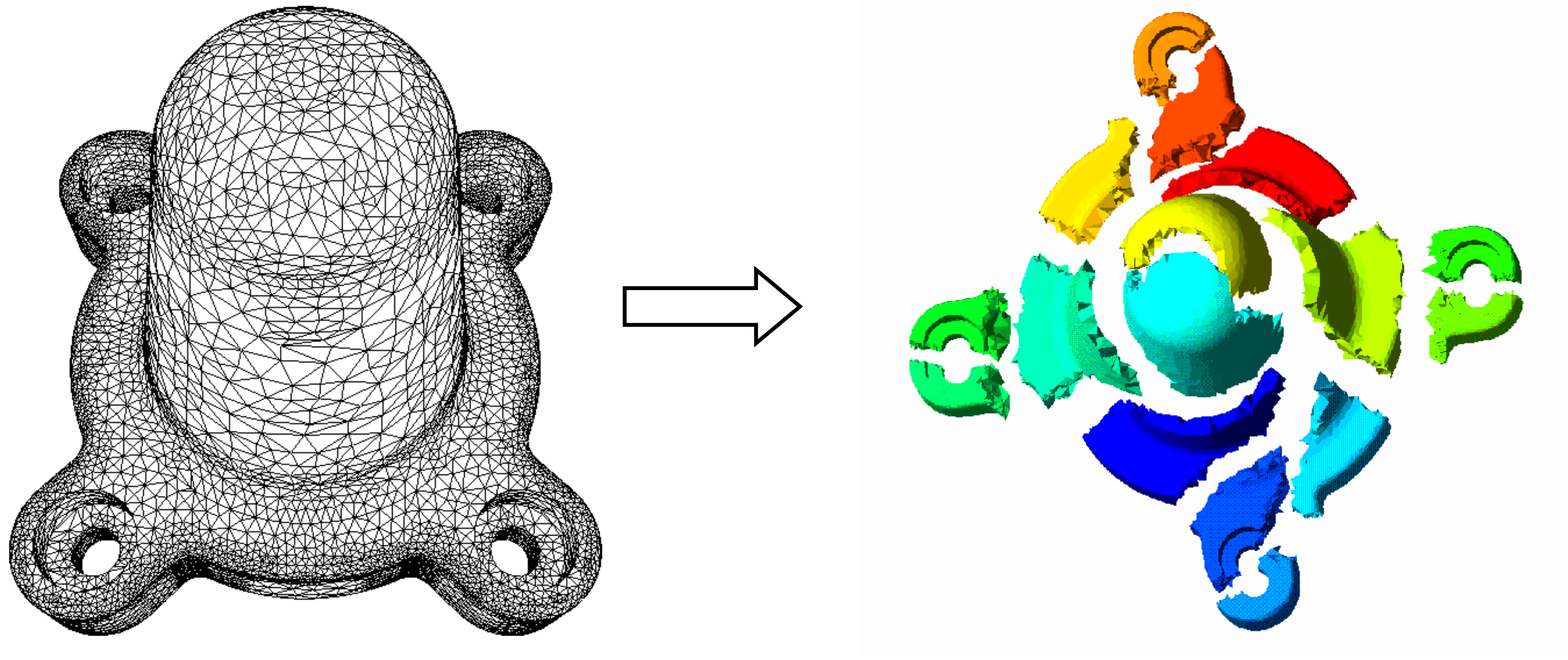
域分解

- 划分的对象是数据，可以是程序中的输入数据、中间处理数据和输出数据；
- 将数据分解成大致相等的小数据片；
- 划分时考虑数据上的相应操作；
- 如果一个任务需要别的任务中的数据，则会产生任务间的通信；

示例：三维网格的域分解，各格点上计算都是重复的。下图是三种分解方法



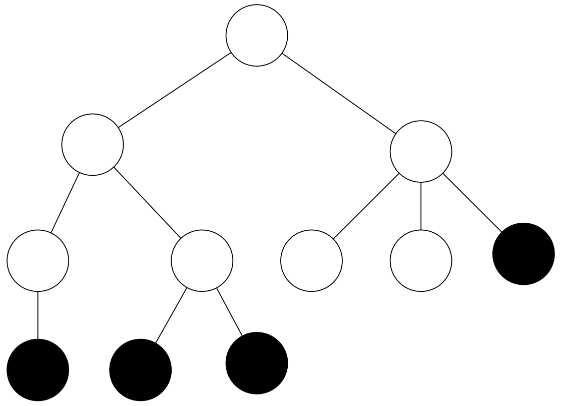
不规则区域的分解示例



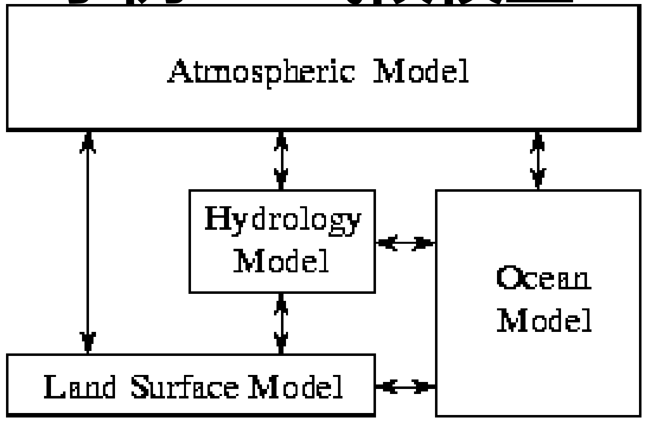
功能分解

- 划分的对象是计算（亦称为任务分解或计算划分），将计算划分为不同的任务，其出发点不同于域分解；
- 划分后，研究不同任务所需的数据。如果这些数据不相交的，则划分是成功的；如果数据有相当的重叠， 意味着存在大量的通信开销，要重新进行域分解和功能分解；
- 功能分解是一种更深层次的分解

示例1：搜索树



示例2：气候模型

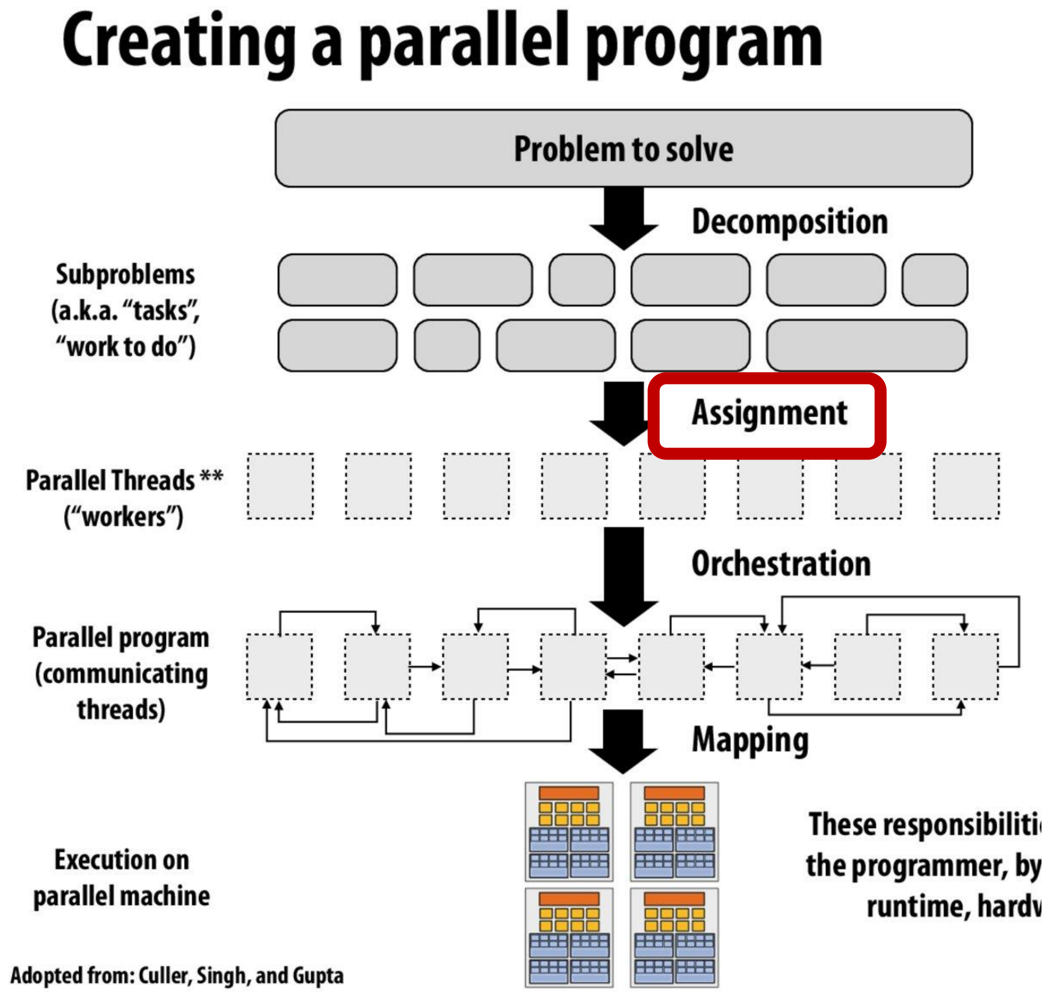


通信

- 通信是PCAM设计过程的重要阶段；
- 划分产生的诸任务，一般不能完全独立执行，需要在任务间进行数据交流；从而产生了通信；
- 功能分解确定了诸任务之间的数据流；
- 诸任务是并发执行的，通信则限制了这种并发性

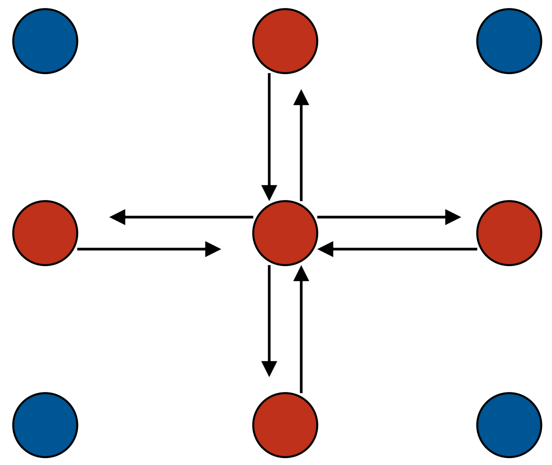
四种通信模式

- 局部/全局通信
- 结构化/非结构化通信
- 静态/动态通信
- 同步/异步通信


Creating a parallel program
Problem to solve
Decomposition
Subproblems (a.k.a. "tasks", "work to do")
Assignment
Parallel Threads ** ("workers")
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine
These responsibiliti
the programmer, by
runtime, hardv
Adopted from: Culler, Singh, and Gupta

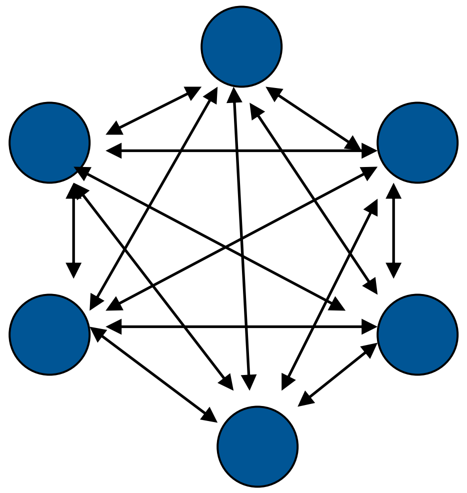
局部通信

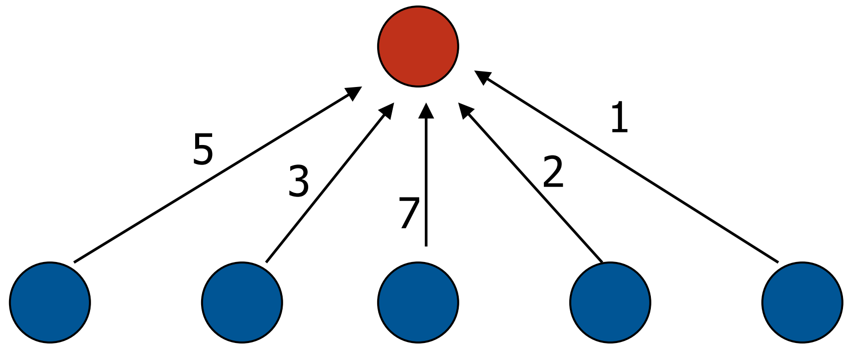
- 通信限制在一个邻域内



全局通信

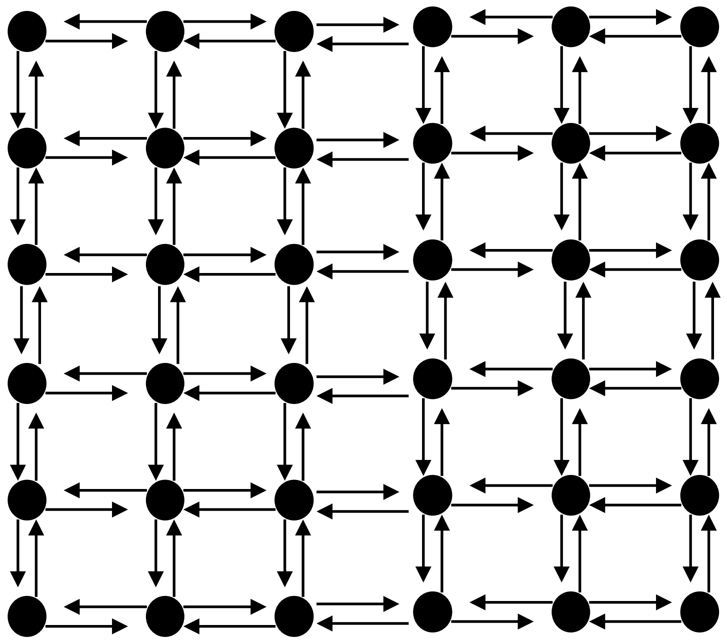
- 例如：
  - - All to All
  - - Master-Worker

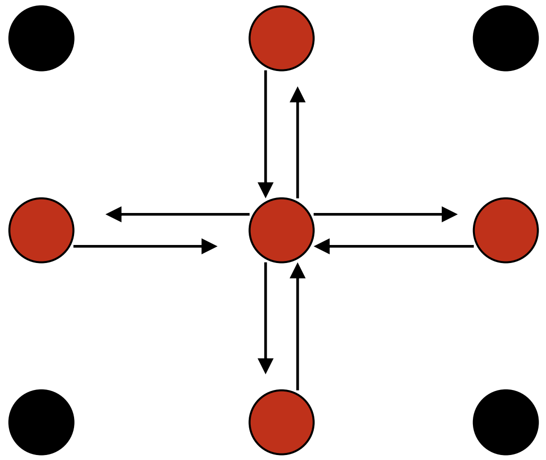




结构化通信

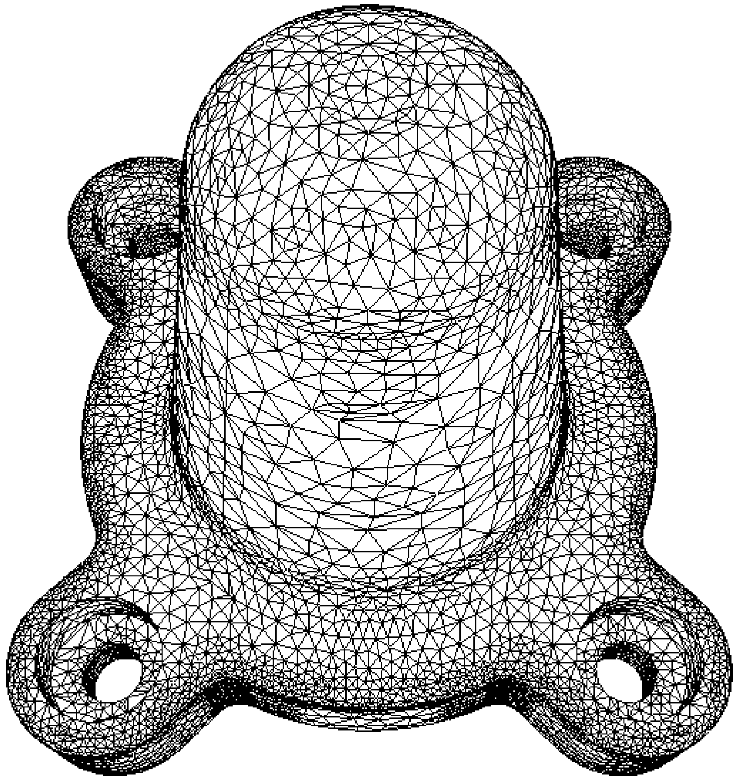
- 每个任务的通信模式是相同的；
- 下面是否存在一个相同通信模式？





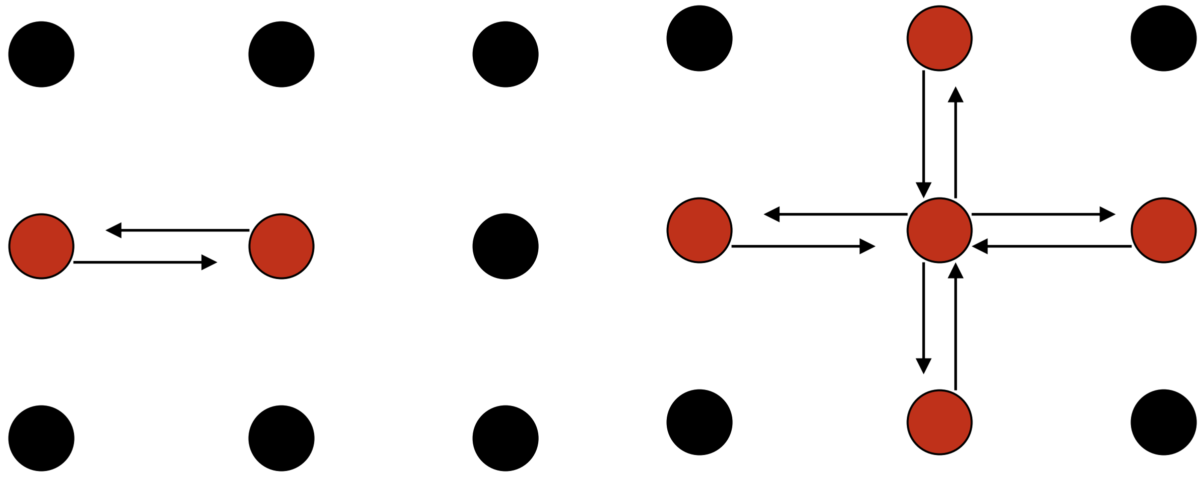
非结构化通信

- 没有一个统一的通信模式
  例如：无结构化网格



静态/动态通信

通信伙伴的身份不随时间改变；动态通信中，通信伙伴的身份则可能由运行时所计算的数据决定且是可变的



同步/异步通信

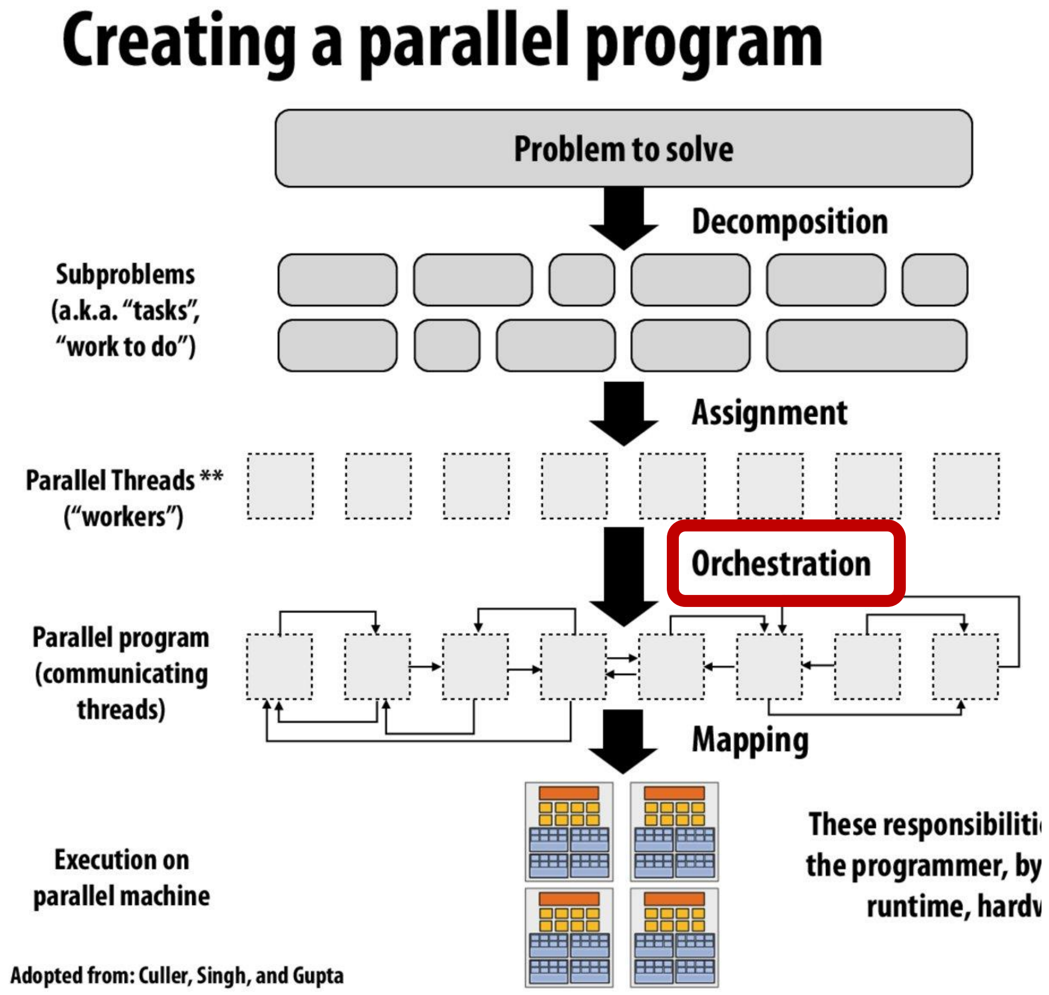
- 同步通信时，接收方和发送方协同操作；异步通信中，接收方获取数据无需与发送方协同

通信判据

- 所有任务是否执行大致相当的通信?
- 是否尽可能的局部通信?
- 通信操作是否能并行执行?
- 同步任务的计算能否并行执行?

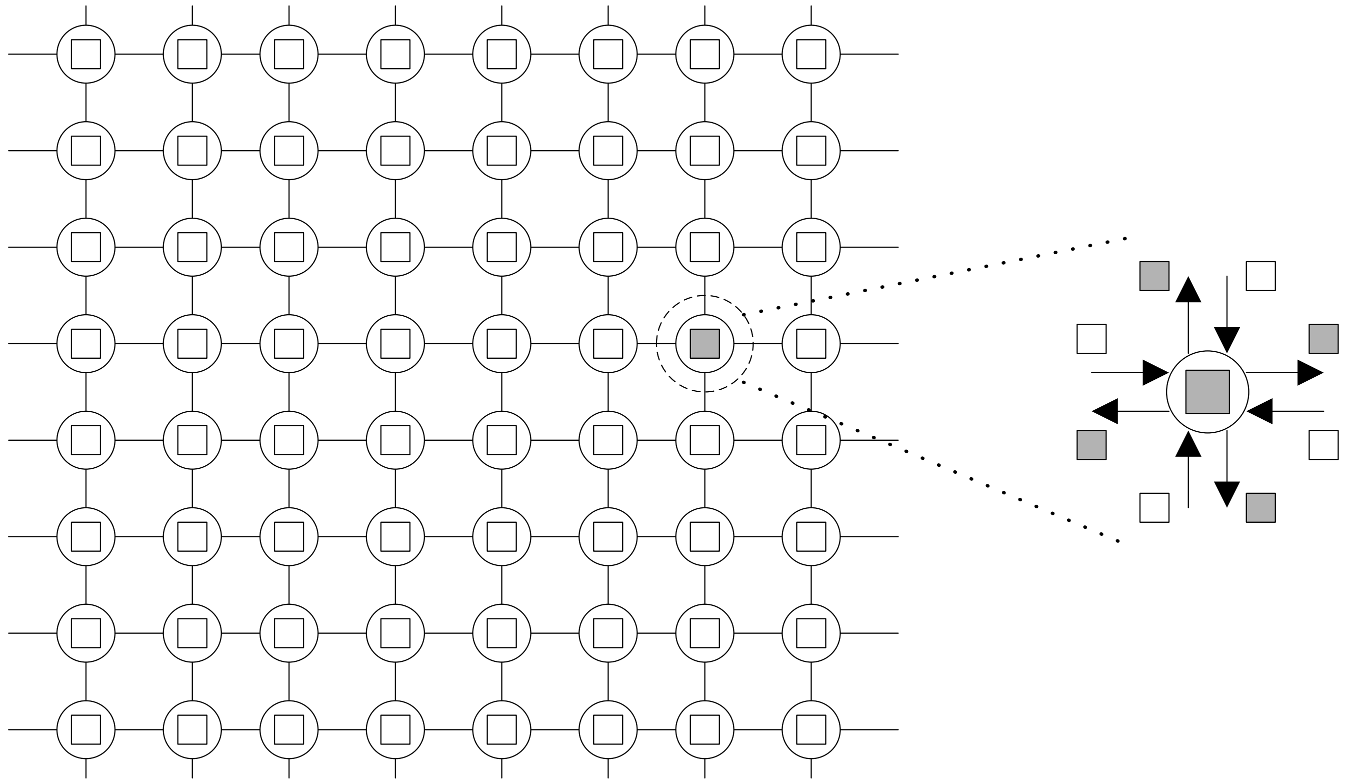
任务组合

- 组合是由抽象到具体的过程，是将组合的任务能在一类并行机上有效的执行；
- 合并小尺寸任务，减少任务数。如果任务数恰好等于处理器数，则也完成了映射过程；
- 通过增加任务的粒度和重复计算，可以减少通信成本；
- 保持映射和扩展的灵活性，降低软件工程成本


Creating a parallel program
Problem to solve
Decomposition
Subproblems (a.k.a. "tasks", "work to do")
Assignment
Parallel Threads ** ("workers")
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine
These responsibiliti
the programmer, by
runtime, hardv
Adopted from: Culler, Singh, and Gupta

表面-容积效应

- 通信量与任务子集的表面成正比，计算量与任务子集的体积成正比；
- 增加重复计算有可能减少通信量

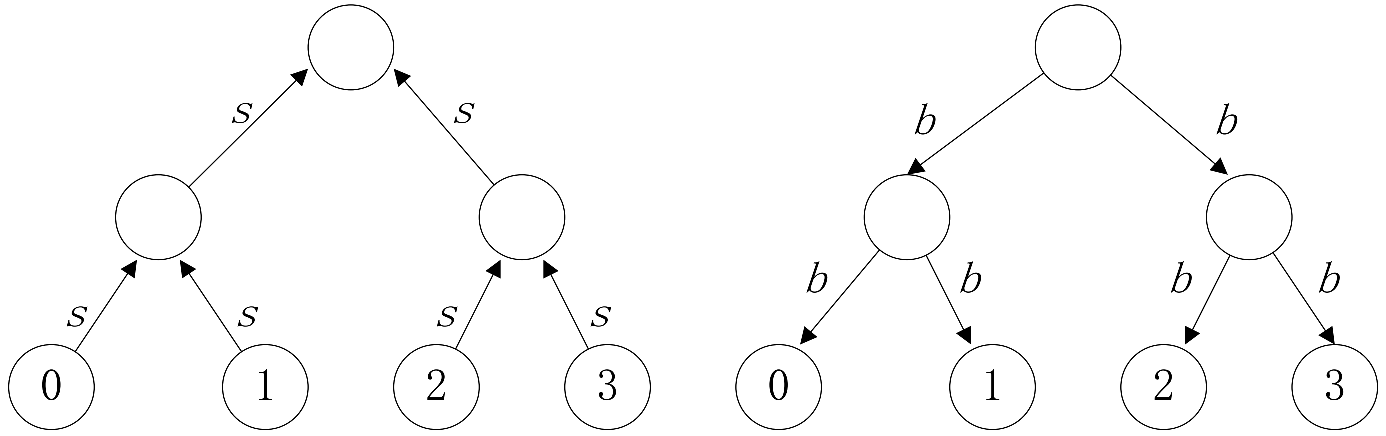


重复计算

- 重复计算减少通信量，但增加了计算量，应保持恰当的平衡；
- 重复计算的目标应减少算法的总运算时间

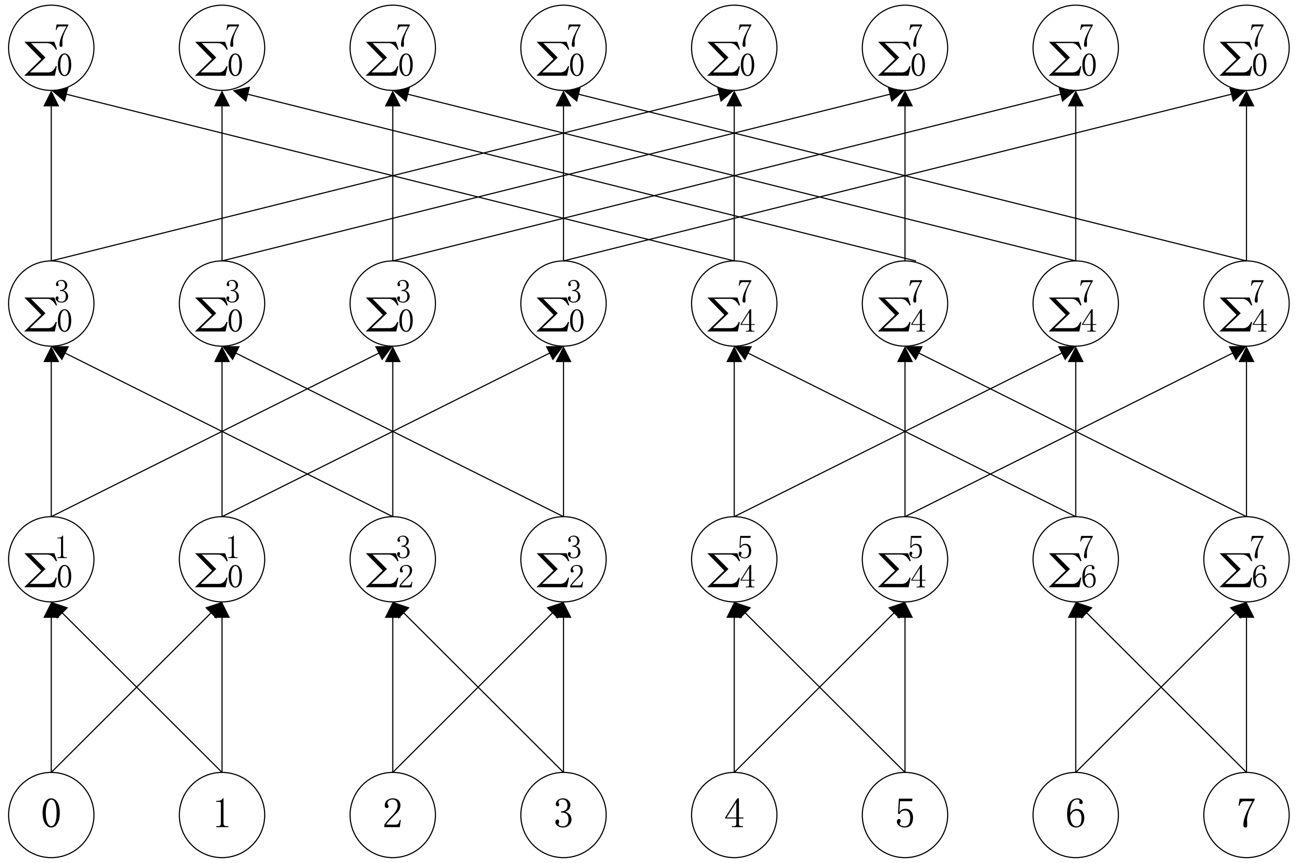
示例：二叉树上N个处理器求N个数的全和，要求每个处理器均保持全和

- 二叉树上求和，共需2logN步



示例：二叉树上N个处理器求N个数的全和，要求每个处理器均保持全和

- 蝶式结构求和，使用了重复计算，共需logN步

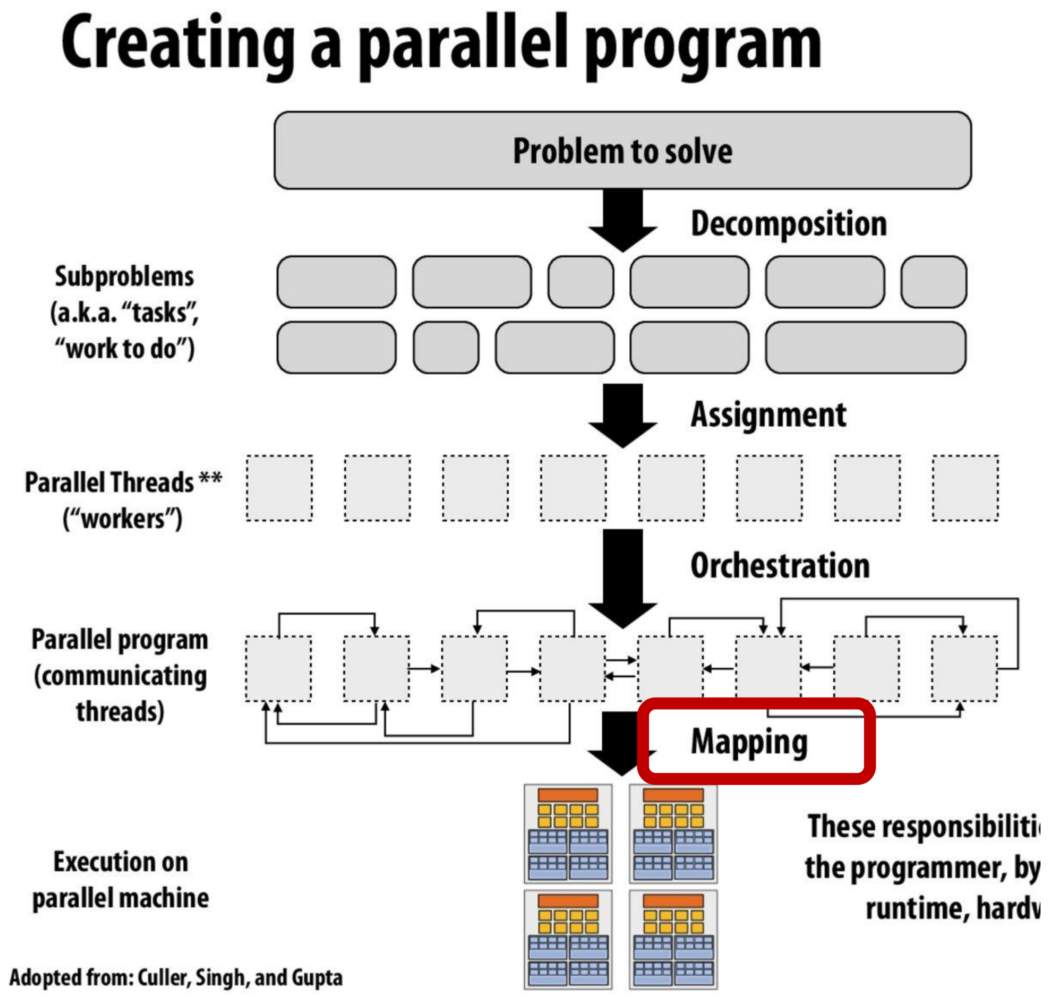


组合判据

- 增加粒度是否减少了通信成本?
- 重复计算是否已权衡了其得益?
- 是否保持了灵活性和可扩放性?
- 组合的任务数是否与问题尺寸成比例?
- 是否保持了类似的计算和通信?
- 有没有减少并行执行的机会?

处理器映射

- 每个任务要映射到具体的处理器，定位到运行机器上；
- 任务数大于处理器数时，存在负载平衡和任务调度问题；
- 映射的目标：减少算法的执行时间
  - 并发的任务　　不同的处理器
  - 任务之间存在高通信的　　同一处理器
- 映射实际是一种权衡，属于**NP完全问题**


Creating a parallel program
Problem to solve
Decomposition
Subproblems (a.k.a. "tasks", "work to do")
Assignment
Parallel Threads ** ("workers")
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine
These responsibiliti
the programmer, by
runtime, hardv
Adopted from: Culler, Singh, and Gupta

负载平衡

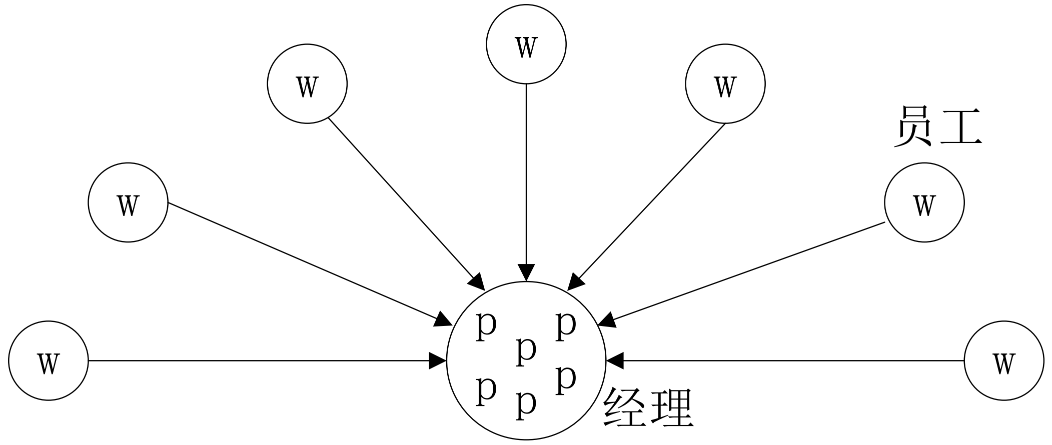
- 静态的：事先确定；
- 概率的：随机确定；
- 动态的：执行期间动态负载；
- 基于域分解的：
  - 递归对剖
  - 局部算法
  - 概率方法
  - 循环映射

任务分配与调度

- 负载平衡与任务分配/调度密切相关，任务分配通常有静态的和动态的两种方法。
- 静态分配一般是任务到进程的算术映射。
  - 静态分配的优点是没有运行时任务管理的开销，但为了实现负载平衡，要求不同任务的工作量和处理器的性能是可以预测的并且拥有足够的可供分配的任务。
  - 静态调度（Static Scheduling）方案一般是静态地为每个处理器分配个连续的循环迭代，其中为迭代次数，是处理器数。也可以采用轮转（Round-robin）的方式来给处理器分配任务，即将第i个循环迭代分配给第i mod p个处理器
- 动态分配与调度相对灵活，可以运行时在不同处理器间动态地进行负载的调整
  - 各种动态调度(Dynamic Scheduling)技术是并行计算研究的热点，包括基本自调度SS(Self Scheduling)、块自调度BSS(Block Self Scheduling)、 指导自调度GSS(Guided Self Scheduling)、因子分解调度FS(Factoring Scheduling)、梯形自调度TSS(Trapezoid Self Scheduling)、 耦合调度AS(Affinity Scheduling)、安全自调度SSS(Safe Self Scheduling)和自适应耦合调度AAS(Adapt Affinity Scheduling)

经理/雇员模式任务调度

- 任务放在集中的或分散的任务池中，使用任务调度算法将池中的任务分配给特定的处理器

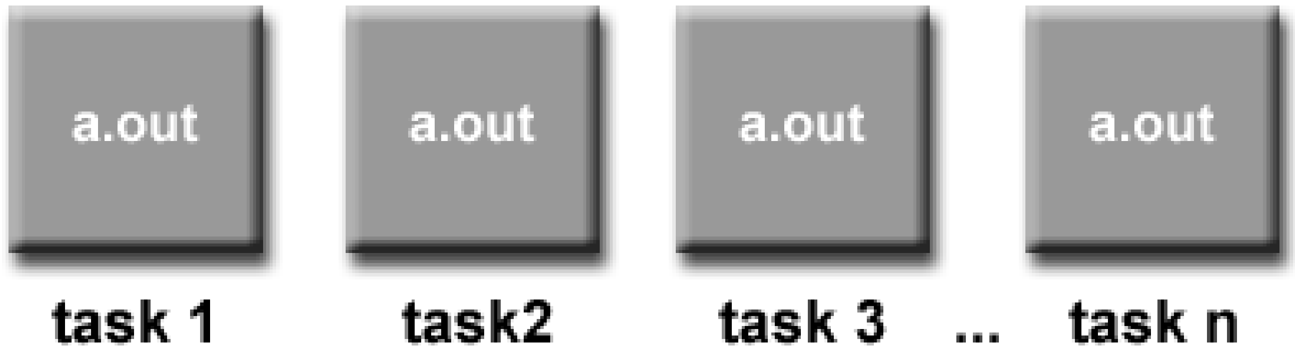


映射判据

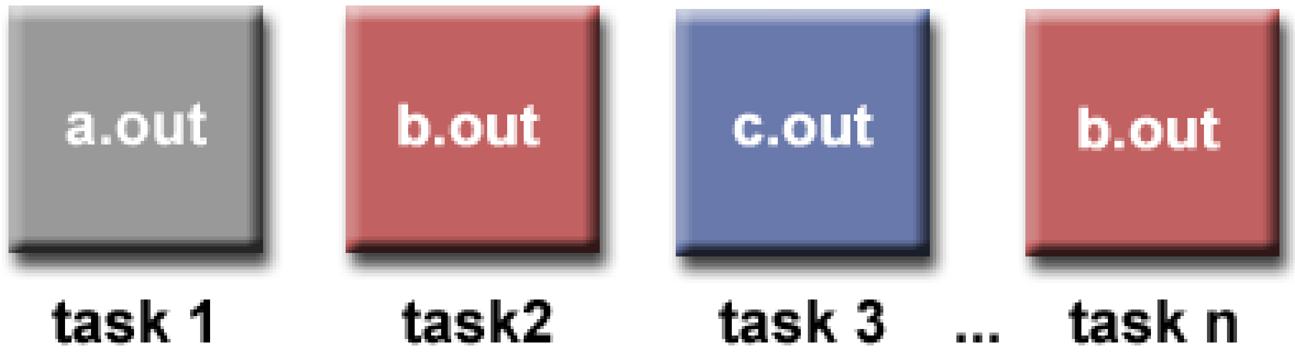
- 采用集中式负载平衡方案，是否存在通信瓶颈?
- 采用动态负载平衡方案，调度策略的成本如何?

SPMD or MPMD

- Single Program, Multiple Data

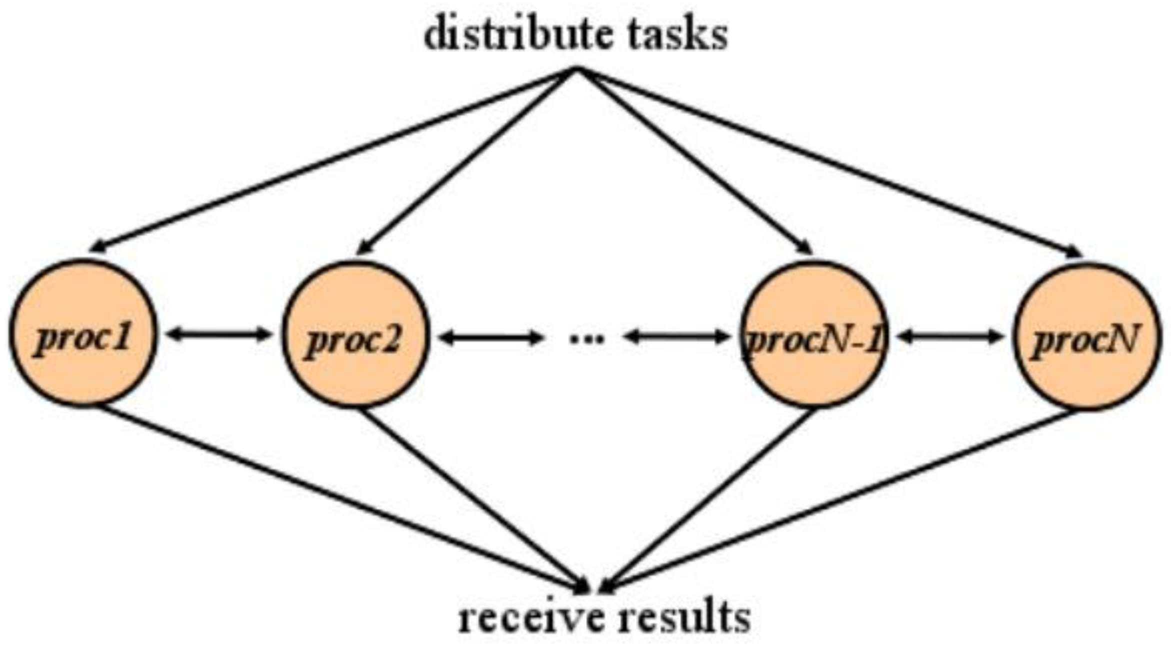


- Multiple Program Multiple Data

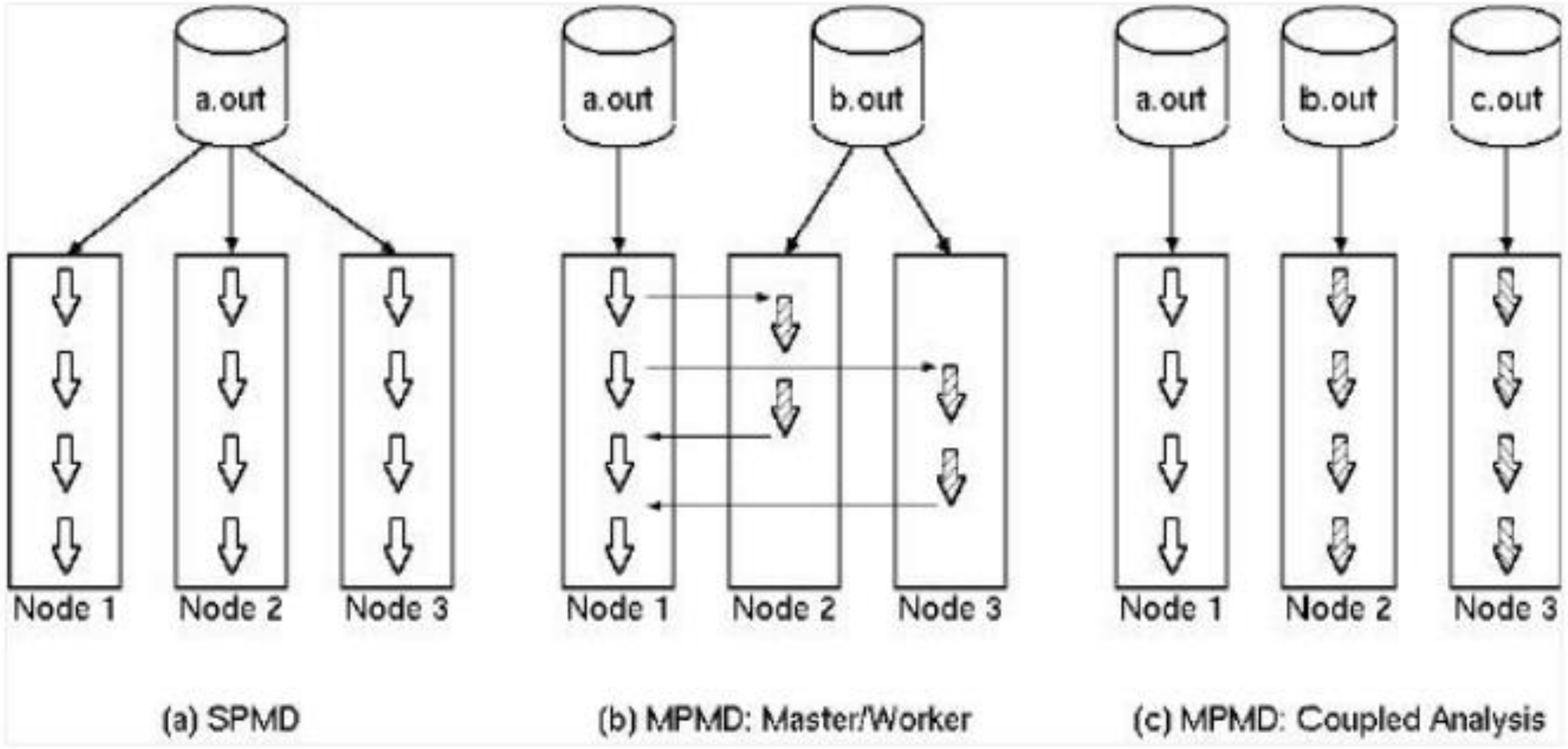


Single Program Multiple Data (SPMD)

- All processes execute the same program, but on different parts of data.
  - also known as data parallelism
  - similar to Master/Worker, but here we might have communication between tasks.

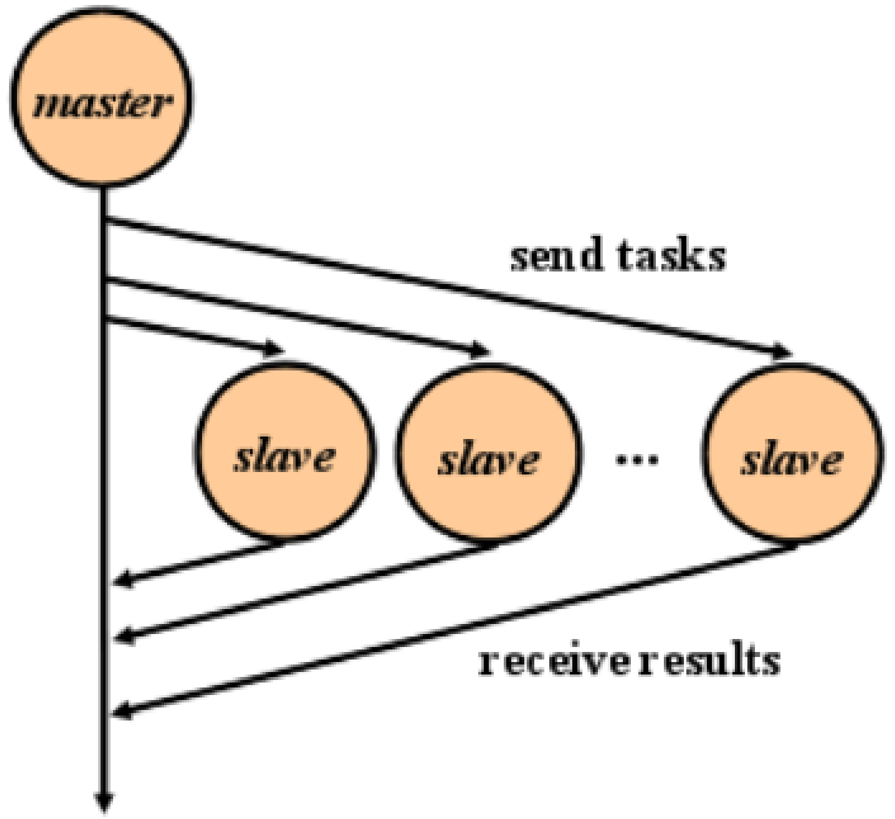


SPMD (Single Program Multiple Data) *(a)*
MPMD (Multiple Program Multiple Data) (*b*,*c*)



Master/Slave or Master/Worker

- A master process is responsible for
  - initiating the computation
  - possibly determine the tasks
  - distribute tasks to worker processors
  - aggregate partial results from workers, and produce final result
- slaves/workers execute a simpler execution cycle
  - receive task
  - compute task
  - send task-result to master



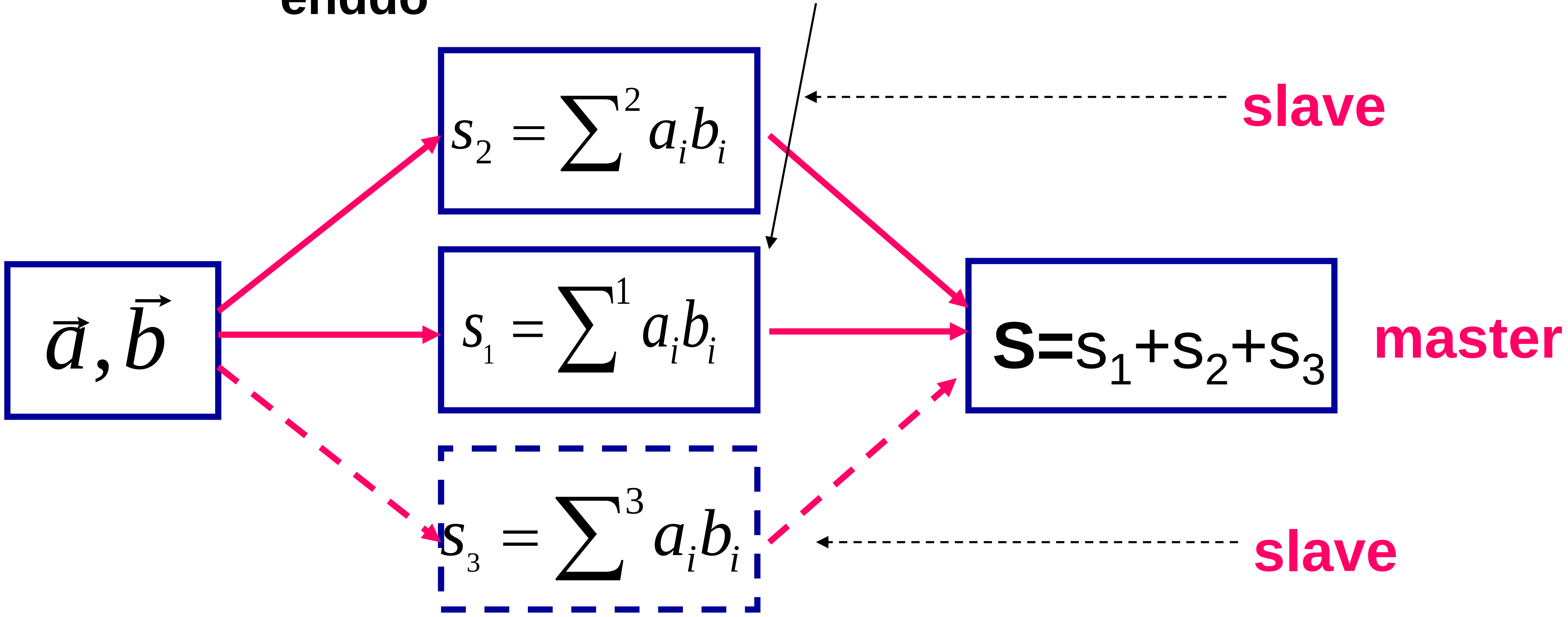
What is Master/Slave principle?

- The master has the control over the running application, it controls all data and it calls the slaves to do there work

```
PROGRAM
    IF (process = master) THEN
            master-code
    ELSE
            slave-code
    ENDIF
END
```

Simple Example SPMD&Master/Slave

```
For i from rank step size to N do
        $s=s+a_i b_i$
enddo
```



Decomposition/Partitioning

- Identify concurrency and decide level at which to exploit it
  - Break up computation into tasks to be divided among processes
- Tasks may become available dynamically
  - No. of available tasks may vary with time
- Goal: Enough tasks to keep processes busy, but not too many
  - Number of tasks available at a time is upper bound on achievable speedup

One simple example – Data Parallelism

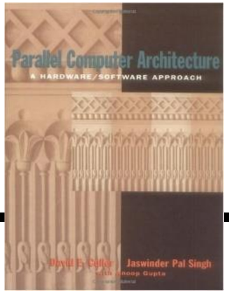
- A parallelizing compiler must id
  dependences BETWEEN ITE

Of course, real problems are more complex!!

Example:

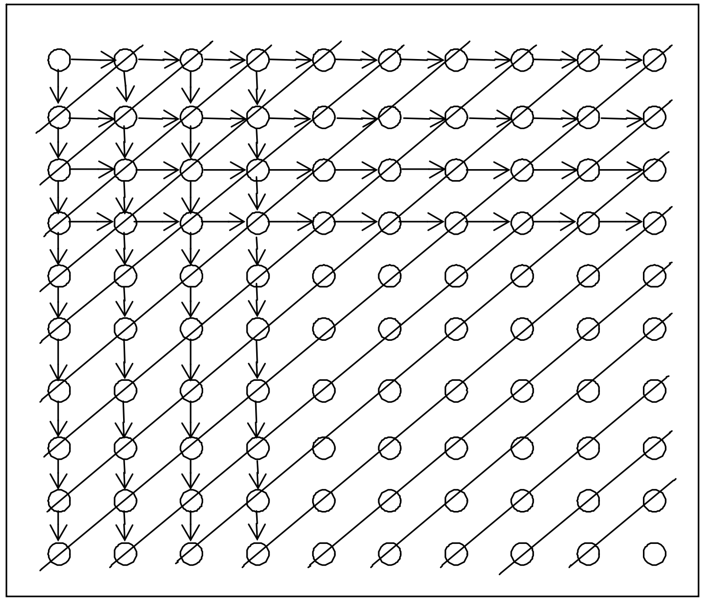
```
do I = 1, 1000
   A(I) = B(I) + C(I)
   D(I) = A(I)
end do
```

```
Fork one thread for each processor
  Each thread executes the loop:
    do I = my_lo, my_hi
      A(I) = B(I) + C(I)
      D(I) = A(I)
    end do
Wait for all threads to finish before proceeding.
```

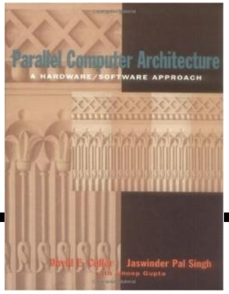


Decomposition/Partitioning

- Simple way to identify concurrency is to look at loop iterations
  - *dependence analysis*; if not enough concurrency, then look further
- Not much concurrency here at this level (all loops *sequential*)
- Examine fundamental dependences

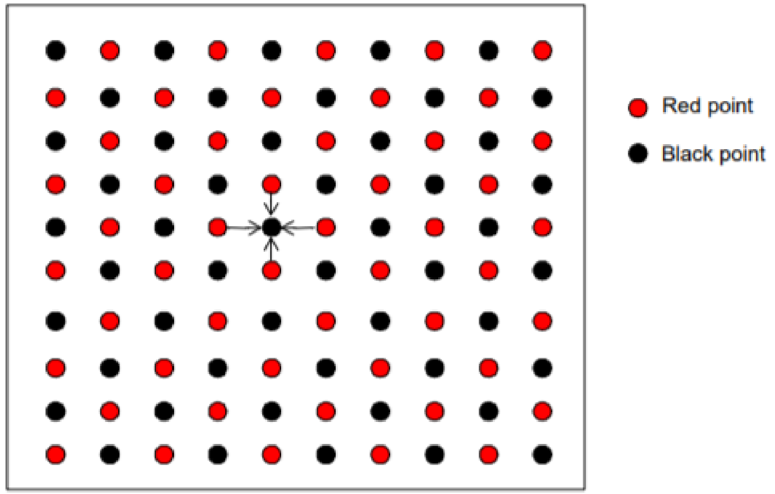


- Concurrency *O(n)* along anti-diagonals, serialization *O(n)* along diag.
- Retain loop structure, use pt-to-pt synch; Problem: too many synch ops.
- Restructure loops, use global synch; imbalance and too much synch



Exploit Application Knowledge

- Reorder grid traversal: red-black ordering



- Different ordering of updates: may converge quicker or slower
- Red sweep and black sweep are each fully parallel:
- Global synch between them (conservative but convenient)
- Ocean uses red-black
- We use simpler, asynchronous one to illustrate
    - no red-black, simply ignore dependences within sweep
    - parallel program *nondeterministic*

Assignment/Agglomeration

- Specify mechanism to divide work up among processes
  - E.g. which process computes forces on which stars, or which rays
  - Balance workload, reduce communication and management cost
- Structured approaches usually work well
  - Code inspection (parallel loops) or understanding of application
  - Well-known heuristics
  - *Static* versus *dynamic* assignment
- As programmers, we worry about partitioning first
  - *Usually* independent of architecture or prog model
  - But cost and complexity of using primitives may affect decisions

Orchestration

- Naming data
- Structuring communication
- Synchronization
- Organizing data structures and scheduling tasks temporally

- Goals
  - Reduce cost of communication and synch.
  - Preserve locality of data reference
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management

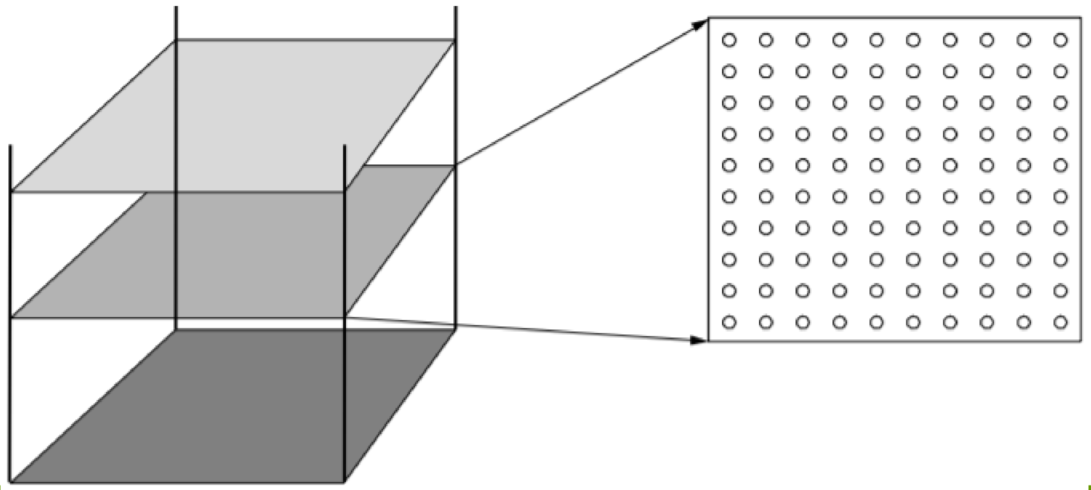
- Choices depend on Prog. Model., comm. abstraction, efficiency of primitives
- Architects should provide appropriate primitives efficiently

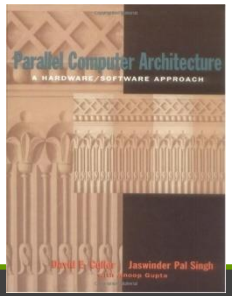
Mapping

- Two aspects:
  - Which process runs on which particular processor?
    - ✓ mapping to a network topology
  - Will multiple processes run on same processor?
- *space-sharing*
  - Machine divided into subsets, only one app at a time in a subset
  - Processes can be pinned to processors, or left to OS
- System allocation
- Real world
  - User specifies desires in some aspects, system handles some
- Usually adopt the view: process ↔ processor

Example: iterative equation solver

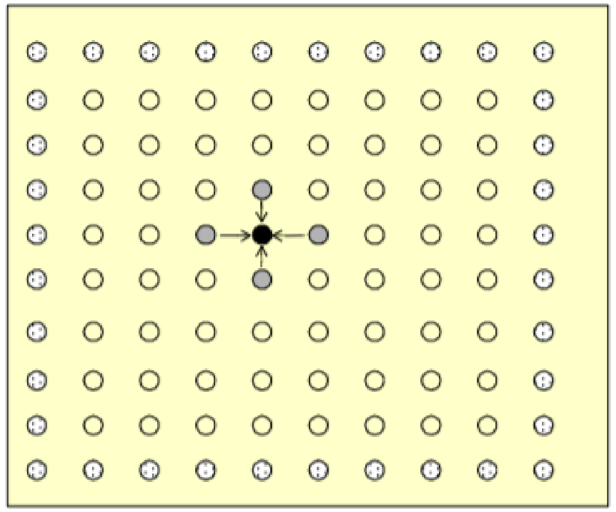
- **Simplified version of a piece of Ocean simulation**
  - Goal:
    - Simulate the motion of water currents in the ocean. Important to climate modeling. Motion depends on atmospheric forces, friction with ocean floor, and “friction” with ocean walls.
    - To predict the state of the ocean at any instant, we need to solve complex systems of equations.


(a) Cross sections



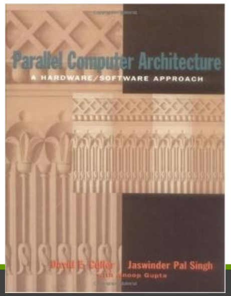
Example: iterative equation solver

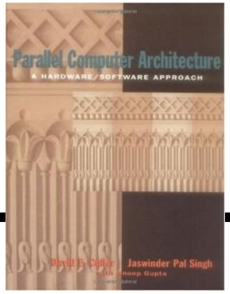
- **Simplified version of a piece of Ocean simulation**
- **Illustrate program in low-level parallel language**
  - C-like pseudocode with simple extensions for parallelism
  - Expose basic comm. and synch. primitives
  - State of most real parallel programming today



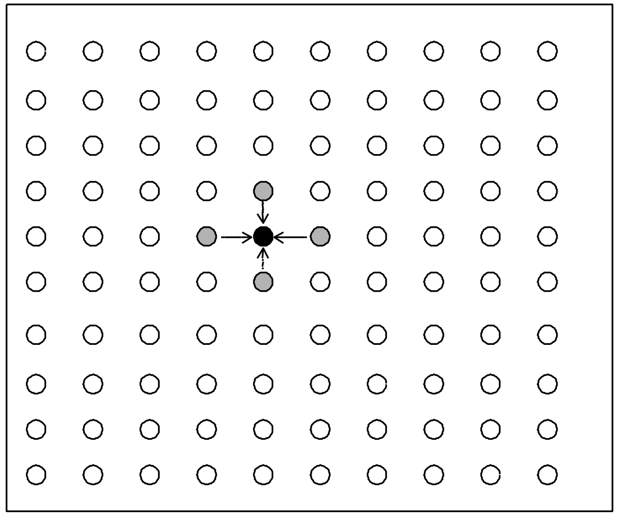
Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$





Grid Solver

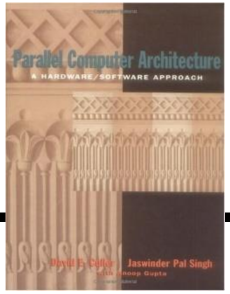


Expression for updating each interior point:

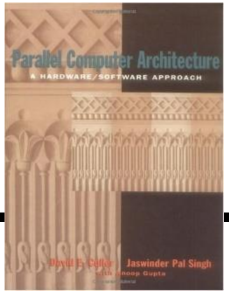
A[i,j] = 0.2 × (A[i,j] + A[i,j ? 1] + A[i ? 1, j] +
A[i,j + 1] + A[i + 1, j])

- <span>Gauss-Seidel (near-neighbor)</span> sweeps to convergence
  - interior n-by-n points of (n+2)-by-(n+2) updated in each sweep
  - updates done in-place in grid
  - difference from previous value computed
  - accumulate partial diffs into global diff at end of every sweep
  - check if has converged
    - ✓ to within a tolerance parameter

Sequential Version

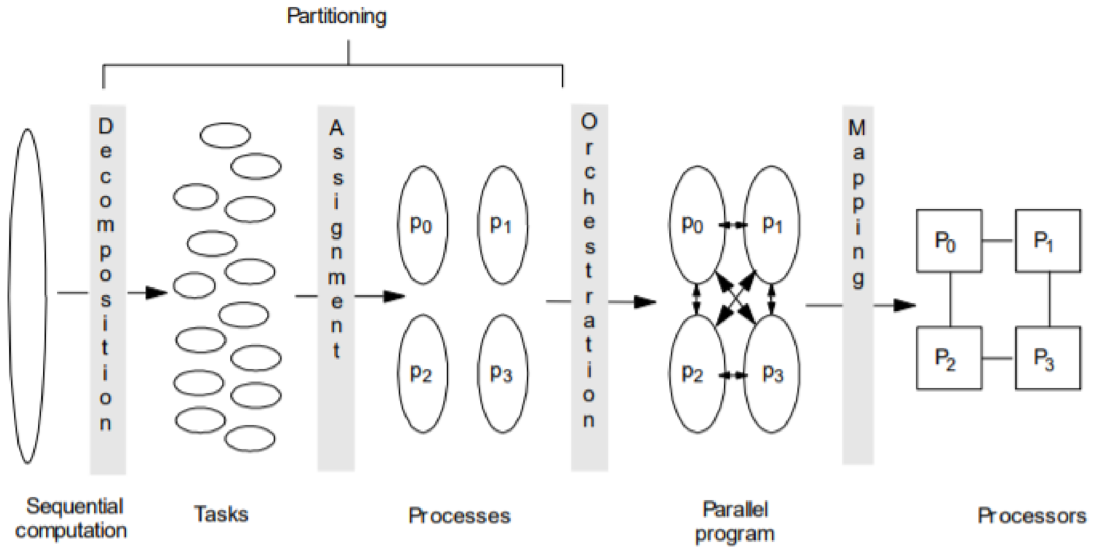


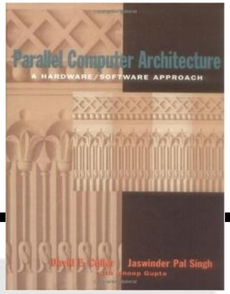
```
int n;                                          /*size of matrix: (n + 2-by-n + 2) elements*/
float **A, diff = 0;

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

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] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
               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
```



Four steps in parallelizing a program:

- **D**ecomposition of the computation into tasks.
- **A**ssignment of tasks to threads.
- **O**rchestration of the necessary data access, communication, and synchronization among threads.
- **M**apping of threads to processors.





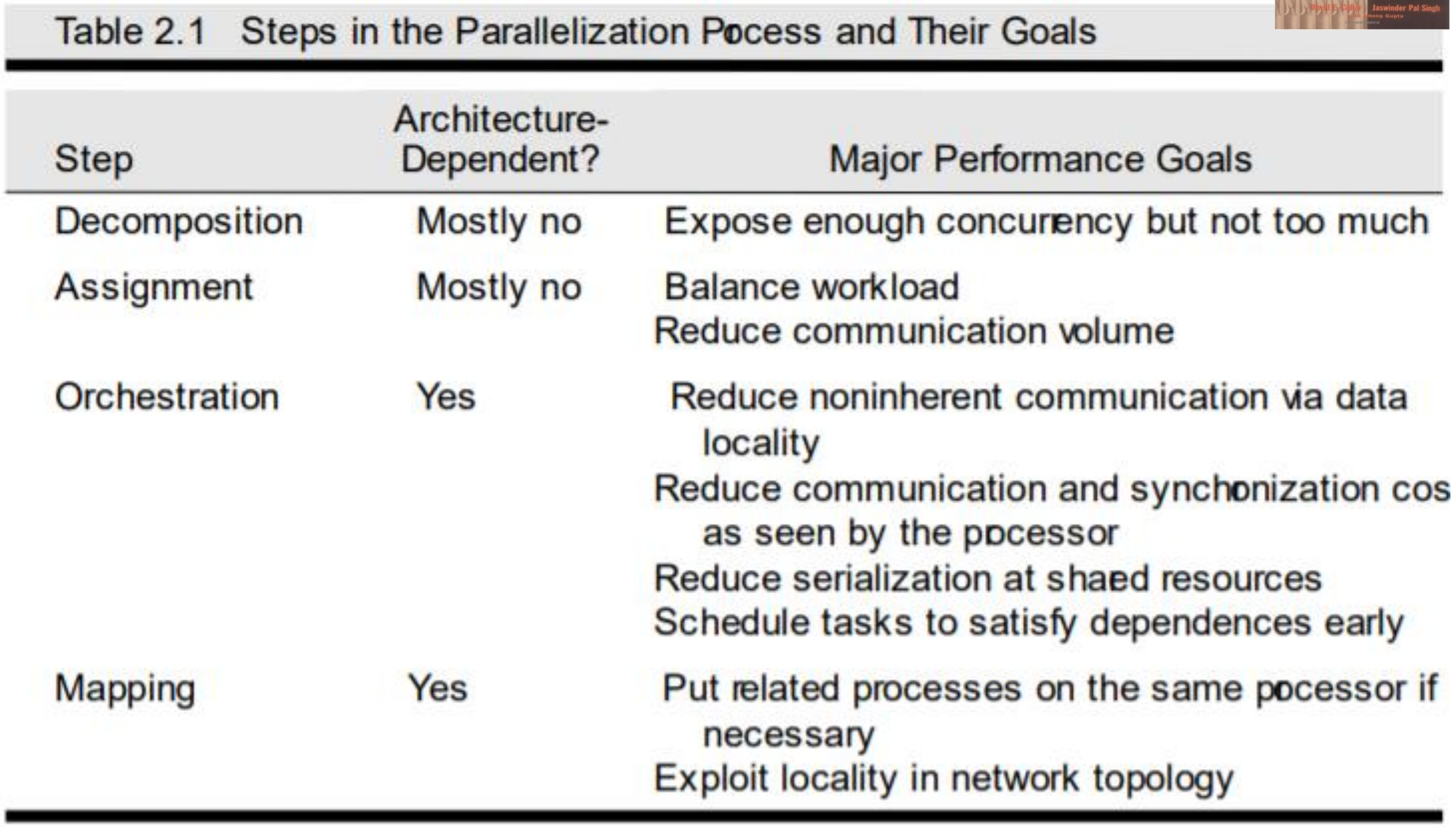
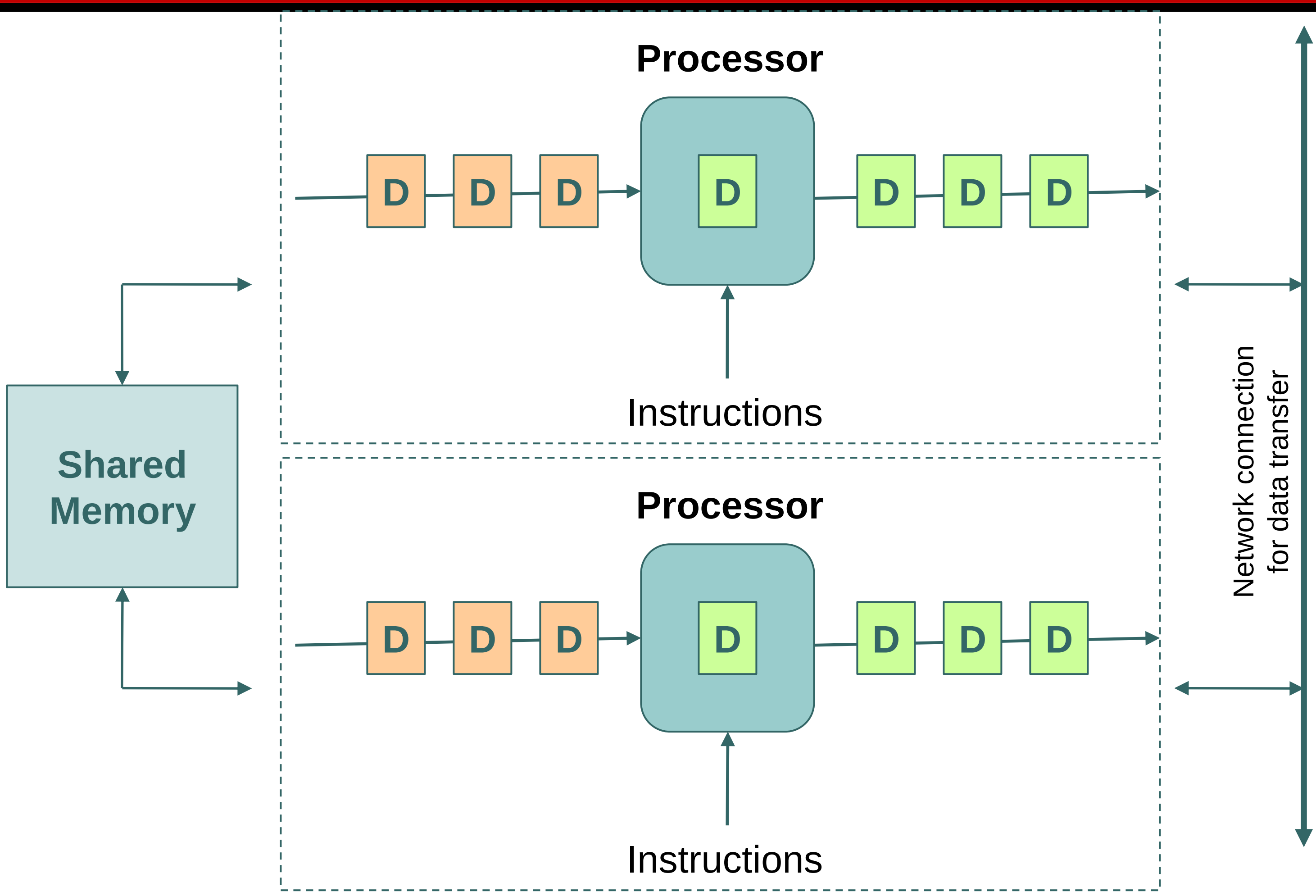


Table 2.1 Steps in the Parallelization Pocess and Their Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchonization cos as seen by the pocessor Reduce serialization at shaed resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put elated processes on the same pocessor if necessary Exploit locality in network topology

How to execute the cooperated units? – Parallel/Distributed



Parallel: Multiple CPUs within a shared memory machine

Distributed: Multiple machines with own memory connected over a network

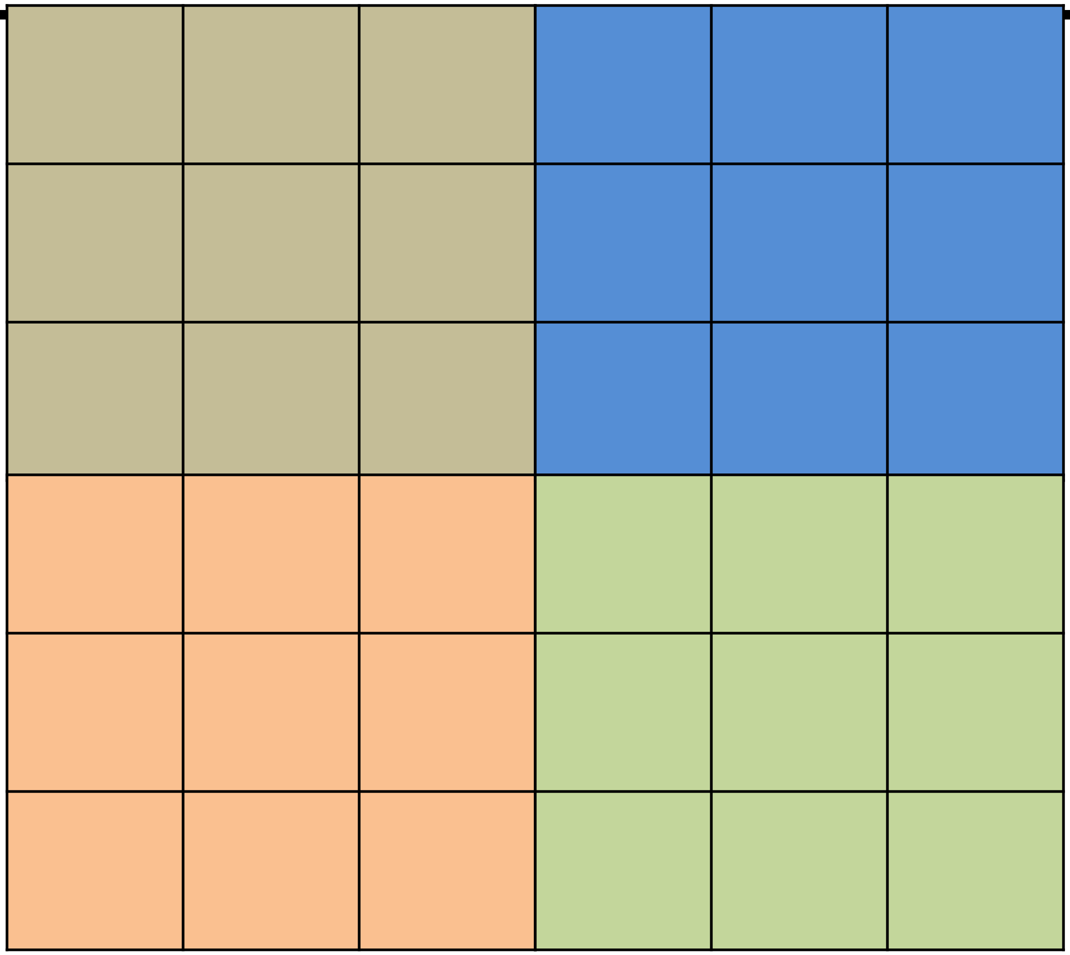
Chapter 02: HPC with example

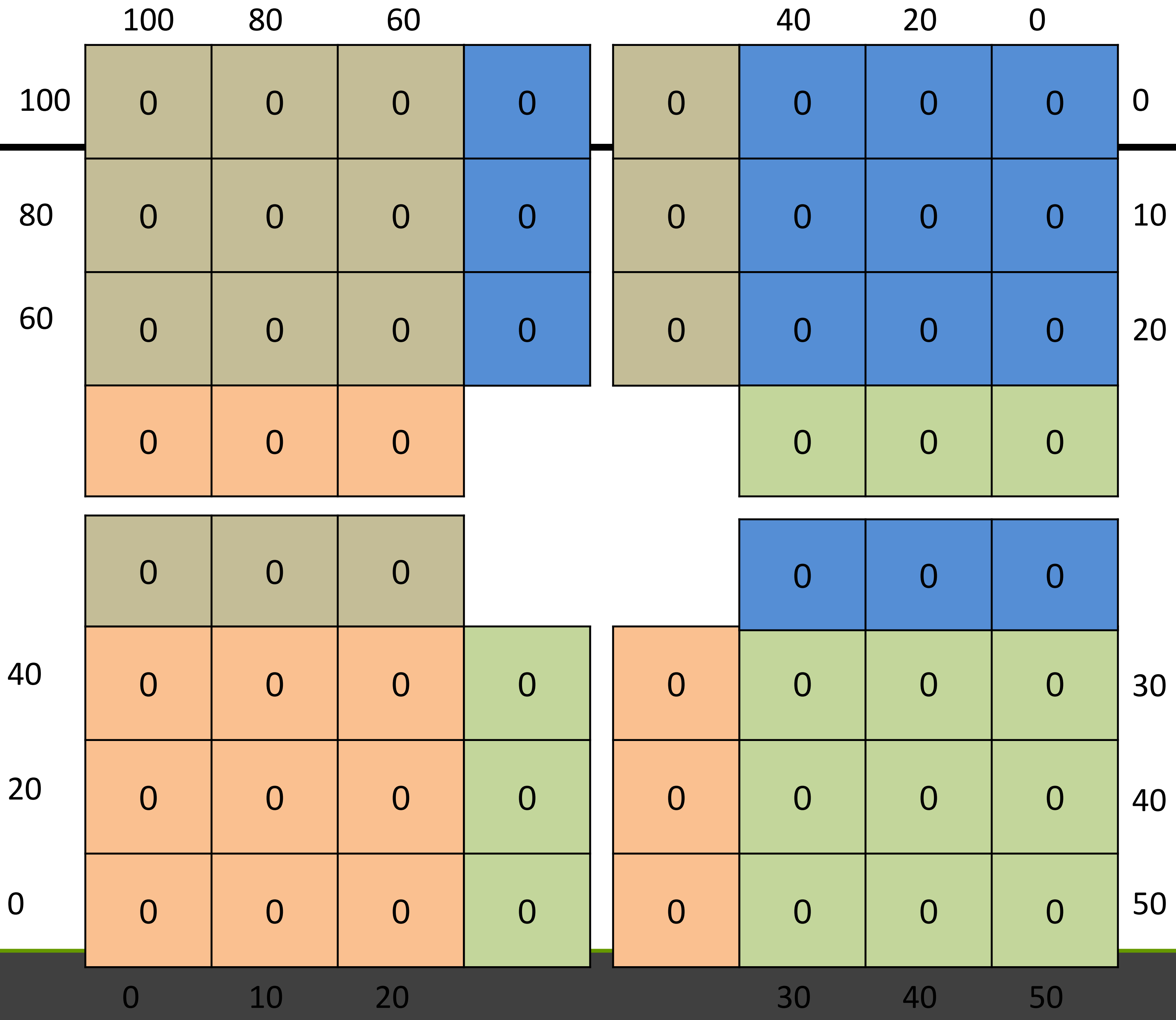
Faster for larger data

- Sequential implementation with Python
  - "Using Python to Solve Computational Physics Problems"
- Ideas to convert Sequential to Parallel
  - Hint to get the **EU**s for the Heat Equation
- Measure the performance

Hint to get the EUs for the Heat Equation

1. **You have to convert your program into EUs**
   - **DAOM**/PCAM
2. **Choose environment to finish EUs**
   - Systems
     - **P**arallel: multi-processor system – Multi-Core, GPU, MPP, …
     - **D**istributed: Cluster
   - Frameworks
     - Data parallel, SAS (Shared Address Space), Message passing
     - MPI (P or D), CUDA (P), MR (D)
3. **Execute**

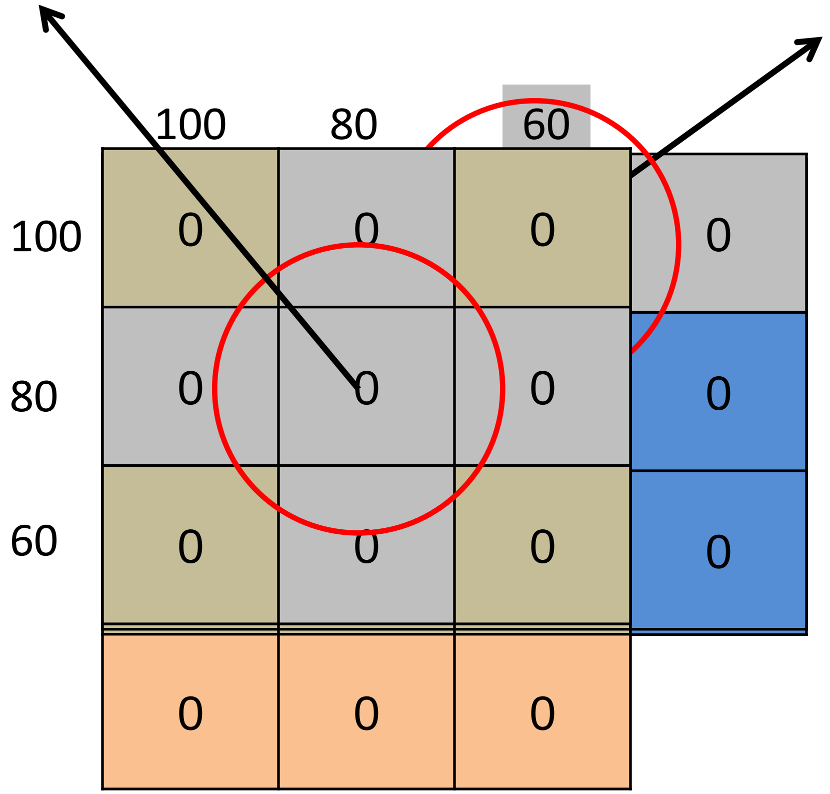



100
80
60
40
20
0
100
80
60
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
10
20
40
20
0
30
40
50
0
10
20
30
40
50

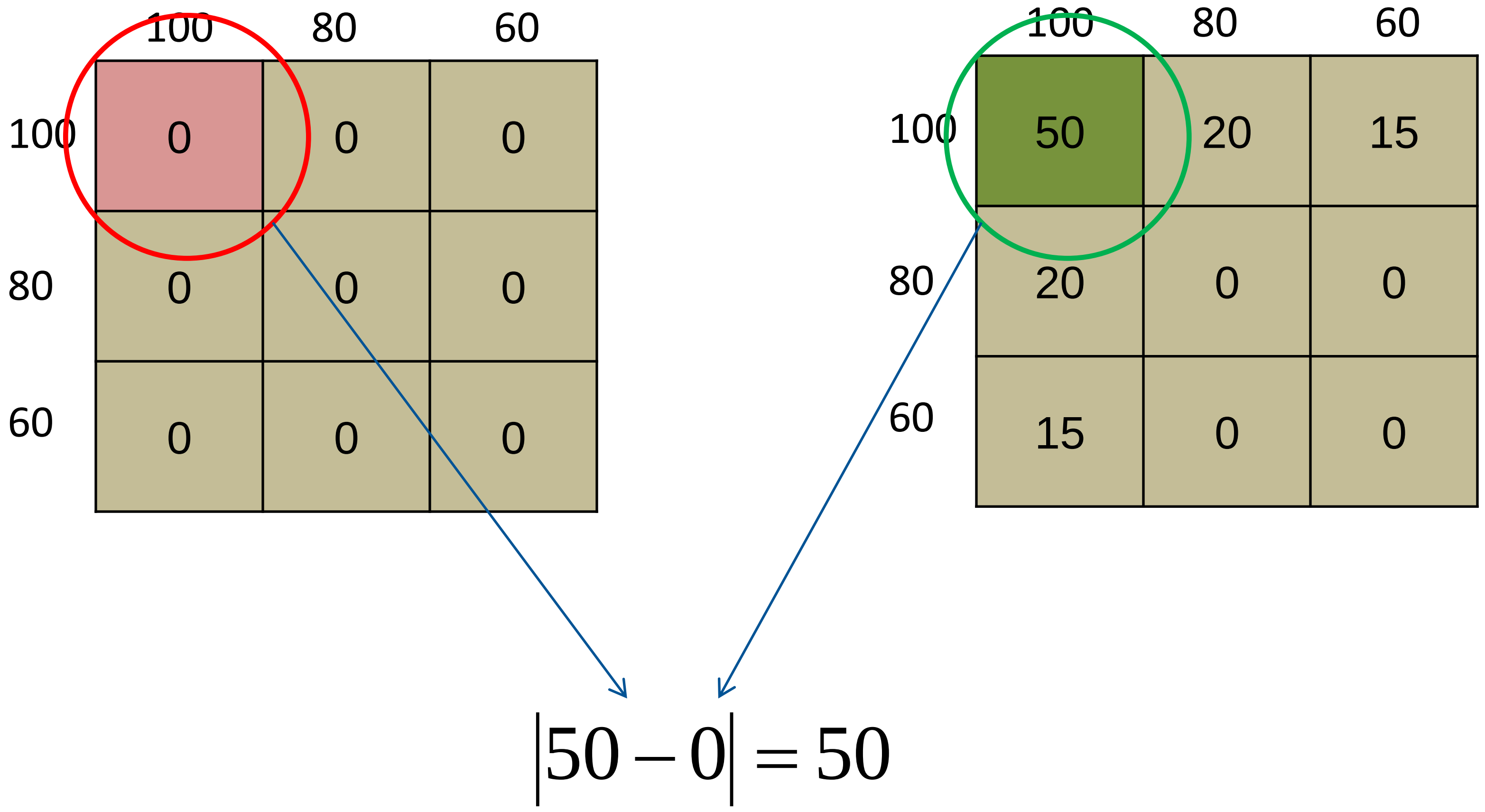
Calculate the value of each cell by averaging its 4 neighboring cells

$$\frac{0+0+0+0}{4}=\frac{0}{4}=0$$

$$\frac{60+0+0+0}{4}=\frac{60}{4}=15$$



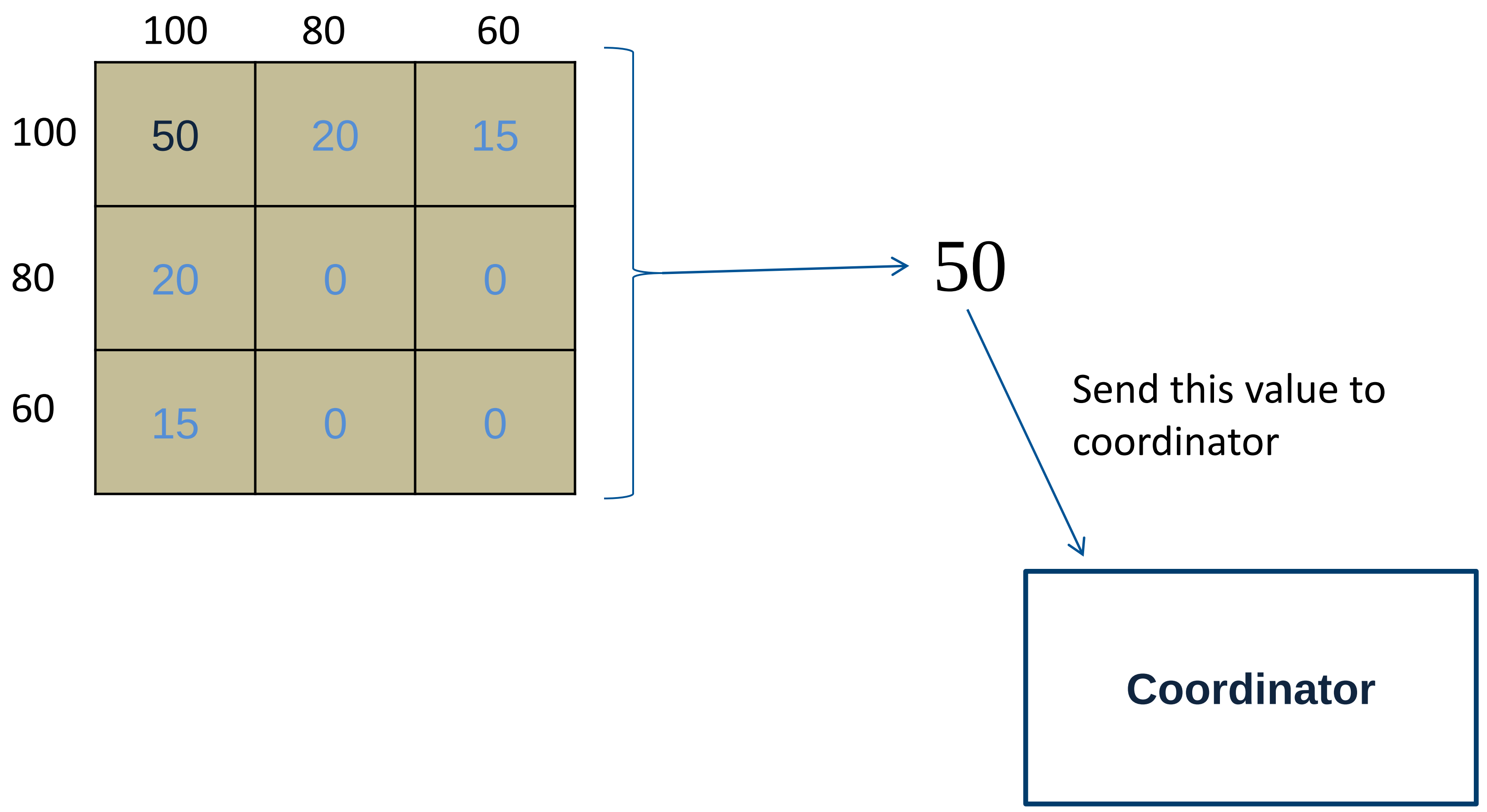
Calculate the difference between the previous cell values and new cell values



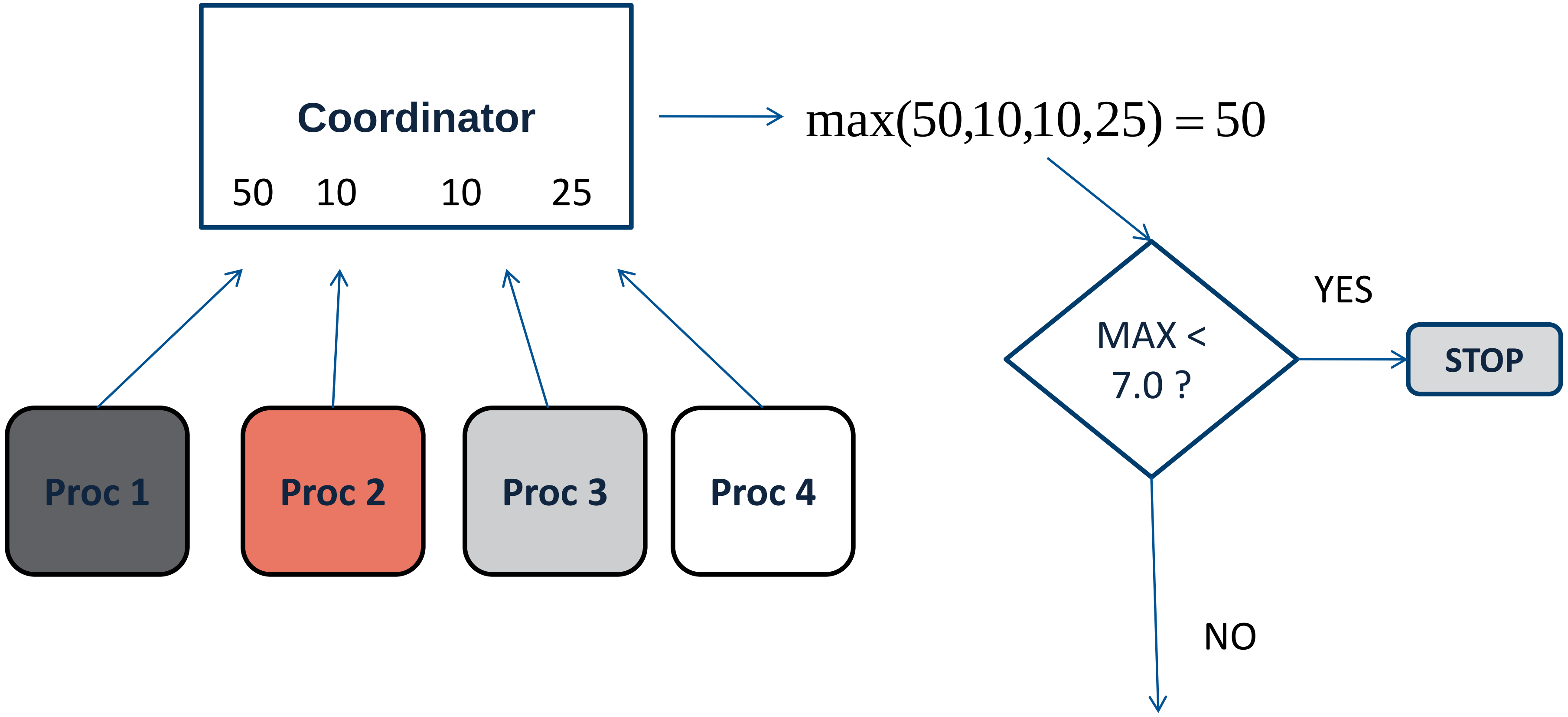
$$\begin{array}{c|ccc} & 100 & 80 & 60 \\ \hline 100 & 0 & 0 & 0 \\ 80 & 0 & 0 & 0 \\ 60 & 0 & 0 & 0 \end{array} \qquad \begin{array}{c|ccc} & 100 & 80 & 60 \\ \hline 100 & 50 & 20 & 15 \\ 80 & 20 & 0 & 0 \\ 60 & 15 & 0 & 0 \end{array}$$

$$|50 - 0| = 50$$

After computing the difference for each cell, Determine the Maximum Temperature ACROSS your problem chunk

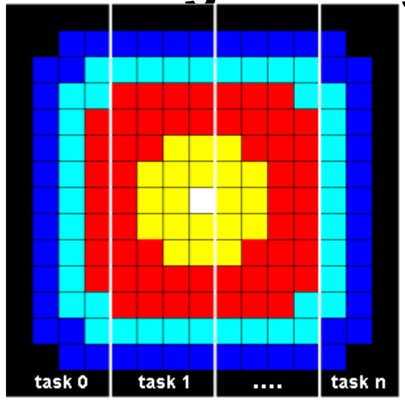


Coordinator Waits for all processing elements to send their values and determines the maximum of all the values it receives



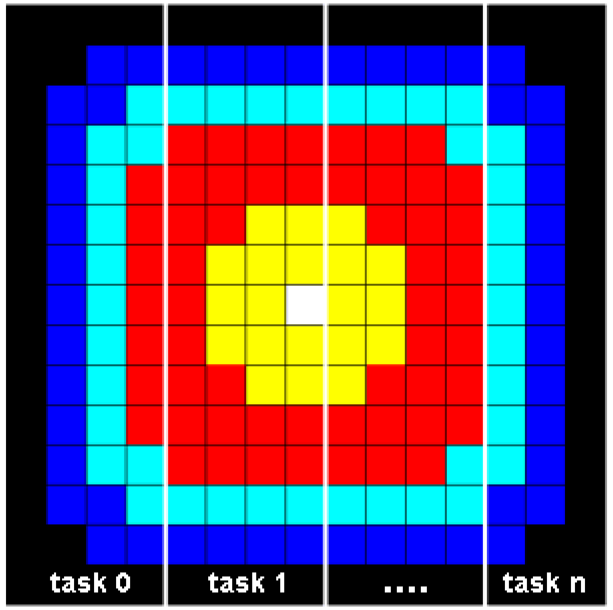
Parallel Solution 1

- Implement as an SAS/SPMD (Shared Address Space/Single Program Multiple Data) model



- The entire array is partitioned and distributed as subarrays to all tasks. Each task owns a portion of the total array.
- Determine data dependencies
  - interior elements belonging to a task are independent of other tasks
  - border elements are dependent upon a neighbour task's data, necessitating communication.
- Master process sends initial info to workers, checks for convergence and collects results
- Worker process calculates solution, communicating as necessary with neighbour processes
- Pseudo code solution: red highlights changes for parallelism.

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time
    send neighbors my border info
    receive from neighbors their border info

    update my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```



Parallel Solution 2

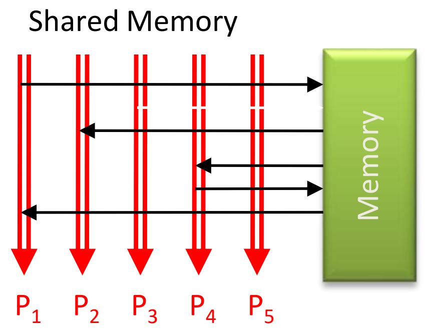
Overlapping Communication and Computation

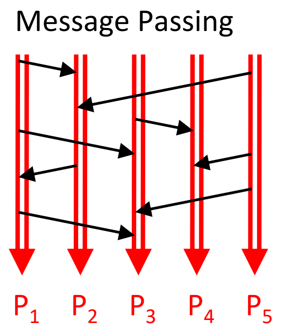
- **In the previous solution, it was assumed that blocking communications were used by the worker tasks.**
  - Blocking communications wait for the communication process to complete before continuing to the next program instruction.
  - In the previous solution, neighbour tasks communicated border data, then each process updated its portion of the array.
- **Computing times can often be reduced by using non-blocking communication.**
  - Non-blocking communications allow work to be performed while communication is in progress.
  - Each task could update the interior of its part of the solution array while the communication of border data is occurring, and update its border after communication has completed.
- **Pseudo code for the second solution: red highlights changes for non-blocking communications.**

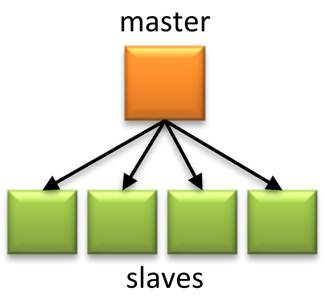
```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time

    non-blocking send neighbors my border info
    non-blocking receive neighbors border info

    update interior of my portion of solution array
    wait for non-blocking communication complete
    update border of my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```

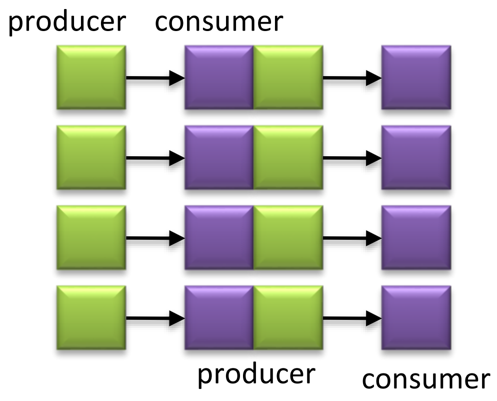
Summary

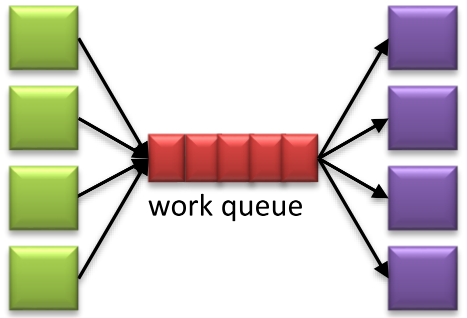
- **Programming models**
  - Shared memory (threads)
  - Message passing (MPI)
- **Design Patterns**
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues



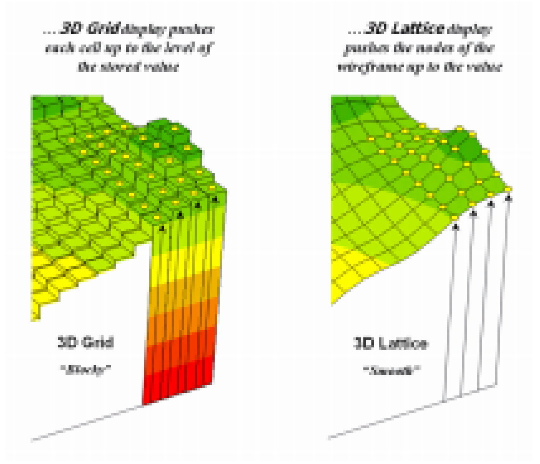




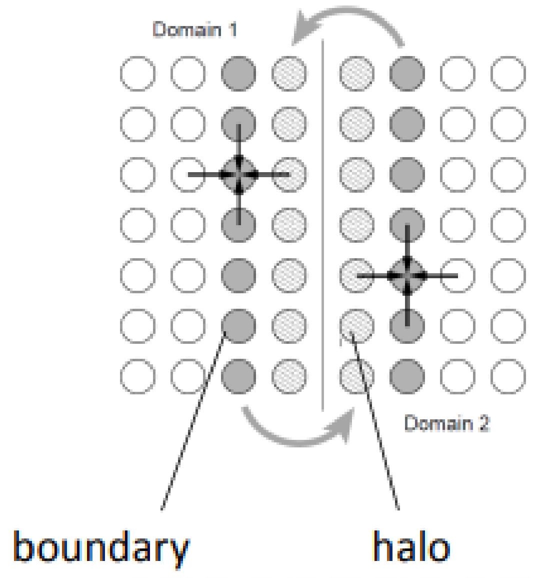


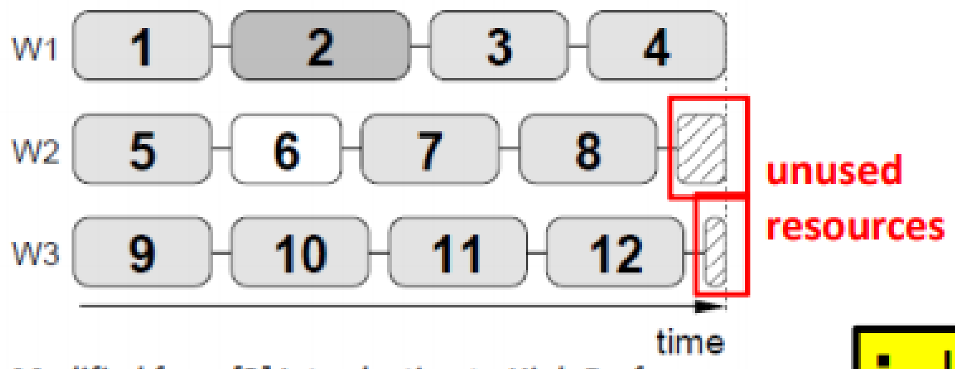


Challenges: Domain Decomposition & Load Imbalance

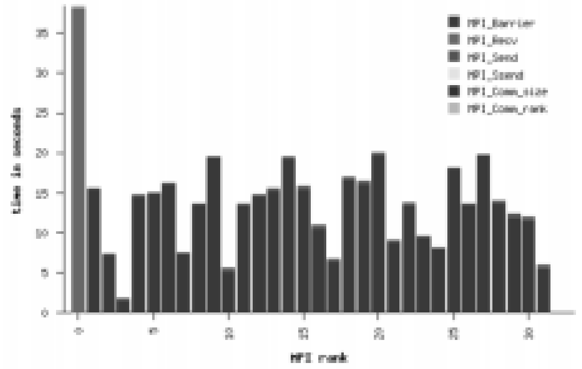

...3D Grid display pushes each cell up to the level of the stored value
...3D Lattice display pushes the nodes of the wireframe up to the value
3D Grid
"Blocky"
3D Lattice
"Smooth"

[16] Map Analysis - Understanding Spatial Patterns and Relationships, Book



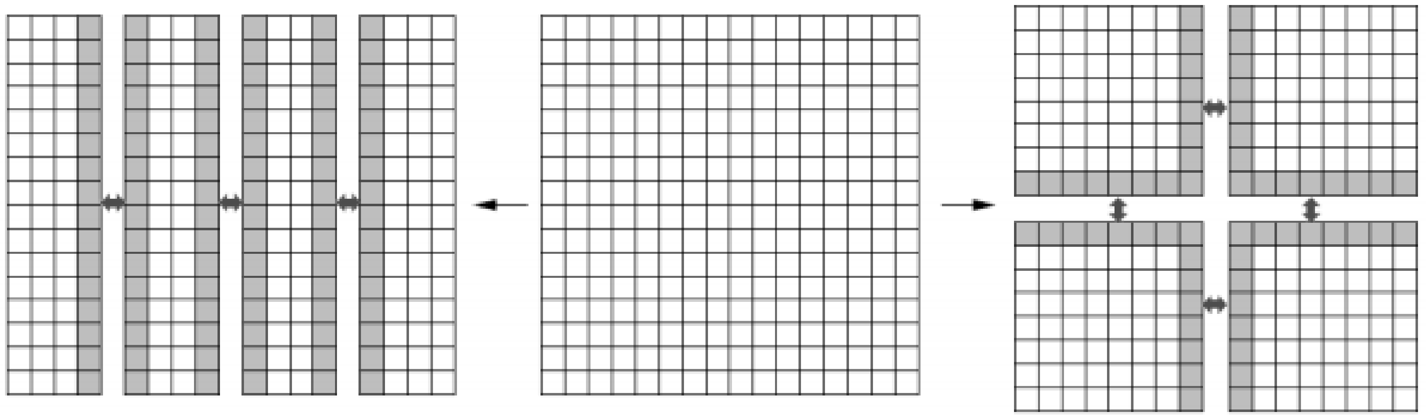


Modified from [2] Introduction to High Performance Computing for Scientists and Engineers



- Load imbalance hampers performance, because some resources are underutilized

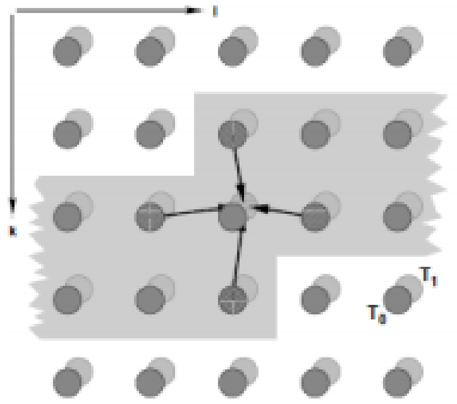
Challenges:
Ghost/Halo Regions & Stencil Methods

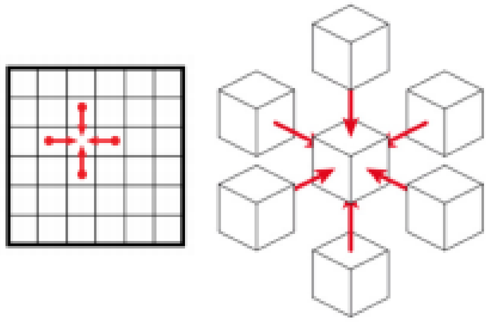


3 * 16 = 48

[2] Introduction to High Performance Computing for Scientists and Engineers

4 * 8 = 32





- Stencil-based iterative methods update array elements according to a fixed pattern called ‘stencil‘
- The key of stencil methods is its regular structure mostly implemented using arrays in codes

Chapter 02: HPC with example

- **Faster for larger data**
  - Sequential implementation with Python
    - "Using Python to Solve Computational Physics Problems"
  - Ideas to convert Sequential to Parallel
    - Shared Memory programming, Distributed Memory programming
    - Hint to get the EUs for the Heat Equation
  - Measure the performance

Units of Measure

- High Performance Computing (HPC) units are:
  - Flop: floating point operation, usually double precision unless noted
  - Flop/s: floating point operations per second
  - Bytes: size of data (a double precision floating point number is 8 bytes)
- Typical sizes are millions, billions, trillions…

Mega	Mflop/s = 10^6 flop/sec	Mbyte = 2^{20} = 1048576 ~ 10^6 bytes
Giga	Gflop/s = 10^9 flop/sec	Gbyte = 2^{30} ~ 10^9 bytes
Tera	Tflop/s = 10^{12} flop/sec	Tbyte = 2^{40} ~ 10^{12} bytes
Peta	Pflop/s = 10^{15} flop/sec	Pbyte = 2^{50} ~ 10^{15} bytes
Exa	Eflop/s = 10^{18} flop/sec	Ebyte = 2^{60} ~ 10^{18} bytes
Zetta	Zflop/s = 10^{21} flop/sec	Zbyte = 2^{70} ~ 10^{21} bytes
Yotta	Yflop/s = 10^{24} flop/sec	Ybyte = 2^{80} ~ 10^{24} bytes

- Current fastest (public) machine ~ 55 Pflop/s, 3.1M cores
  - Up-to-date list at www.top500.org

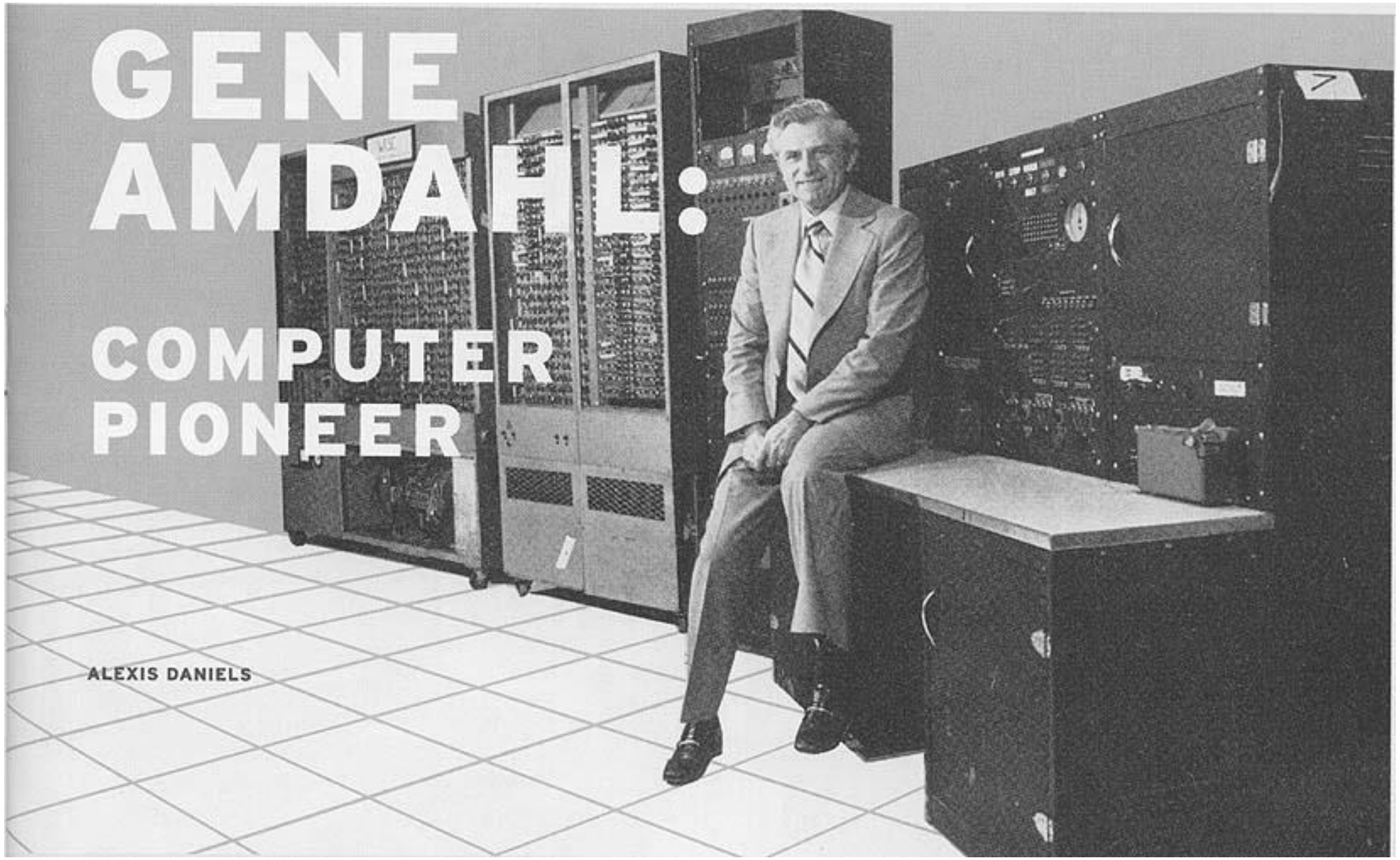
Linpack (LINear algebra PACKage) Overview

- Introduced by Jack Dongarra in 1979
- Based on LINPACK linear algebra package developed by J. Dongarra, J. Bunch, C. Moler and P. Stewart (now superseded by the LAPACK library)
- Solves a dense, regular system of linear equations, using matrices initialized with pseudo-random numbers
- Provides an estimate of system's effective floating-point performance
- Does not reflect the overall performance of the machine!

Is parallelization worth it ?

- **We parallelize our programs in order to run them faster**
- **How much faster will a parallel program run?**
  - Suppose that the sequential execution of a program takes T_1 time units and the parallel execution on p processors takes T_p time units
  - Suppose that out of the entire execution of the program, s fraction of it is not parallelizable while $1-s$ fraction is parallelizable
  - Then the speedup (Amdahl's formula):

$$\frac{T_1}{T_p} = \frac{T_1}{(T_1 \times s + T_1 \times \frac{1-s}{p})} = \frac{1}{s + \frac{1-s}{p}}$$



GENE AMDAHL:

COMPUTER PIONEER

ALEXIS DANIELS

Amdahl's Law: An Example

- Suppose that 80% of you program can be parallelized and that you use 4 processors to run your parallel version of the program
- The speedup you can get according to Amdahl is:

$$\frac{1}{s+\frac{1-s}{p}} = \frac{1}{0.2+\frac{0.8}{4}} = 2.5 \text{ times}$$

- Although you use 4 processors you cannot get a speedup more than 2.5 times (or 40% of the serial running time)

高性能并行计算

迟学斌 中国科学院计算机网络信息中心

chi@sccas.cn, chi@sc.cnic.cn

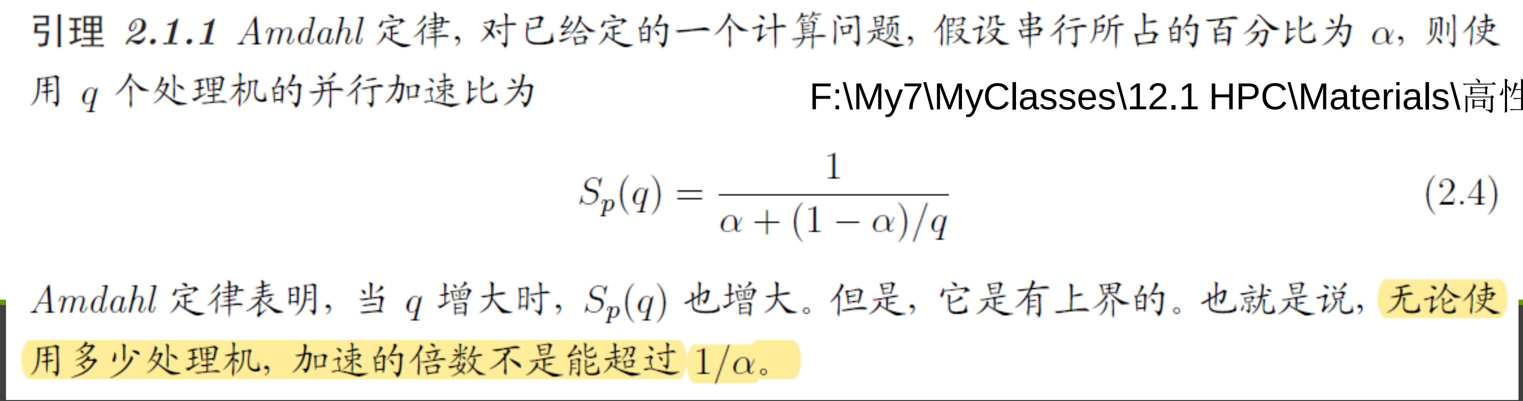
http://lssc.cc.ac.cn/

http://www.sccas.cn/

http://www.scgrid.cn/

http://www.cngrid.org/

2005 年 4 月 6 日



引理 *2.1.1* *Amdahl* 定律，对已给定的一个计算问题，假设串行所占的百分比为 α，则使用 q 个处理机的并行加速比为

$$S_p(q) = \frac{1}{\alpha+(1-\alpha)/q} \qquad (2.4)$$

Amdahl 定律表明，当 q 增大时，$S_p(q)$ 也增大。但是，它是有上界的。也就是说，无论使用多少处理机，加速的倍数不是能超过 $1/\alpha$。

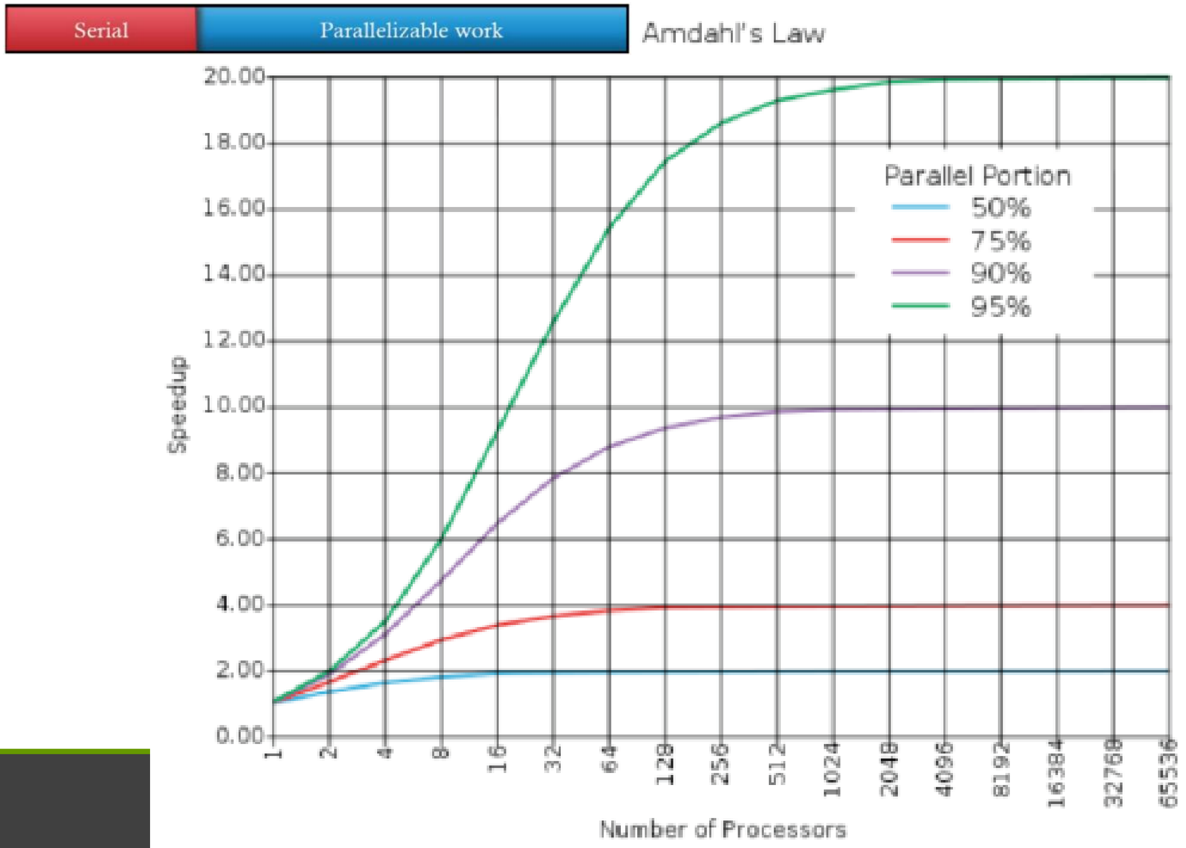
F:\My7\MyClasses\12.1 HPC\Materials\高性能并行计算(2005年4月6日).pdf

Amdahl's law: $R = \frac{1}{(1-p)+p/N}$

p: fraction of work that can be parallelized

$1 - p$: fraction of sequential work

R: prediction maximum speed-up using N parallel processors

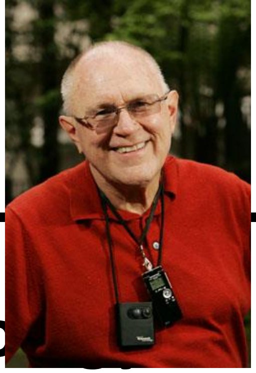


计 算 机 科 学 丛 书
高性能科学与工程计算
（德）Georg Hager Gerhard Wellein 著
Introduction to High Performance Computing for Scientists and Engineers
Introduction to High Performance Computing for Scientists and Engineers
Georg Hager
Gerhard Wellein
China Machine Press

- 高性能科学与工程计算
- [德] Georg Hager, [德] Gerhard Wellein
  - 《计算机科学丛书：高性能科学与工程计算》从工程实践的角度介绍了高性能计算的相关知识。主要内容包括现代处理器的体系结构、为读者理解当前体系结构和代码中的性能潜力和局限提供了坚实的理论基础。
  - 接下来讨论了高性能计算中的关键问题，包括串行优化、并行、OpenMP、MPI、混合程序设计技术。
  - 作者根据自身的研究也提出了一些前沿问题的解决方案，如编写有效的C++代码、GPU编程等。

Bell’s Law

https://gordonbell.azurewebsites.net/



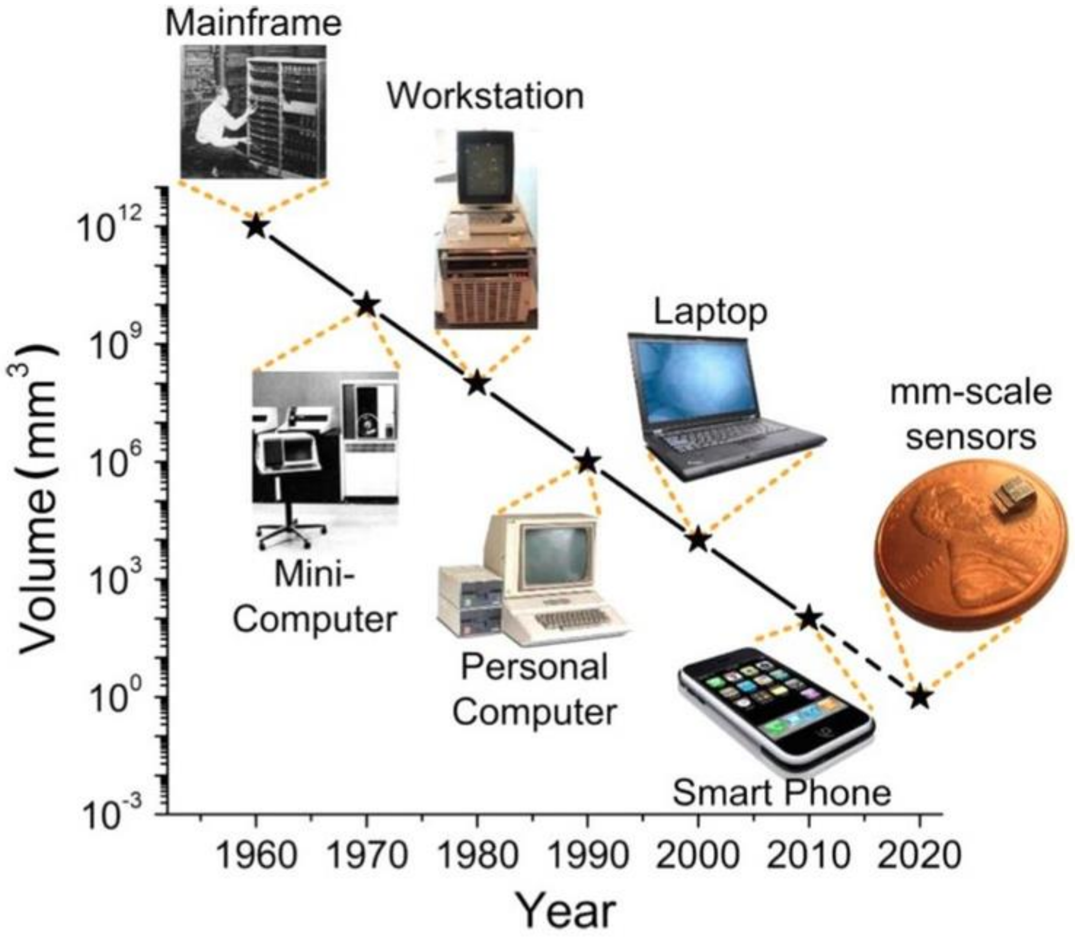
Bell's Law of Computer Class formation was discovered about 1972. It states that techno advances in semiconductors, storage, user interface and networking advance every decade enable a new, usually lower priced computing platform to form. Once formed, each class is maintained as a quite independent industry structure.

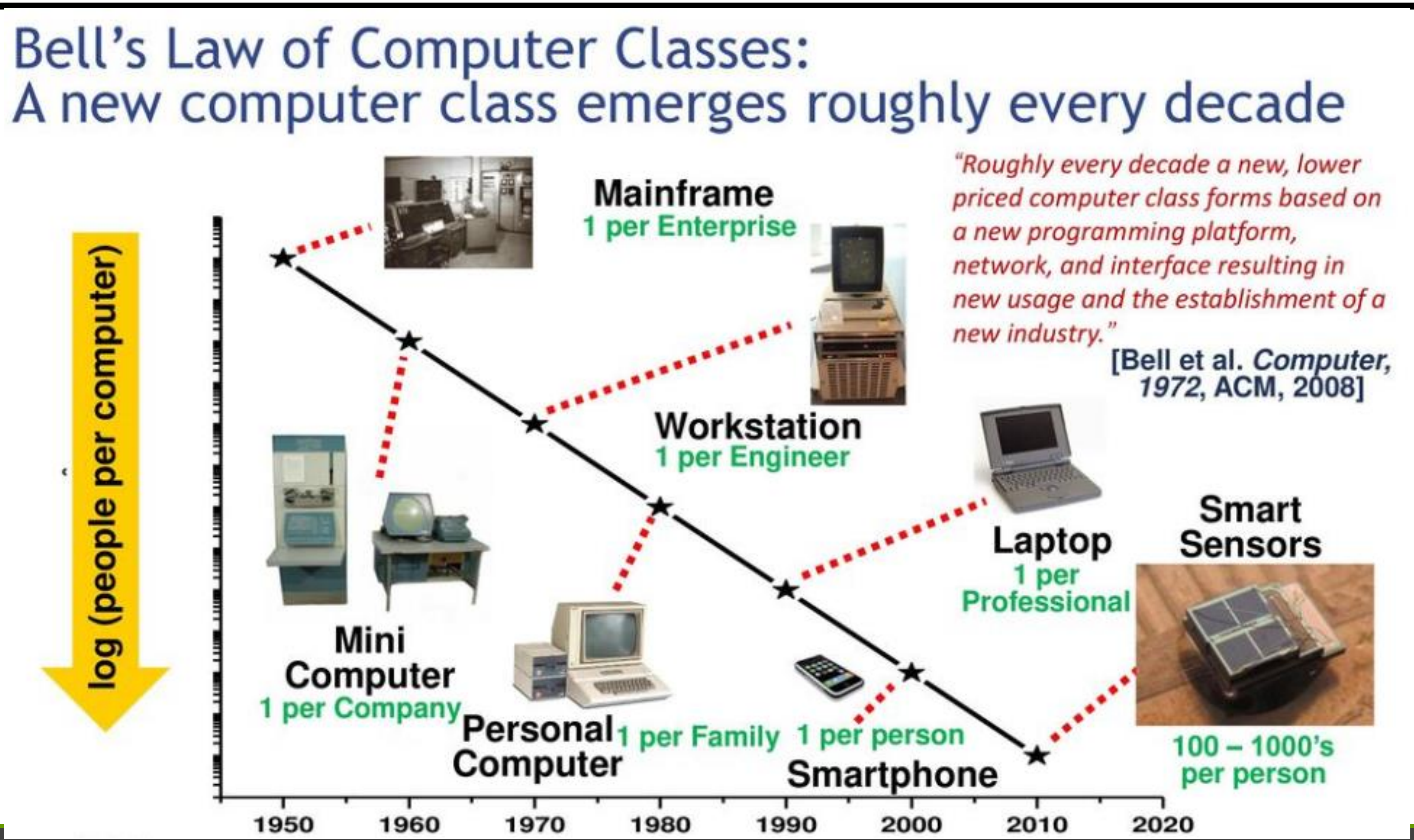
This explains mainframes, minicomputers, workstations and Personal computers, the web, emerging web services, palm and mobile devices, and ubiquitous interconnected networks. We can expect home and body area networks to follow this path.

From Gordon Bell (2007), http://research.microsoft.com/~GBell/Pubs.htm https://www.computerhistory.org/atchm/the-worlds-smallest-computer/

Bell’s Law states, that:

- **Important classes of computer architectures come in cycles of about 10 years.**





Bell's Law of Computer Classes:
A new computer class emerges roughly every decade

"Roughly every decade a new, lower priced computer class forms based on a new programming platform, network, and interface resulting in new usage and the establishment of a new industry."
[Bell et al. *Computer, 1972*, ACM, 2008]

HPC Performance Tools

- Local module setup
- Compilers*
- Libraries*

Make it work,
make it right,
make it fast.

Kent Beck

Debugger:

- TotalView
- DDT
- STAT
- MUST

Performance Tools:

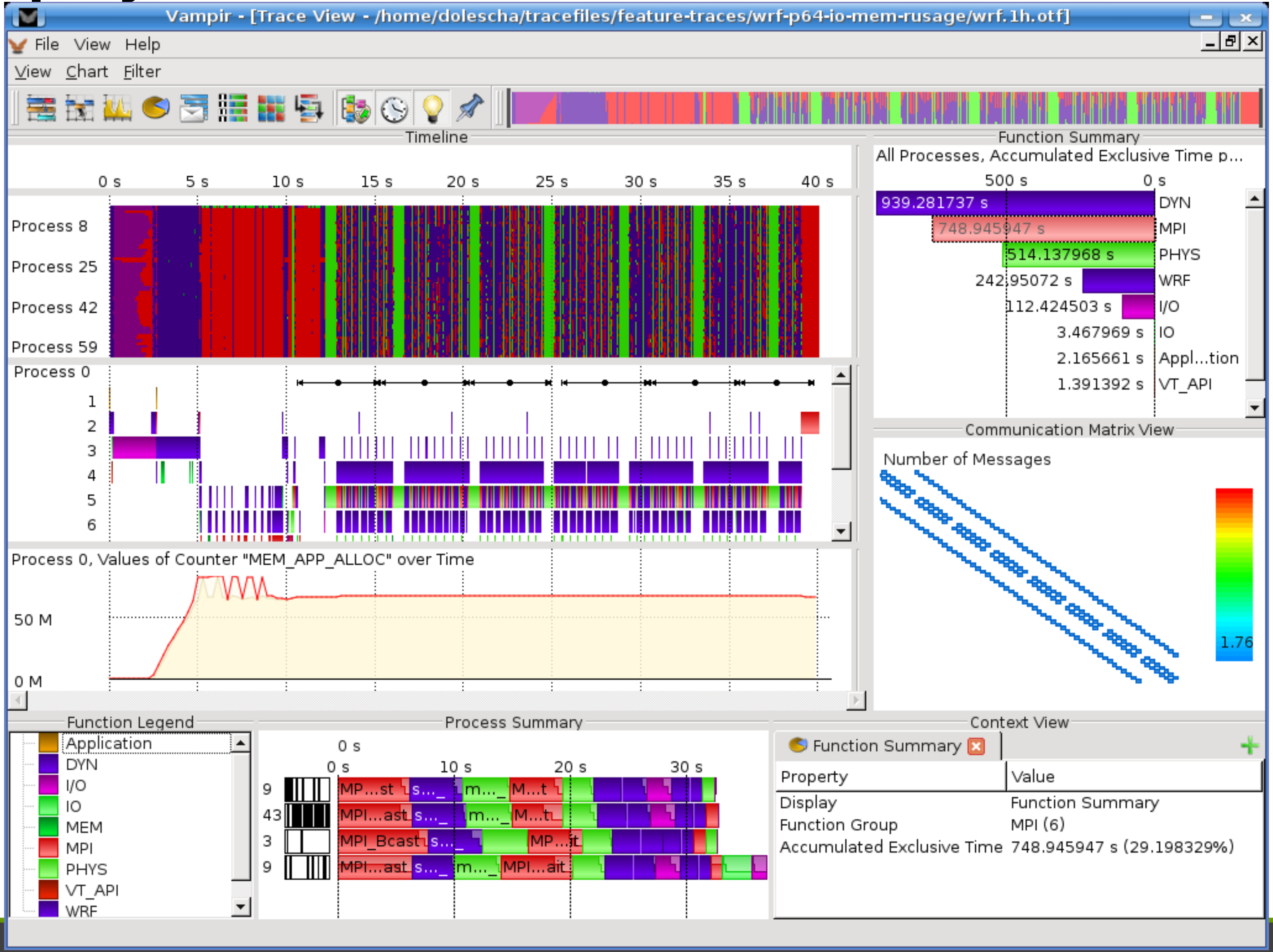
- Performance Reports
- Darshan
- ***Score-P***
- ***Scalasca***
- *Vampir*
- HPCToolkit
- TAU
- NVIDIA Visual Profiler
- Extrae/Paraver
- PAPI

Vampir Event Trace Visualizer

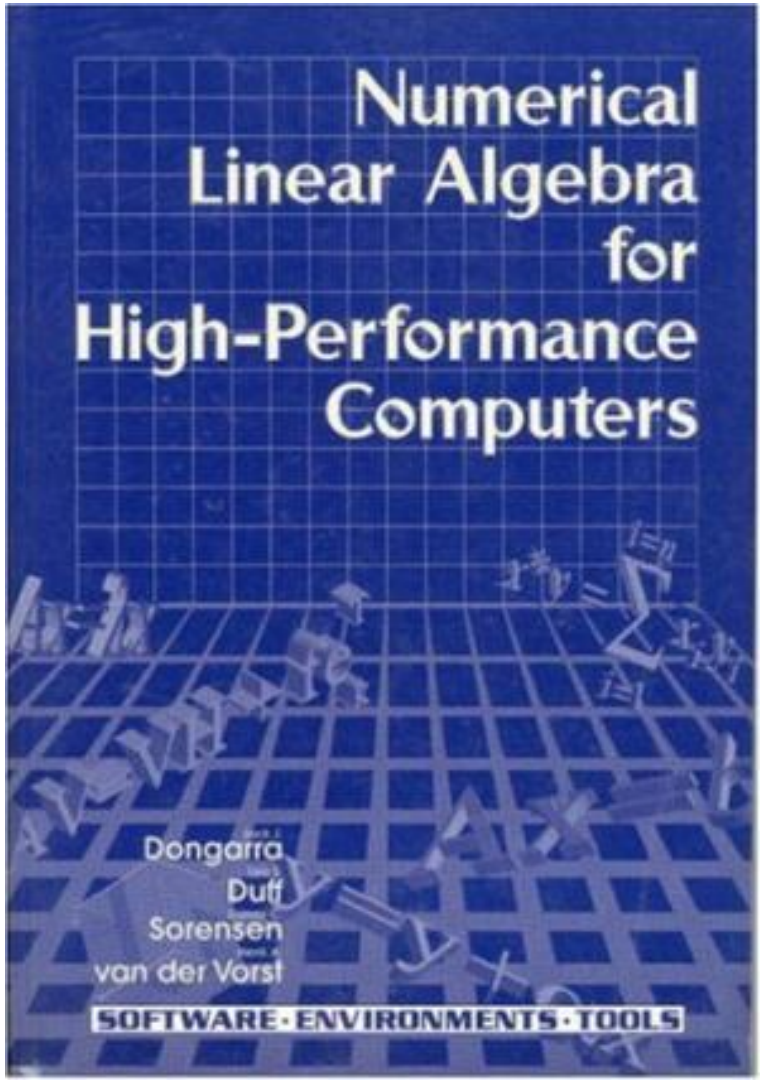
- Offline trace visualization for Score-P's OTF2 trace files
- Visualization of MPI, OpenMP and application events:
  - All diagrams highly customizable (through context menus)
  - Large variety of displays for ANY part of the trace
- http://www.vampir.eu/

- Advantage:
  - Detailed view of dynamic application behaviour
- Disadvantage:
  - Requires event traces (huge amount of data)
  - Completely manual analysis

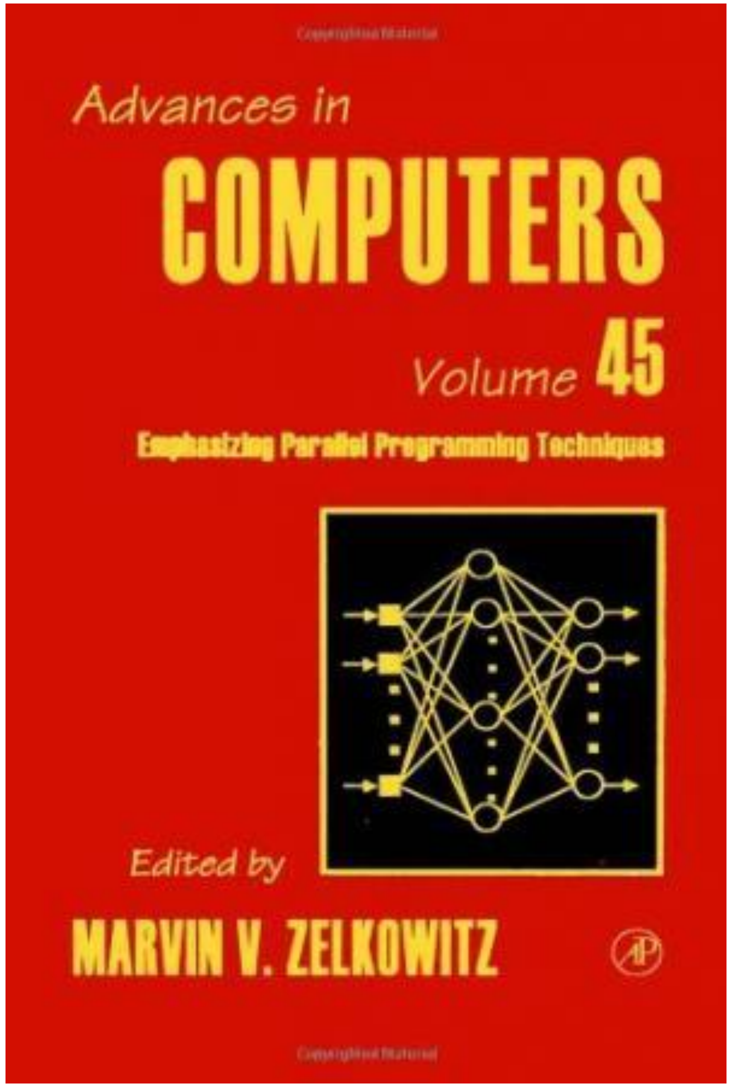
Vampir Displays



Property	Value
Display	Function Summary
Function Group	MPI (6)
Accumulated Exclusive Time	748.945947 s (29.198329%)



- Numerical Linear Algebra on High-Performance Computers
- *Jack J. Dongarra*, *Iain S. Duff*, *Danny C. Sorensen*, *Hank A. van der Vorst*
- **1987**


Copyrighted Material

Advances in

COMPUTERS

Volume 45

Emphasizing Parallel Programming Techniques

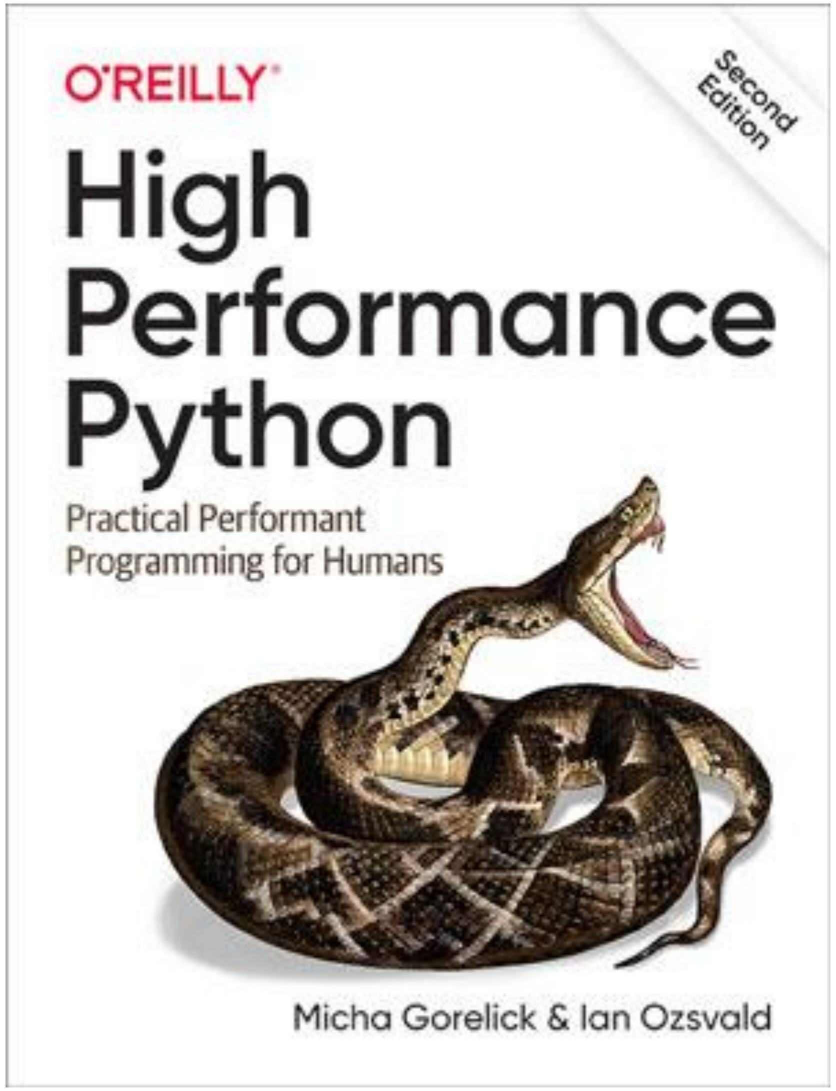
Edited by

MARVIN V. ZELKOWITZ

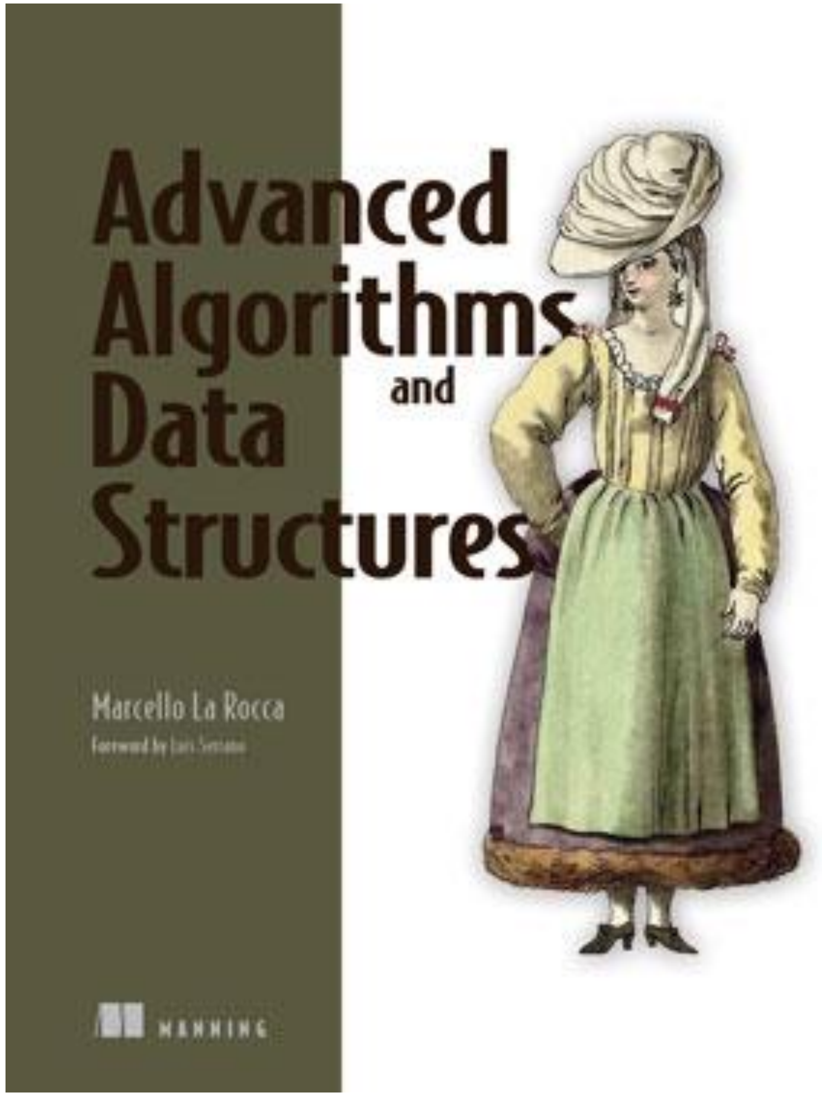
AP

Copyrighted Material

- Emphasizing Parallel Programming Techniques
- *Marvin Zelkowitz Ph.D. MS BS.*
- **1997**


O'REILLY
Second Edition
High Performance Python
Practical Performant Programming for Humans
Micha Gorelick & Ian Ozsvald

- High Performance Python, 2nd Edition
- Micha Gorelick, Ian Ozsvald
- April 2020



- Advanced Algorithms and Data Structures
- Marcello La Rocca
- May 2021