Introduction to Parallel Computing I

ZHANG, Rui

Department of Physics, HKUST

Sep 23rd, 2023

Materials used or referenced from the following resources:
https://cosy.univ-reims.fr/~fnolot/Download/Cours/HPC/introduction_to_parallel_computing.ppt
"Introduction to Parallel Computing" course by Aleksandar Prokopec from coursera.org
MSDM 5001 course taught by Huijie Guan in Fall 2019.
MPI course taught by Kevin Connington at CCNY.

Instructor Info

- ZHANG Rui, ruizhang@ust.hk.
- Research group link.
- Computational Soft Matter physicist interested in complex phenomena including complex fluids (biofluids, liquid crystals, active matter, etc.), micro and nano fluids, and metamaterials.
- Computational methods:
  - Molecular dynamics, Monte Carlo, etc.
  - Computational Fluid Dynamics such as lattice Boltzmann method.
  - Continuum Mechanics simulations.
  - Machine learning.

The Three Parallel Computing Sessions

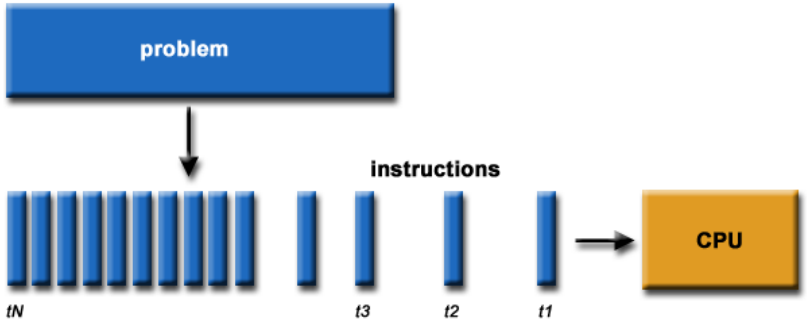
- 4th class (today):
  - Concepts, definitions, and theorems.
- 5th class:
  - Hands-on experience using multiprocessing and multithreading modules in Python.
- 6th class:
  - Hands-on experience of parallel computing using MPI.
  - GPU computing.
  - In-class exercises.
  - Announcement of assignments and mini project.

Outline of Today's Class

- Definition & History
- Architecture & Platform
- Algorithm examples and time complexity
- Design Strategy
- Analysis of Speedup & Efficiency
- "*Hello World*" Example
- Installation Guidance

What is *Serial Computing* in the first place?

- Traditionally, software has been written for *serial* computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may be executed at any moment in time.

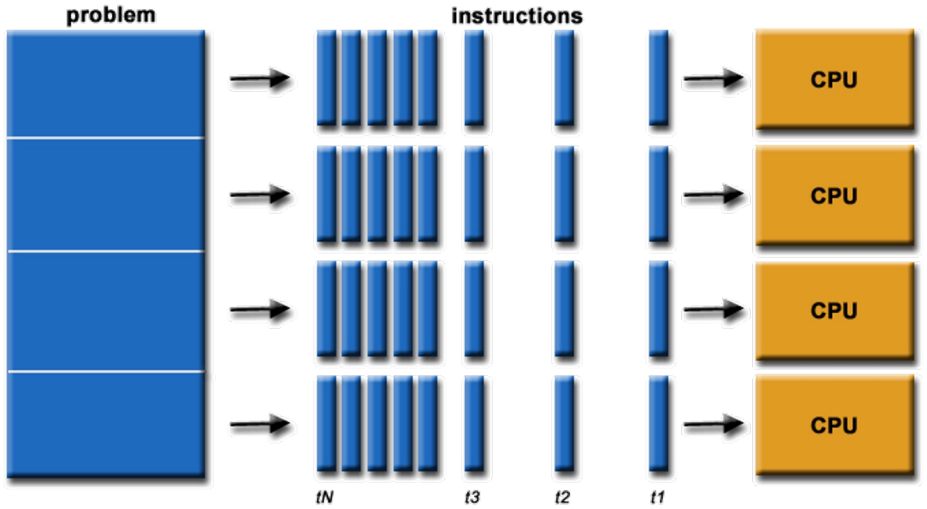


A *Turing machine* or a computing machine as Turing called it, in Turing's original definition defines an abstract machine that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, given any computer algorithm, a Turing machine capable of simulating that algorithm's logic can be constructed. - Wikipedia.

More details of Turing machine can be found in https://plato.stanford.edu/entries/turing-machine/

What is *Parallel Computing*?

- *Definition* - In contrast to serial computing, parallel computing is a type of computation in which many calculations are performed at the same time.
- *Basic principle* - The computation can be divided into smaller **subproblems**, each of which can be solved simultaneously.
- *Assumption* - Parallel hardware is available at our disposal and the computation task is parallelizable.

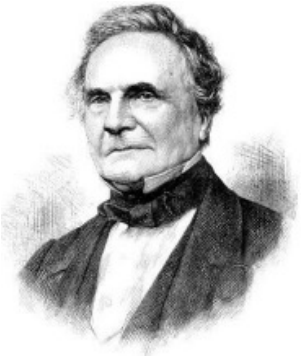


Parallel vs. Concurrency Computing

- *Parallel Computing* - it uses parallel hardware to execute computation more quickly.
  - Speedup and Efficiency are its main concern.
- *Concurrency Computing* - may or may not execute multiple executions at the same time. Improves modularity, responsiveness or maintainability.
  - Used for web server, user interface, database etc.
  - For convenience, responsiveness, maintainability.
- Q: Is quantum computing serial, parallel, or concurrency computing?
  - A: Parallel.
- Q: If DNAs in cells can be regarded as a computer, what type of computing they are doing?
  - A: Parallel again. But each DNA chain is doing serial computing.
- The same question can be posed regarding brain.

In computer science, *concurrency* is the ability of different parts or units of a program, algorithm or problem to be executed out-of-order or at the same time simultaneously partial order, without affecting the final outcome. This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems. - Wiki.

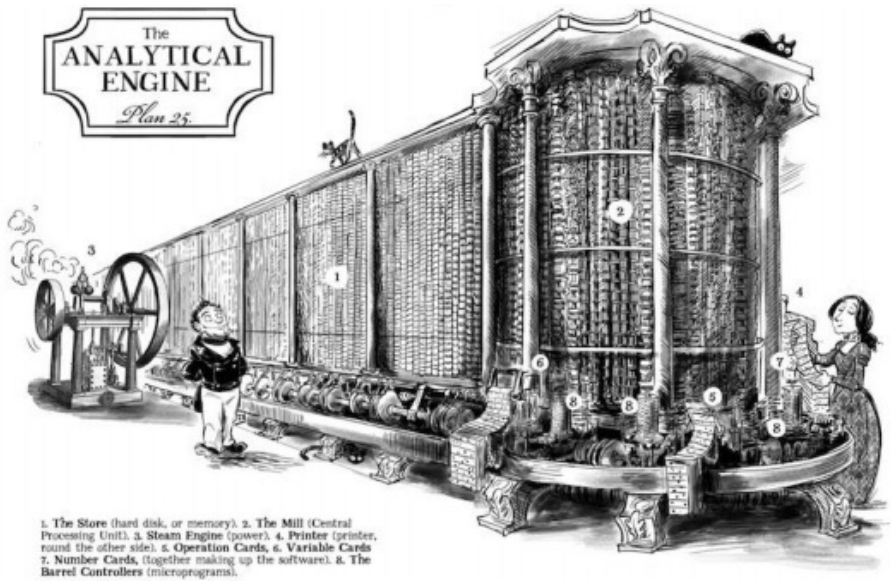
History of Parallel Computing



Charles Babbage

Luigi Menabrea

- In 1837, English mathematician and computer pioneer Charles Babbage proposed the idea of *Analytical Engine*.
- In 1842, Italian general, statesman and mathematician Luigi Menabrea published the first computer program, envisioned parallelism to speed up the performance of the Analytical Engine.



1. **The Store** (hard disk, or memory). 2. **The Mill** (Central Processing Unit). 3. **Steam Engine** (power). 4. **Printer** (printer, round the other side). 5. **Operation Cards**, 6. **Variable Cards** 7. **Number Cards**, (together making up the software). 8. **The Barrel Controllers** (microprograms).

The *Analytical Engine* incorporated an arithmetic logic unit, control flow in the form of conditional branching and loops, and integrated memory, making it the first design for a mechanical **general-purpose** computer that could be described in modern terms as Turing-complete. In other words, the logical structure of the Analytical Engine was essentially the same as that which has dominated computer design in the electronic era. – wiki.

History of Parallel Computing (Continued)

In 1964, Control Data Corporation produces the CDC 6600 supercomputer, 3 megaFLOPS. Each machine contained one 60-bit CPU and 10 peripheral processing units (PPUs).

ASCI Red Supercomputer build in 1997, 1.3 teraFLOPS, Takes up 150 m^2, 9298 Processors.

Q: what governs the historical trend of FLOPS speed?
A: Moore's law.

Tianhe-2, debuted in 2013, 33.86 petaFLOPS, 16,000 computer nodes, 3,120,000 cores. world's fastest supercomputer during 2013-2015.

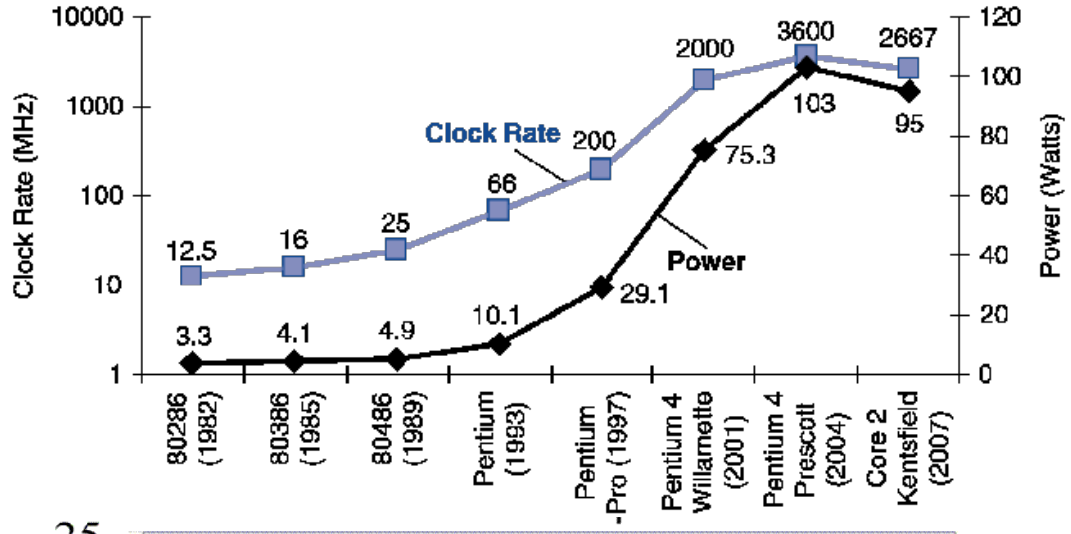
floating point operations per second (FLOPS, flops or flop/s) is a measure of computer performance, useful in fields of scientific computations that require floating-point calculations. Different floating-point instructions take different amounts of time. In measuring FLOPS, it is often assumed to be a floating point multiply or divide as these are the most intensive floating-point operations a processor can perform.

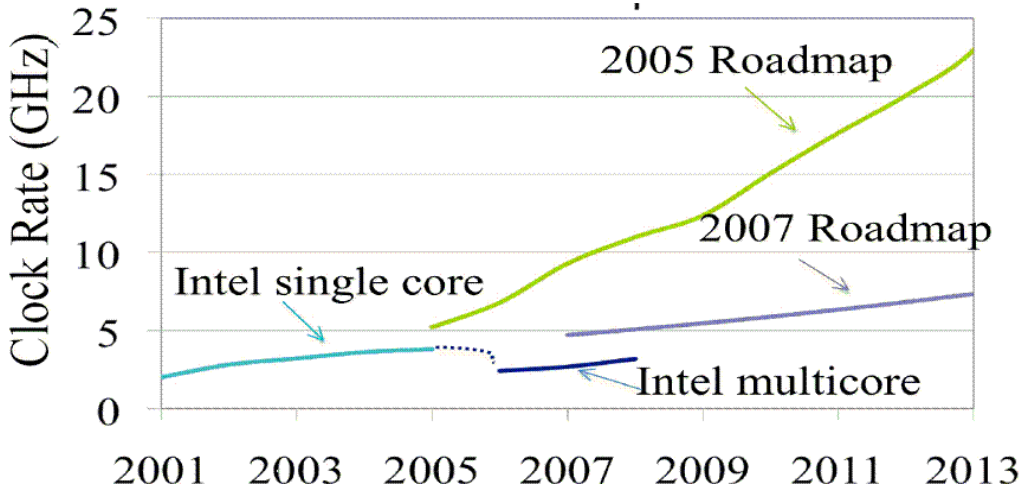
Decimal			
Value		SI	
1000	10^3	k	kilo
1000^2	10^6	M	mega
1000^3	10^9	G	giga
1000^4	10^{12}	T	tera
1000^5	10^{15}	P	peta
1000^6	10^{18}	E	exa
1000^7	10^{21}	Z	zetta
1000^8	10^{24}	Y	yotta

Popularity of Parallel Computing

- In the recent decades, multi-core processing becomes the mainstream for parallel computers.
- It is now ubiquitous in laptops, desktops, and even cellphones.
- The reason for the boom of multiprocessor machines:
  - Sequential Computer Hits the *Power Wall*.
- *Power Wall* - the limitation of CPU clock rate, CPU design, and CPU performance improvements due to the thermal and electrical power constraints, i.e., technological inability to use higher clock rates, more transistor switching elements, to use higher volumes of electrical energy and inability to maintain overall thermal stability.

Power Wall



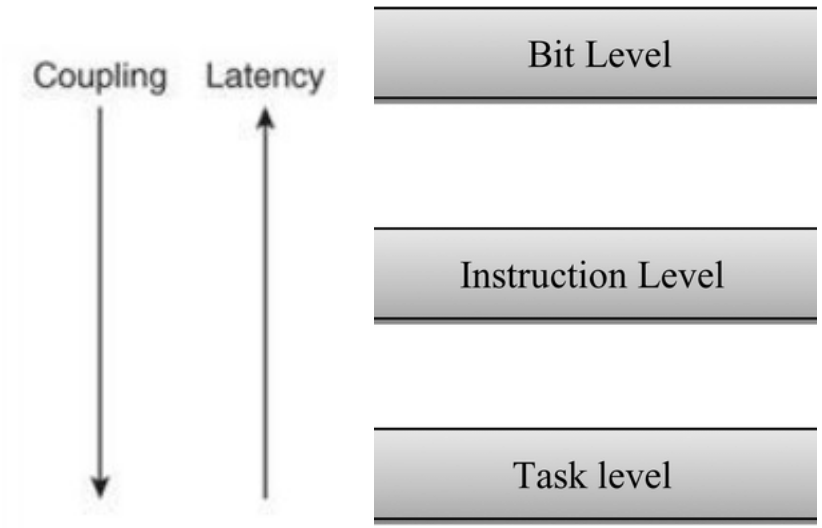


Clock rate - the maximum speed which the logic registers are working at.
Clock rate is a simple measure of performance that can be very deceptive. It was a common comparison measure from the 1970s up to the turn of the century to compare processors based on raw clock rate and decree that the higher clocked processor was "faster" without taking into account architectural differences. Two processors clocked at the same rate but one being superscalar (two or more instructions executed in parallel such as in a pipelined architecture) versus one that isn't, the former processor will win.

A *superscalar processor* is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. In contrast to a scalar processor that can execute at most one single instruction per clock cycle, a superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor. - wiki.

Various Levels of Parallel Computing

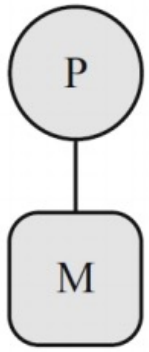
- *Bit-level parallelism* - processing multiple bits of data in parallel.
- *Instruction-level parallelism* - executing different instructions from the same instruction stream in parallel.
- *Task-level parallelism* - executing separate instruction streams in parallel.



Latency is a time delay or a time lag between cause and its effect.
Column Address Strobe (CAS) latency, or CL, is the delay in clock cycles between the READ command and the moment data is available.

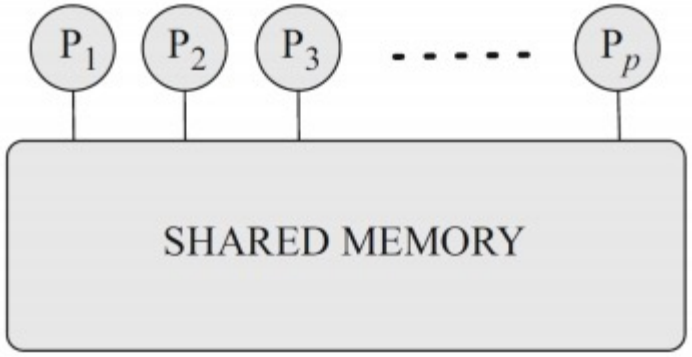
Q: which level of parallelism is superscalar technology?
A: Instruction level.

Architecture

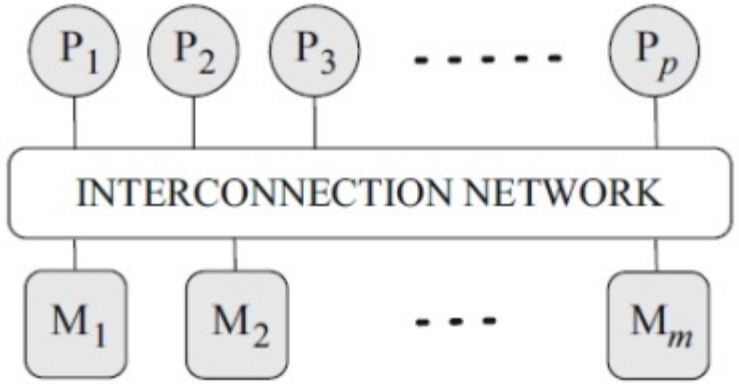


P - processor
M - memory

Serial Computing

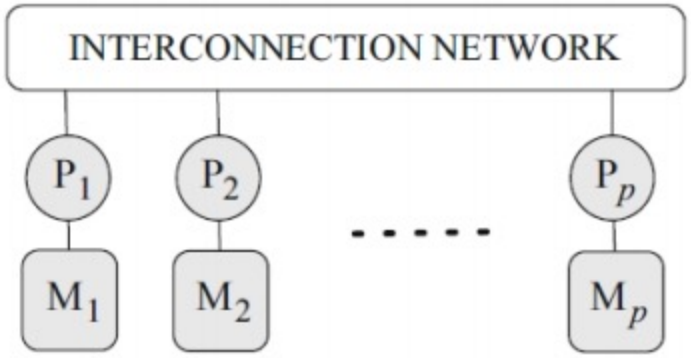


Shared memory

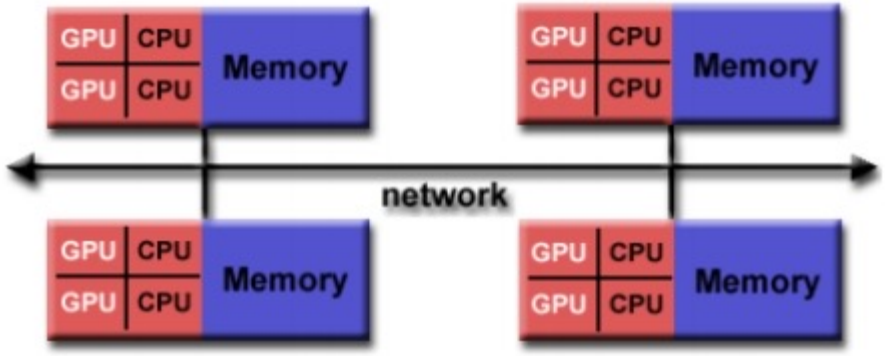


Memory module machine, e.g., GPU

InfiniBand (IB) is a computer networking communications standard used in high-performance computing that features very high throughput and very low latency. It is used for data interconnect both among and within computers. InfiniBand is also used as either a direct or switched interconnect between servers and storage systems, as well as an interconnect between storage systems. -wiki.



Private/distributed memory



Heterostructure

Parallelism Classification - Flynn's Classical Taxonomy

- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- It distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

S I S D **Single Instruction, Single Data**	**S I M D** **Single Instruction, Multiple Data**
M I S D **Multiple Instruction, Single Data**	**M I M D** **Multiple Instruction, Multiple Data**

MISD: Multiple instructions operate on one data stream. This is an uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer. -wiki.

Parallel Computing is Hard

- Identification of independent subproblems is sometimes challenging.
- Balancing work-load is tricky.
- Efficient communication/synchronizing Protocols are tricky.
- Parallel codes are error-prone.

Parallel Computing Benefits

- Save time: speedup the calculation with a price of code complexity.
- Solve larger problems - High Performance Computing (HPC)
  - As our computational power increases, the number of problems that we can seriously consider solving also increases such as Climate Modeling, Protein Folding, Drug Discovery, Energy Research, Materials Design Data Analysis.
- Other reasons:
  - Provide Concurrency (do multiple tasks at the same time).
  - Cost friendly: Microprocessor has 1% speed of a supercomputer with 0.1% cost of the supercomputer.

Parallel Computing Concepts

- *Task* - A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.
- *Parallel Task* - A task that can be executed by multiple processors safely (yields correct results) .
- *Main or root core/processor* versus *computing or task cores/processors*.
  - Root core is responsible for initialization, task dispatch, data collection, I/O etc.
- *Shared memory* versus *Distributed memory*.
  - *Shared memory* - directly address and access the same logical memory locations regardless of where the physical memory exists.
  - *Distributed memory* - tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.
- *Communications* - Parallel tasks typically need to exchange data.

Parallel Computing Concepts (Continued)

- *Synchronization* - The coordination of parallel tasks in real time, very often associated with communications, usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.
- *Observed speedup* - $S = T(1)/T(P)$ with $T(P)$ being the computational time of using *P* processors.
- *Overhead* - The amount of time required to coordinate parallel tasks, as opposed to doing useful work.
- Parallel overhead can include factors such as:
  - Task start-up time;
  - Synchronizations;
  - Data communications;
  - Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
  - Task termination time.

Parallel Computing Concepts (Continued)

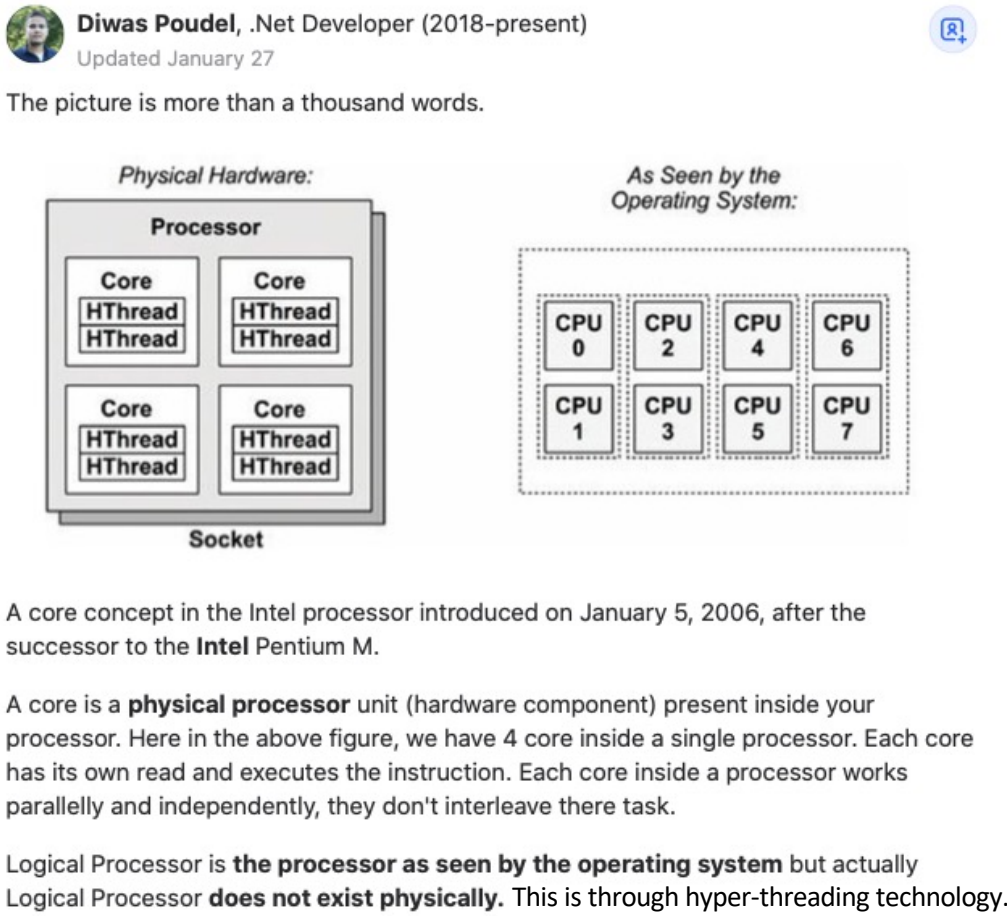
- *Scalability* **-** a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors.
- Factors that contribute to scalability include:
  - Hardware - particularly memory-cpu bandwidths and network communications;
  - Application algorithm;
  - Parallel overhead related;
  - Characteristics of your specific application and coding.

Parallelizability

- Not all computational tasks are suitable for parallelization. Below are two problems that need your judgement on whether they are parallelizable.
- Problem 1: Free energy minimization.
  - Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.
  - This problem is parallelizable because each of the molecular conformations is independently determinable.
- Problem 2: Calculation of the Fibonacci series.
  - $F(0) = 1$, $F(1) = 1$, and $F(k + 2) = F(k + 1) + F(k)$ for $k \geq 2$.
  - This is non-parallelizable because it entails dependent calculations rather than independent ones. The calculation of the $k + 2$ value uses those of both $k + 1$ and k. These three terms cannot be calculated independently and therefore, not in parallel.
- Most problems contain parallelizable and unparallelizable components.

Some Clarifications

Physical Versus Logical CPU Cores:



https://www.quora.com/What-is-the-difference-between-core-and-logical-processor

Multi-core versus superscalar processor:

A **multi-core processor** is a processor that includes multiple processing units (called "cores") on the same chip. This processor differs from a **superscalar processor**, which includes multiple execution units and can issue multiple instructions per clock cycle from one instruction stream (thread); in contrast, a multi-core processor can issue multiple instructions per clock cycle from multiple instruction streams. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is a prominent multi-core processor. Each core in a multi-core processor can potentially be superscalar as well—that is, on every clock cycle, each core can issue multiple instructions from one thread. -wiki.

Example: Global Sum – serial method

Serial Code:
P = 1, n = 24

```
sum = 0
DO i=1,n
   x=Compute_next_value(...)
   sum=sum+x
END DO
```

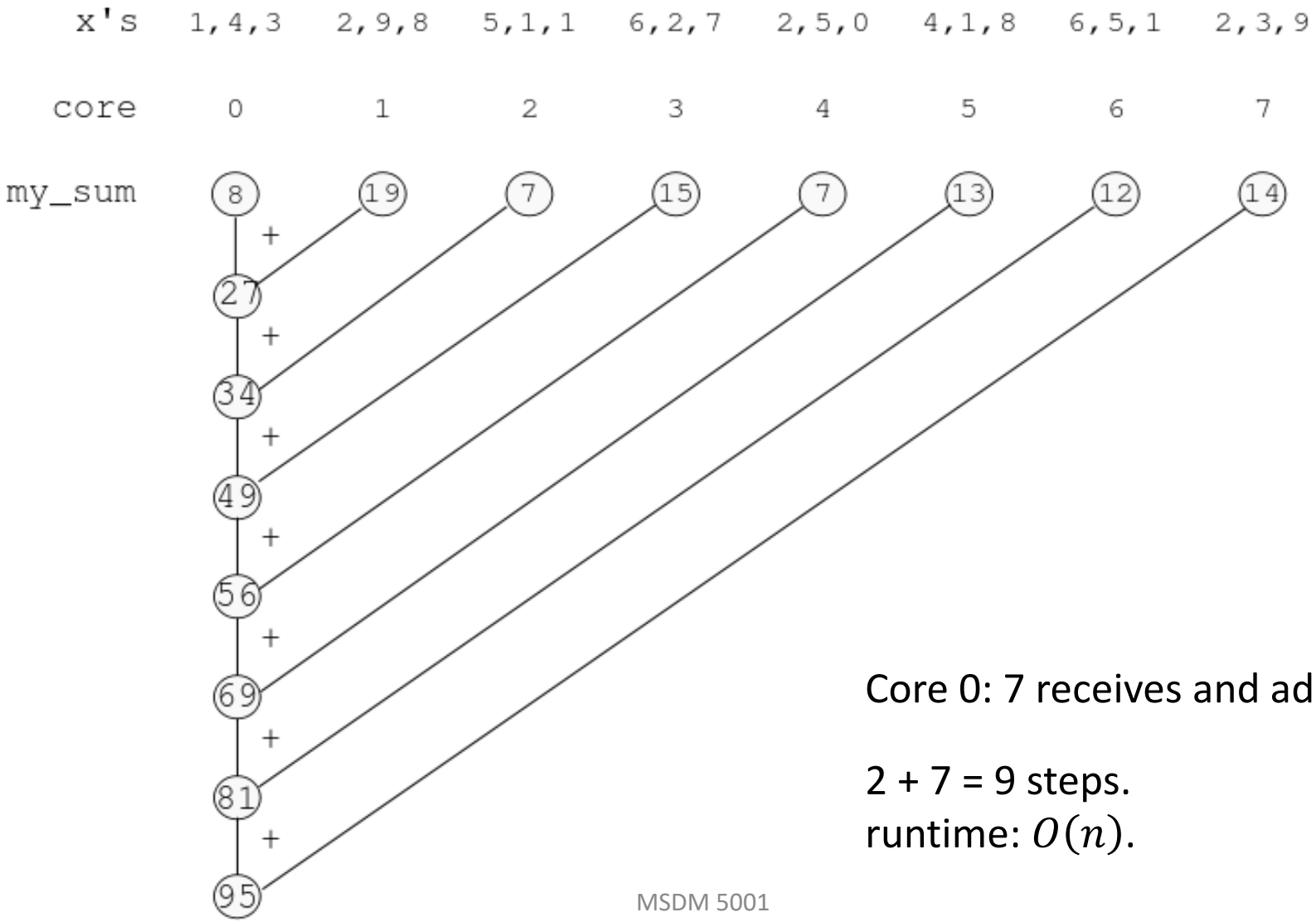
Initialize target variable
For loop - *i* sweeps from 1 to n
Fetch the *i*th data
Add it to the result
End the loop

24 steps.
runtime: $O(n)$.

Definition of the big *O* notation:
Let's say $f(x) = O(g(x))$ as $x \to \infty$ if there exist a positive real number M and a real number x_0 such that $|f(x)| \leq Mg(x)$ for all $x \geq x_0$.

Example: Global Sum – parallel method 1

Parallel Code:
P = 8, n = 24

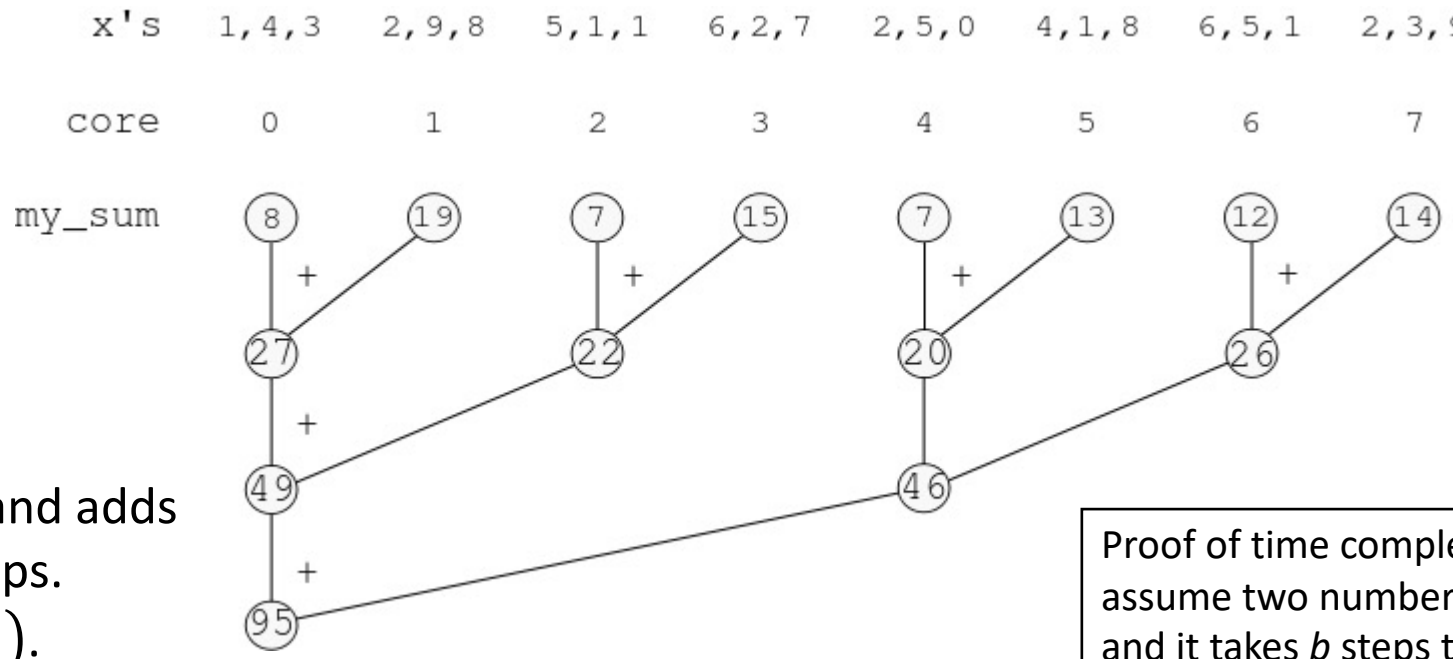


Core 0: 7 receives and adds

2 + 7 = 9 steps.
runtime: $O(n)$.

Example: Global Sum – parallel method 2

Parallel Code:
P = 8, n = 24



- Core 0: 3 receives and adds
- 2 + 1 + 1 + 1 = 5 steps.
- Runtime: $O\big(\log(n)\big)$.

- Compared to parallel method 1, it is improved by more than a factor of 2.
- With P = 1024 and n = 3072:
- Method 1: 1023 receives & adds;
- Method 2: 10 receives & adds.

Proof of time complexity: For simplicity, let's assume two numbers are added at each step and it takes *b* steps to reach the answer. Thus $\frac{n}{2^b} \cong 1$, we have $b \cong \log_2(n)$.

Q: when n is sufficiently large, which one is smaller, $n^{0.001}$ or log(n)?
A: log(n).

The Takeaway from the Global Sum Example

- Parallel computing often-times requires a particular “parallel algorithm” that is not straightforward to implement.
- A main core/processor is usually needed to dispatch, collect the data, and process the results.
- *Communication costs* - more processors do not necessarily guarantee a faster computation.

Example: Matrix Vector Multiplication

- $y = Ax$ or $y_i = A_{ij}x_j$
- Pseudo code with parallelization

```
Mat_Vect(A,x)
// n = number of rows
Let y be a new vector of length n
parallel for i = 1 to n
            y[i] = 0
parallel for i = 1 to n
   for j = 1 to n
      y[i] = y[i] + a[i][j]*x[j]
Return y
```

Einstein notation or *Einstein summation convention*: when an index variable appears twice in a single term and is not otherwise defined, it implies summation of that term over all the values of the index. This index is usually called dummy index. e.g., $a_i b_i \triangleq \sum_i a_i b_i$.

Key word:
Parallel: iteration in loop run concurrently.

$T_1 = O(n^2), T_n = O(n).$
Speedup $S = \frac{T_1}{T_n} = O(n).$

Q: can we further parallelize it?
A: Yes.

Example: Matrix Multiplication

- $C = AB$ or $C_{ij} = A_{ik}B_{kj}$.
- $n = 1$: $C_{11} = A_{11}B_{11}$.
- $n = 2$: $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$;
- $C = AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$.
- $n \geq 2$: $C = AB = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{bmatrix}\begin{bmatrix} B_{11} & \cdots & B_{1n} \\ \vdots & \ddots & \vdots \\ B_{n1} & \cdots & B_{nn} \end{bmatrix} = \begin{bmatrix} A_{1j}B_{j1} & \cdots & A_{1j}B_{jn} \\ \vdots & \ddots & \vdots \\ A_{nj}B_{j1} & \cdots & A_{nj}B_{jn} \end{bmatrix}$.

Example: Matrix Multiplication (Continued)

- Pseudo code:

MULT (C, A, B, n):
  create tempory matrix $T_{n\times n}$
  if $n = 1$
  then $C_{11} = A_{11} \times B_{11}$
  else
    // partition matrix, $O(1)$ *time*
    spawn Mult$(C_{11}, A_{11}, B_{11}, n/2)$
    spawn Mult$(C_{12}, A_{11}, B_{12}, n/2)$
    spawn Mult$(C_{21}, A_{21}, B_{11}, n/2)$
    spawn Mult$(C_{22}, A_{21}, B_{12}, n/2)$
    spawn Mult$(T_{11}, A_{12}, B_{21}, n/2)$
    spawn Mult$(T_{12}, A_{12}, B_{22}, n/2)$
    spawn Mult$(T_{21}, A_{22}, B_{21}, n/2)$
    spawn Mult$(T_{22}, A_{22}, B_{22}, n/2)$
    sync
    ADD(C, T, n)

$$C = AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

Key words:
Spawn: run subroutine at the same time as parent.
Sync: wait until all subroutines are done.

ADD(D, A, B, n)
  if n=1
  then $D_{11} = A_{11} + B_{11}$
  else
    // partition matrix
    spawn ADD$(D_{11}, A_{11}, B_{11}, n/2)$
    spawn ADD$(D_{12}, A_{12}, B_{12}, n/2)$
    spawn ADD$(D_{21}, A_{21}, B_{21}, n/2)$
    spawn ADD$(D_{22}, A_{22}, B_{22}, n/2)$
    sync

We will compute its time complexity later.

Brent's Theorem

- Assumption: a parallel computer where each processor can perform an arithmetic operation in unit time. Further, assume that the computer has exactly enough processors to exploit the maximum concurrency in an algorithm with *N* operations, such that T_∞ time steps suffice.
- *Brent's Theorem* says that a similar computer with fewer number of processors, *P*, can perform the algorithm in time:

$$T_P \leq T_\infty + \frac{(T_1 - T_\infty)}{P}.$$

- Brent's Theorem sets the upper bound of or the worst runtime of a parallel program given the resource and help estimate the performance or the efficiency of such parallel program.

Brent's Theorem Proof

- Say the work has *N* sequential steps, the j_{th} of which has n_j tasks that can be parallelized. Thus, it will take a runtime of $\lceil n_j/P \rceil$ for *P* processors.

$$T_P \leq \sum_j \left\lceil \frac{n_j}{P} \right\rceil \leq \sum_j \frac{n_j + P - 1}{P} = \frac{1}{P}\sum_j n_j + \sum_j \frac{P-1}{P} = \frac{T_1}{P} + \frac{P-1}{P} T_\infty$$
$$= T_\infty + \frac{(T_1 - T_\infty)}{P}.$$

Floor: $\lfloor x \rfloor = \max\{m \in \mathbb{Z} | m \leq x\}$.
Ceiling: $\lceil x \rceil = \min\{n \in \mathbb{Z} | n \geq x\}$.

$n_j = 5$, P = 1, the runtime is $n_j/P = 5$.
$n_j = 5$, P = 5, the runtime is $n_j/P = 1$.
$n_j = 5$, P = 2, the runtime is 1 + 1 + 1 = 3 = $\lceil 5/2 \rceil$.
(2) (2) (1)

x	Floor ⌊x⌋	Ceiling ⌈x⌉	Fractional part {x}
2	2	2	0
2.4	2	3	0.4
2.9	2	3	0.9
−2.7	−3	−2	0.3
−2	−2	−2	0

Brent's Theorem (Continued)

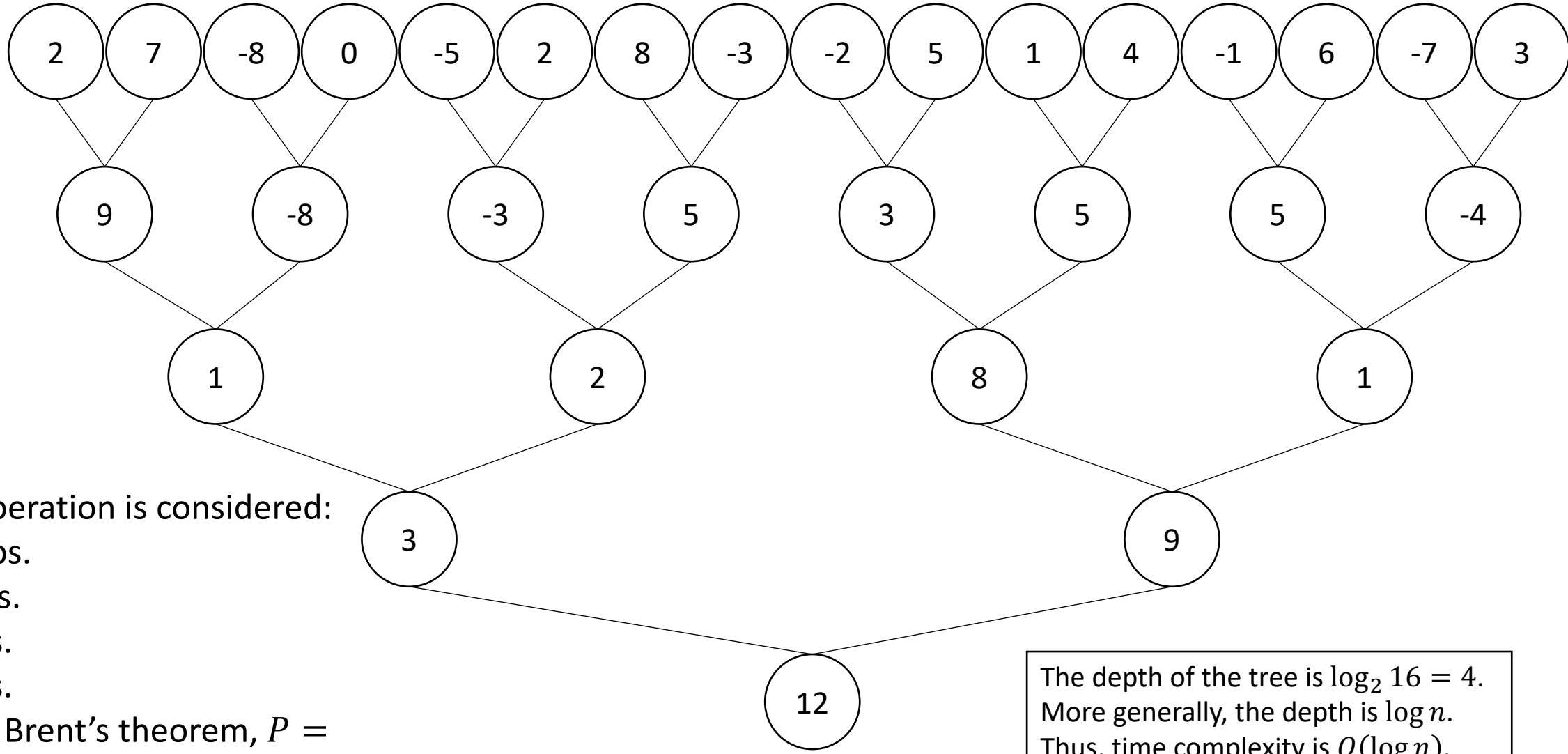
- Lower and upper bound of the runtime with *P* processors:

$$T_1/P \leq T_P \leq T_\infty + \frac{(T_1 - T_\infty)}{P} \leq \frac{T_1}{P} + T_\infty.$$

- Application: in parallel program of global sum for n numbers, the best runtime is $O(\log(n))$, which needs $n/2$ processors; One can reduce the number of processors to $O(n/\log(n))$ to retain the speedup while saving computational resources.
- Reason: $T_P \leq \frac{T_1}{P} + T_\infty = \frac{O(n)}{P} + O\big(\log(n)\big)$. If $P = n/\log(n)$, $T_P \leq O\big(\log(n)\big)$.
- Rationale: in the later stage of the parallel calculation, a smaller number of processors is needed.

Application of Brent's Theorem: Global Sum

P = 8, n = 16



If addition operation is considered:

P = 1: 15 steps.

P = 2: 7 steps.

P = 4: 5 steps.

P = 8: 4 steps.

According to Brent's theorem, $P = \frac{n}{\log(n)} = 4$ is adequate.

The depth of the tree is $\log_2 16 = 4$.
More generally, the depth is $\log n$.
Thus, time complexity is $O(\log n)$.

Amdahl's Law

- The speedup of a parallelized computation is a function of its parallelizable portion.
- The Amdahl's law reads $S = \frac{1}{(1-p)+p/s}$,

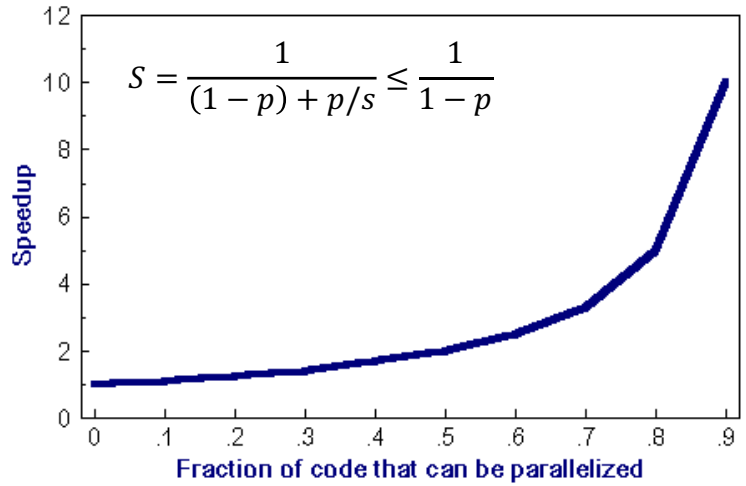
where *s* is the speedup of the parallelizable portion *p* of the job.

- Q: what is the definition of speedup?
  - A: $S = T(1)/T(P)$, see P.15.
- Proof:

$$T(s) = (1-p)T_1 + \frac{pT_1}{s}.$$

$$S = \frac{T_1}{T(s)} = \frac{1}{(1-p) + \frac{p}{s}}.$$

- This sets the maximum speedup of the parallel computation.
- Q: why we consider Amdahl's law?
  - A: Because a program usually consists of parallelizable and unparallelizable parts; overhead is important.

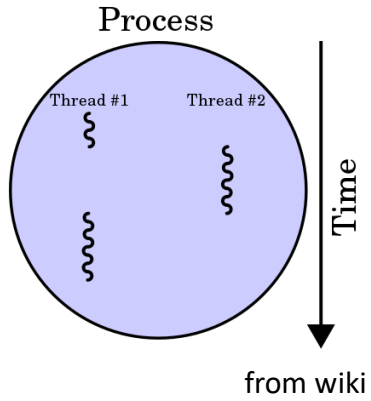


	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Parallel Computing Models & Platforms

- Shared memory
  - Threads
  - Multiprocessing
- Distributed memory
  - Message Passing
- MIMD
  - OpenCL, CUDA for GPGPU

Shared Memory - Threads Model



from wiki

- In the threads model, a single process can have multiple, concurrent execution paths.
- Implementation*:* **OpenMP**
  - Compiler directive based; can use serial code.
  - Portable / multi-platform, including Unix and Windows NT platforms.
  - Available in C/C++ and Fortran implementations.

A *thread* of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

Threads are a way for a program to split itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from one operating system to another, but in general, the way that a thread is created and shares its resources is different from the way a process does. - wiki.

Shared Memory - Multiprocessing

- *Multiprocessing* refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system are broken into smaller routines that run independently. The operating system allocates these threads to the processors improving performance of the system.
- A multiprocessing system can have multiple CPUs or multiple cores.
- In Python, the multiprocessing module includes a very simple and intuitive API for dividing work between multiple processes.
- Python with multiprocessing will be used as our Parallel Computing examples.

An *application programming interface* (API) is a computing interface which defines interactions between multiple software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, etc. It can also provide extension mechanisms so that users can extend existing functionality in various ways and to varying degrees. Through information hiding, APIs enable modular programming, which allows users to use the interface independently of the implementation.

Distributed Memory - Message Passing Model

- Feature of Message Passing Model:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.
- Implementation: **MPI**, i.e., *Message Passing Interface*, which is now the *de facto* industry standard for message passing, replacing virtually all other message passing implementations used for production work.
- MPI is NOT a new programming language
  - It defines a *library* of functions/subroutines that can be called from C, C++, and Fortran programs.
- MPI requires a wrapper script during compilation.
- Single Instruction Multiple Data (SIMD)
  - We compile a single program.
  - A single program/instruction is written so that different processes carry out different actions.

MIMD - GPU Computing

- General-purpose computing on graphics processing units (GPGPU, rarely GPGP) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU).
- GPGPU computing is realized through *Compute Unified Device Architecture* (CUDA), a parallel computing platform and application programming interface model created by Nvidia.

Design of a Parallel Program

- **Automatic vs. Manual Parallelization**
- Understand the Problem and the Program
- **Partitioning**
- **Communications**
- **Synchronization**
- Data Dependencies
- **Load Balancing**
- **Granularity**
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Automatic versus Manual Parallelization

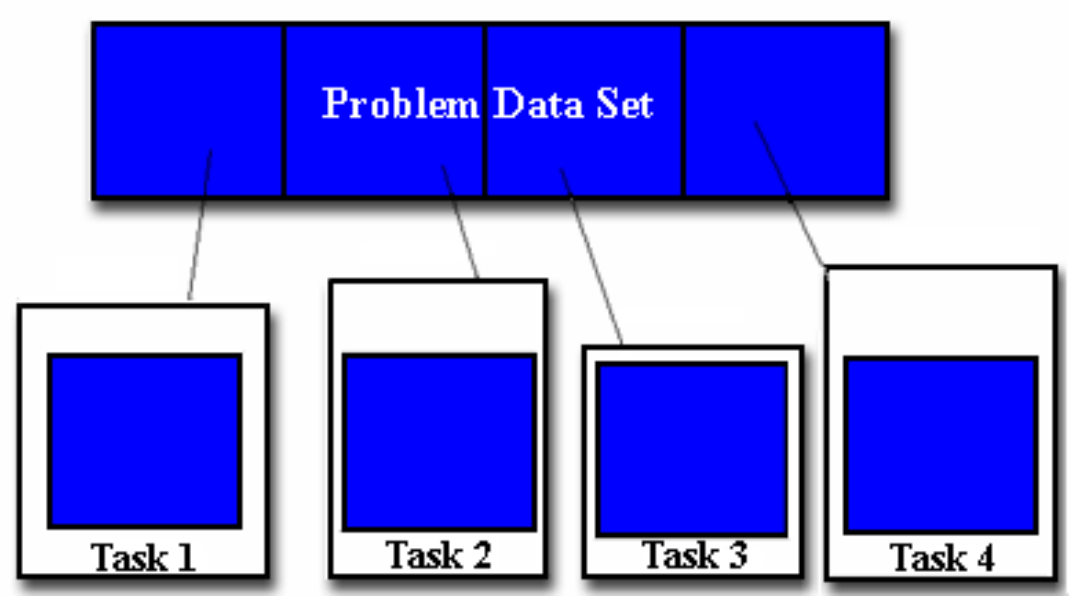
- Very often, manually developing parallel codes is a time consuming, complex, error-prone and iterative process.
- The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor.
- Ways the parallel compiler works:
  - fully automatic - Loops (do, for) loops are the most frequent target for automatic parallelization.
  - programmer directed by using "compiler directives”.
- Caveats for using automatic parallelization:
  - Wrong results, degraded performance, less flexibility…
  - Most automatic parallelization tools are for Fortran.

Design of a Parallel Program

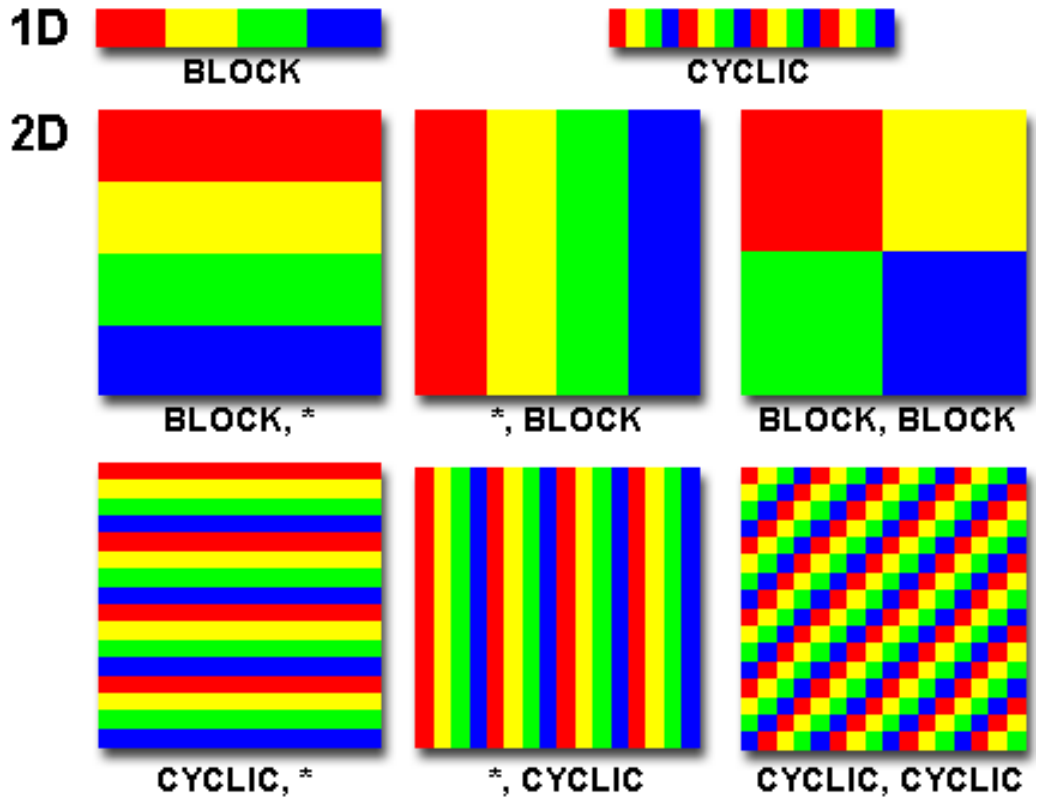
- **Automatic vs. Manual Parallelization**
- Understand the Problem and the Program
- **Partitioning**
- **Communications**
- **Synchronization**
- Data Dependencies
- **Load Balancing**
- **Granularity**
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Design of a Parallel Program - Partitioning

- Partition strategy: Domain decomposition.

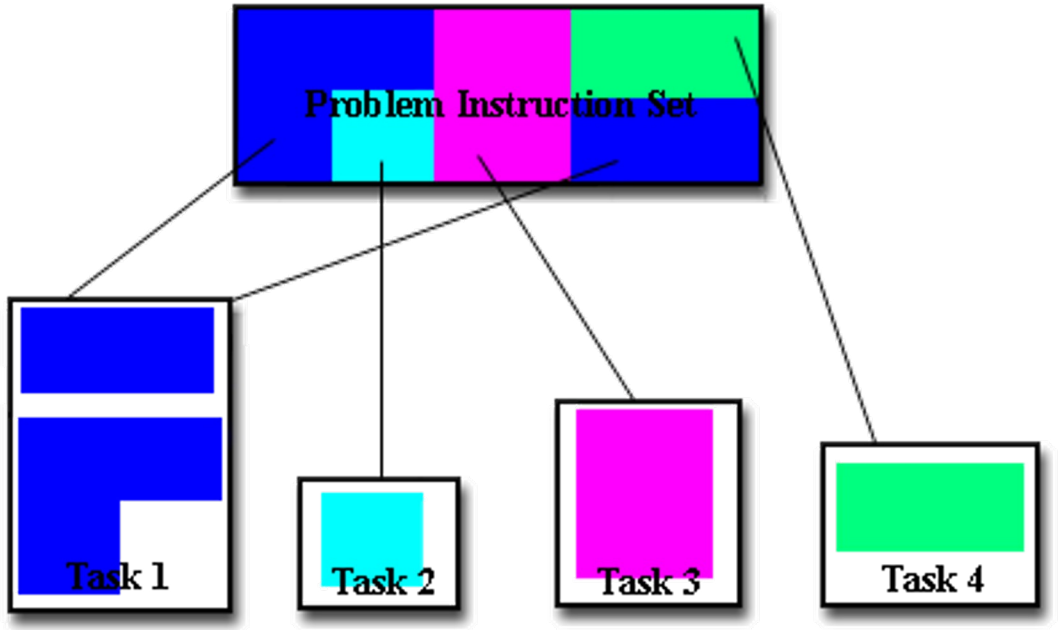


- Ways of data partition:

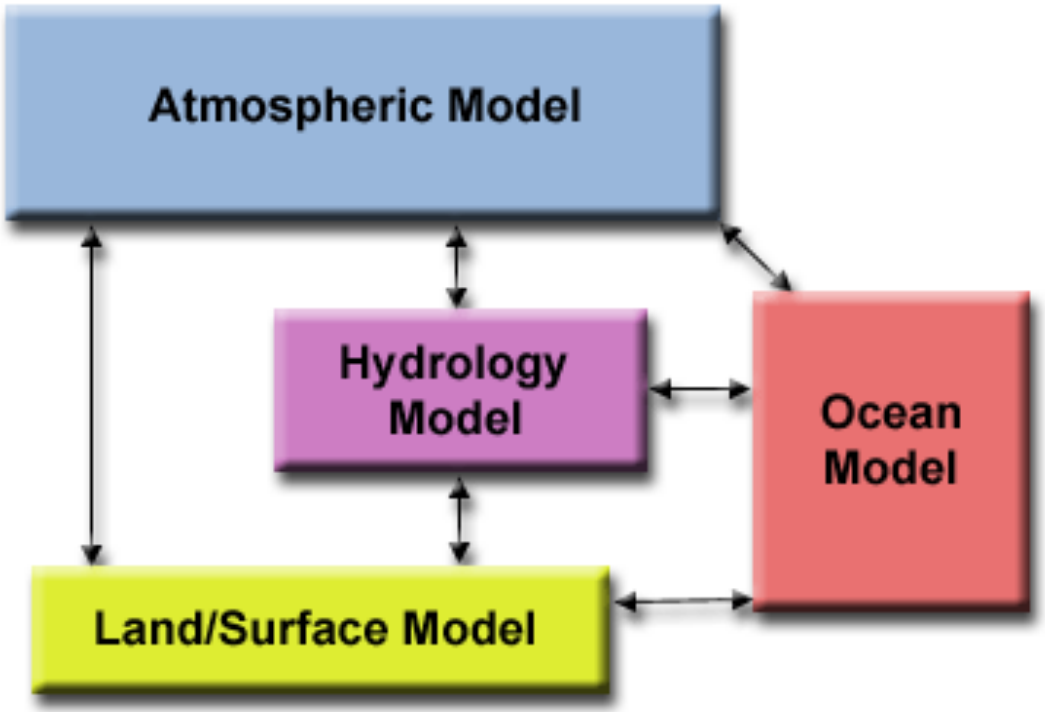


Design of a Parallel Program - Partitioning

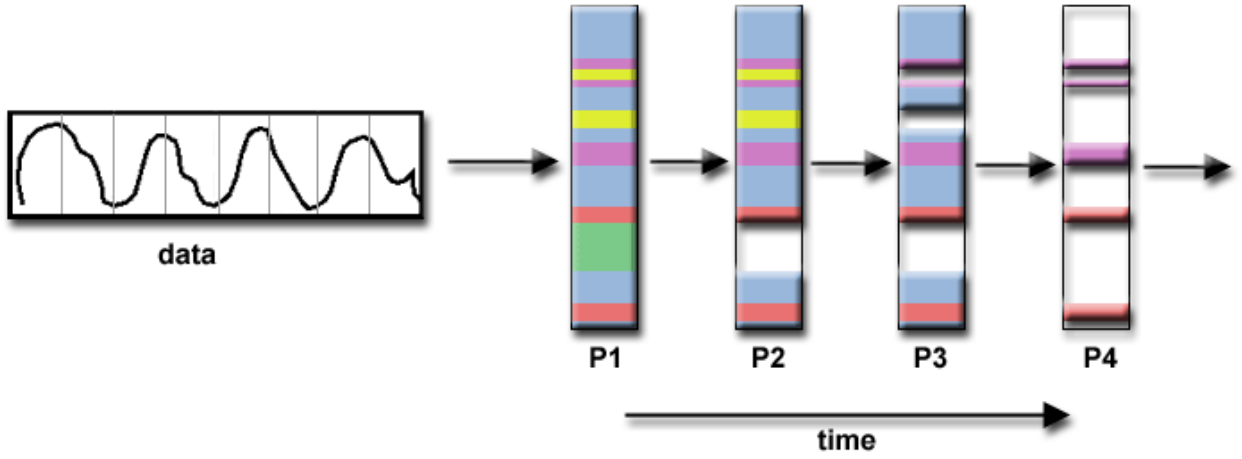
- Partition strategy: Functional decomposition. The focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
- Functional decomposition lends itself well to problems that can be split into different tasks. For example
  - Ecosystem Modeling;
  - Signal Processing;
  - Climate Modeling.



Partitioning - Examples of Functional Decomposition



Climate Modeling



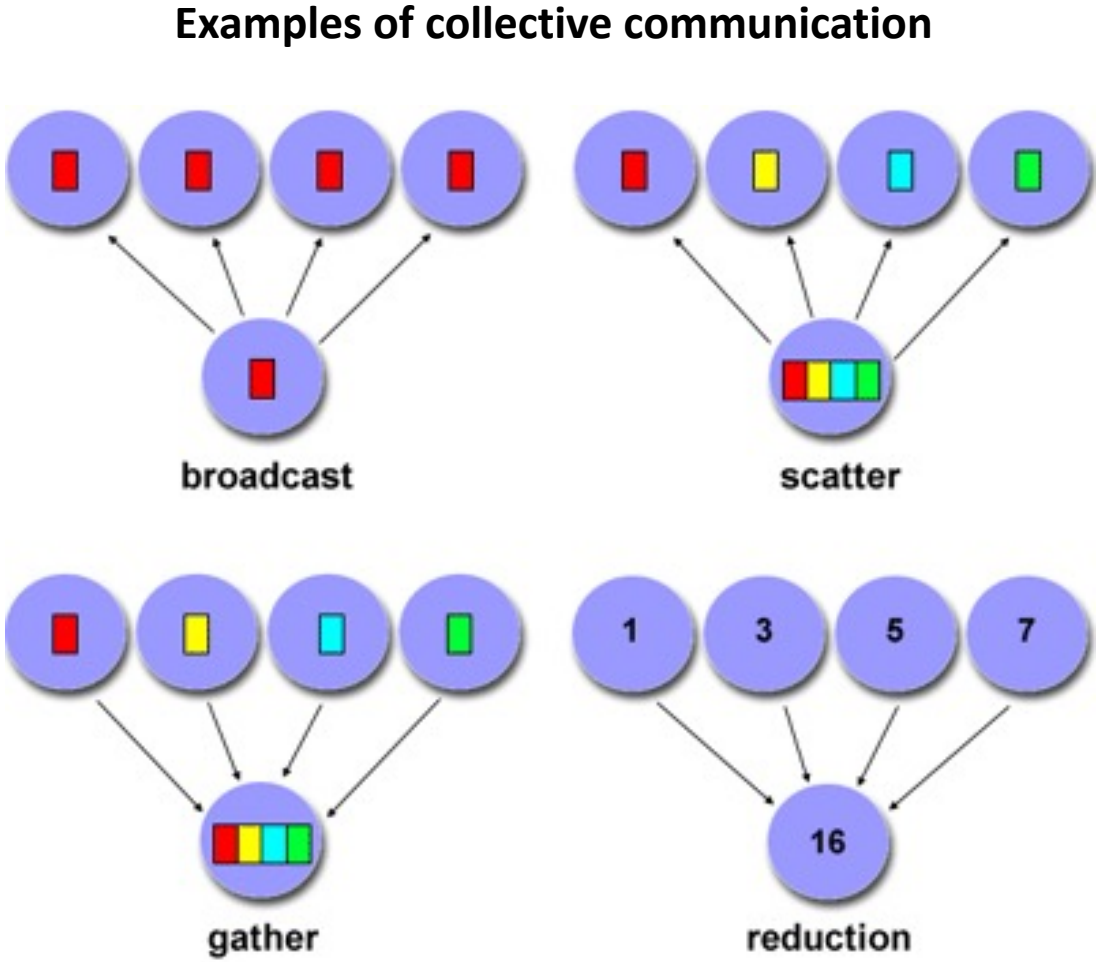
Signal Processing

Design of a Parallel Program

- **Automatic vs. Manual Parallelization**
- Understand the Problem and the Program
- **Partitioning**
- **Communications**
- **Synchronization**
- Data Dependencies
- **Load Balancing**
- **Granularity**
- I/O
- Limits and Costs of Parallel Programming
- **Performance Analysis** and Tuning

Communications

- Communication consumes computing resources, i.e., machine cycle, synchronization, etc.
- *Latency* - the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
- *Bandwidth* - the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec.
- Type of communication: point-to-point or collective.

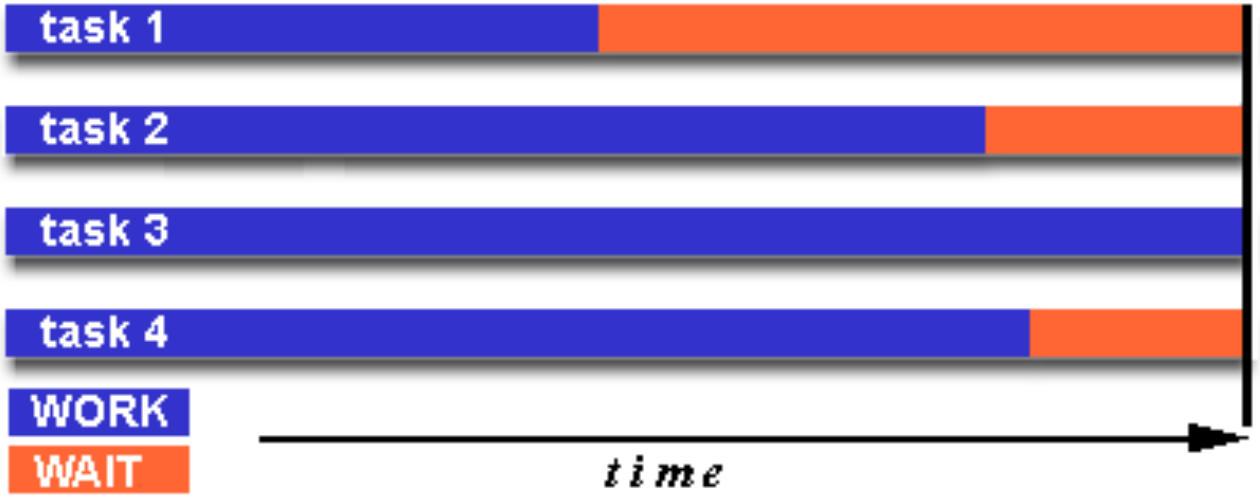


Synchronization

- Barrier
  - Usually implies that all tasks are involved.
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized.
- Lock/semaphore
  - Can involve any number of tasks.
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking.

Load Balancing

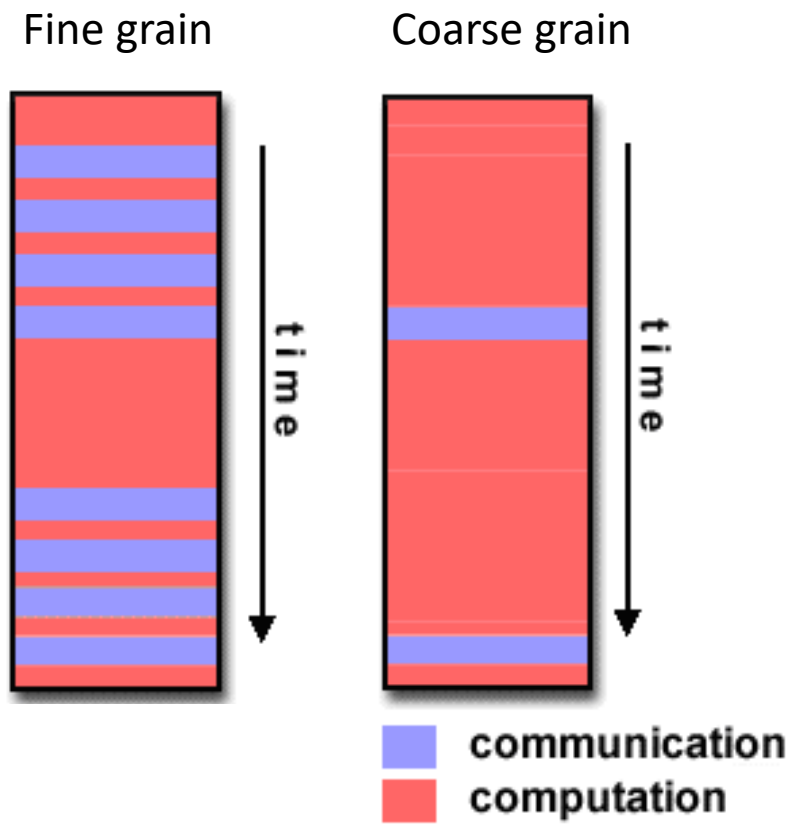
- *Load balancing* refers to the practice of distributing work among tasks so that all tasks are kept busy all the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance – *bottleneck effect*.



Strategy:
By programmer - equally partition the tasks;
Dynamic work assignment - create a task pool.

Granularity

- *Granularity* - Computation / Communication Ratio.
- Fine versus Coarse grain parallelism:
  - In most cases the overhead associated with communications and synchronization is high relative to execution speed, so it is advantageous to have coarse granularity.
  - However, fine-grain parallelism can help reduce overheads due to load imbalance.



Efficiency and Performance Analysis

- Best case: equally divide work among cores, while introducing no additional work

$$T_{\text{parallel}} = T_{\text{serial}}/P$$

- Transmitting data over the network produces overhead which is not present in the serial code
  - This is usually much slower than local memory access.
- Speedup: $S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$. Efficiency: $E = \frac{S}{P}$.
- Because $S \leq P$, we have $E = S/P \leq 1$.

P	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E	1.0	0.95	0.90	0.81	0.68

Scalability

- The *scalability* of a parallel program is a measure of its capacity to effectively utilize an increasing number of processors. ... For a fixed problem size, it may be used to determine the optimal number of processors to be used and the maximum possible speedup that can be obtained.
- Strong scalability: If we keep the efficiency fixed *without* increasing the problem size when we increase the number of processes. It measures how well the program performs when the workload remains constant, but more computational resources are added.
- Weak scalability: If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes.
- Hardware factors play a significant role in scalability:
  - Memory-CPU bus bandwidth on an SMP machine;
  - Communications network bandwidth;
  - Amount of memory available on any given machine or set of machines;
  - Processor clock speed.

Scalability (Continued)

- The scalability is also impacted by the following factors:
  - Load Balance.
  - Communication Overhead.
  - Granularity.
  - Scalable Algorithms.
  - Amdahl's Law: $S = \frac{1}{(1-p)+p/s}$. Problems that increase the percentage of parallel time with their size are more scalable than problems with a fixed percentage of parallel time.
- Q: For the following efficiency tables, are they weak or strong scalability?
  - A: Left: unknown; Right: weak scalability.

P	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E	1.0	0.95	0.90	0.81	0.68

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Time Complexity of Matrix Multiplication

$$T(n) = aT(n/b) + O(1)$$

- $C = AB$ or $C_{ij} = A_{ik}B_{kj}$.

MULT (C, A, B, n):
create tempory matrix $T_{n\times n}$
if $n = 1$
then $C_{11} = A_{11} \times B_{11}$
else
partition matrix // $O(1)$ $time$
spawn Mult$(C_{11}, A_{11}, B_{11}, n/2)$
spawn Mult$(C_{12}, A_{11}, B_{12}, n/2)$
spawn Mult$(C_{21}, A_{21}, B_{11}, n/2)$
spawn Mult$(C_{22}, A_{21}, B_{12}, n/2)$
spawn Mult$(T_{11}, A_{12}, B_{21}, n/2)$
spawn Mult$(T_{12}, A_{12}, B_{22}, n/2)$
spawn Mult$(T_{21}, A_{22}, B_{21}, n/2)$
spawn Mult$(T_{22}, A_{22}, B_{22}, n/2)$
sync
ADD(C, T, n)

Q: how much are a & b for matrix multiplication in serial & parallel computations?

$$A_1(n) = 4\,A_1\left(\frac{n}{2}\right) + O(1)$$
$$M_1(n) = 8M_1\left(\frac{n}{2}\right) + A_1(n)$$
$$A_\infty(n) = A_\infty\left(\frac{n}{2}\right) + O(1)$$
$$M_\infty(n) = M_\infty\left(\frac{n}{2}\right) + A_\infty(n)$$

ADD(D, A, B, n)
if n=1
then $D_{11} = A_{11} + B_{11}$
else
partition matrix
spawn ADD$(D_{11}, A_{11}, B_{11}, n/2)$
spawn ADD$(D_{12}, A_{12}, B_{12}, n/2)$
spawn ADD$(D_{21}, A_{21}, B_{21}, n/2)$
spawn ADD$(D_{22}, A_{22}, B_{22}, n/2)$
sync

Time Complexity of Matrix Multiplication (continued)

$$
\begin{aligned}
T(n) &= aT(n/b) + O(1) && (1)\\
&= a(aT(n/b^2) + O(1)) + O(1) && (2)\\
&= a^2T(n/b^2) + O(1) + O(1) && (3)\\
&\vdots && (4)\\
&= a^kT(n/b^k) + \overbrace{O(1) + O(1) + \ldots + O(1)}^{k} && (5)
\end{aligned}
$$

$T(1) = 1$

If $b^k = n$, i. e. , $k = \log_b n$, then we have

$$T(n) = O\left(a^{\log_b n} + \log_b n\right)$$
$$= O\left(n^{\log_b a} + \log_b n\right)$$

$$(\log_b a)(\log_b n) = (\log_b n)(\log_b a)$$
$$\Rightarrow b^{(\log_b a)(\log_b n)} = b^{(\log_b n)(\log_b a)}$$
$$\Rightarrow \left(b^{\log_b a}\right)^{\log_b n} = \left(b^{\log_b n}\right)^{\log_b a}$$
$$\Rightarrow a^{\log_b n} = n^{\log_b a}$$

Recurrence Relation (Master Theorem)

- More generally, we have the master theorem:
- for $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ and $f(n) = n^c$

$$T(n) = \begin{cases} O\left(n^{\log_b a}\right) & c < \log_b a \\ O(n^c) & c > \log_b a \end{cases}$$

$$A_1(n) = 4A_1\left(\frac{n}{2}\right) + O(1) = O(n^2)$$

$$\begin{aligned} M_1(n) &= 8M_1\left(\frac{n}{2}\right) + A_1(n) \\ &= 8M_1\left(\frac{n}{2}\right) + O(n^2) \\ &= O(n^3) \end{aligned}$$

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Practicing Parallel Programming

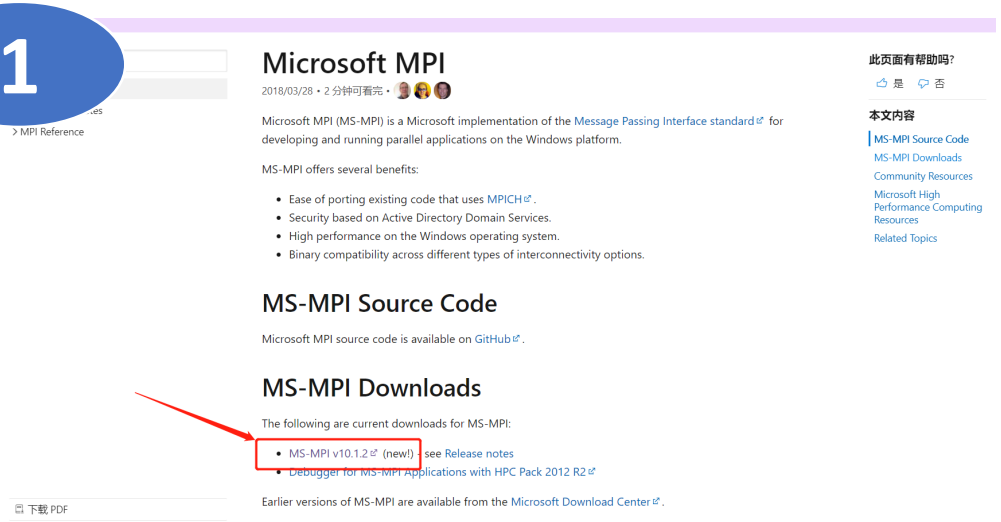
- Online python platform:
  - google: http://colab.research.google.com/
  - cocalc: https://cocalc.com/app?anonymous=jupyter
  - binder: https://mybinder.org/v2/gh/ipython/ipython-in-depth/master?filepath=binder/Index.ipynb

For Windows

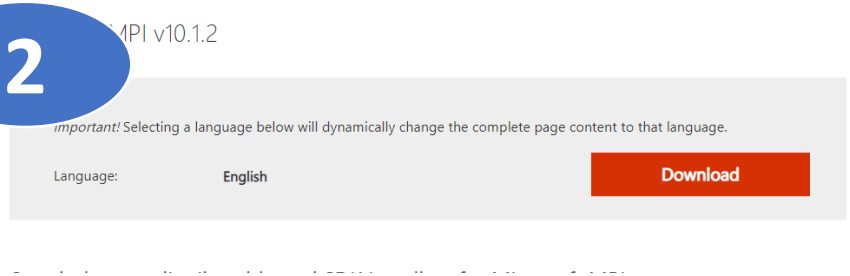
The guide for install Massage Passing Interface (MPI)

Install MPI for Windows

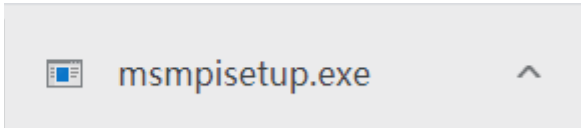
1



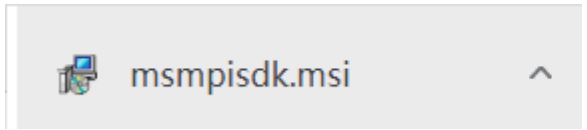
2



3



Install First



Install Later

From LIU, Ming

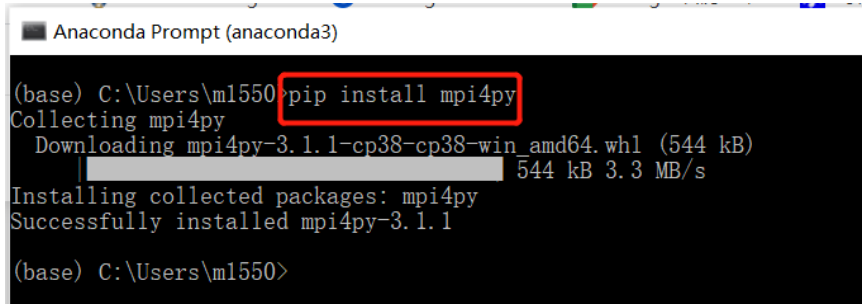
For Windows

Guidance for installing mpi4py module

Before

```
7
8  import mpi4py
9
```

```
    import mpi4py

ModuleNotFoundError: No module named 'mpi4py'
```



Open the anaconda (Powershell) Prompt(anaconda3)

pip install mpi4py (similar step to install other Packages)

The module will be automatically installed for you

After

```
10  import mpi4py
11  Code analysis
12
13  'mpi4py' imported but unused (pyflakes E)
14  sum=0
15  tep=rank
```


From LIU, Ming

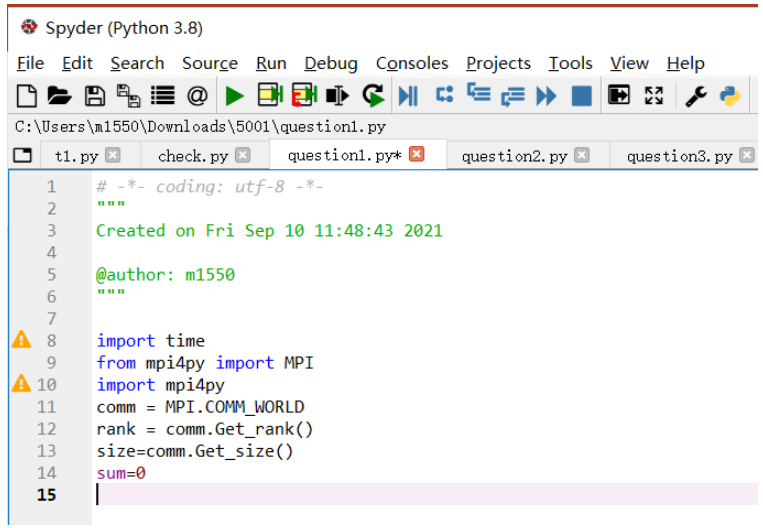
For Windows

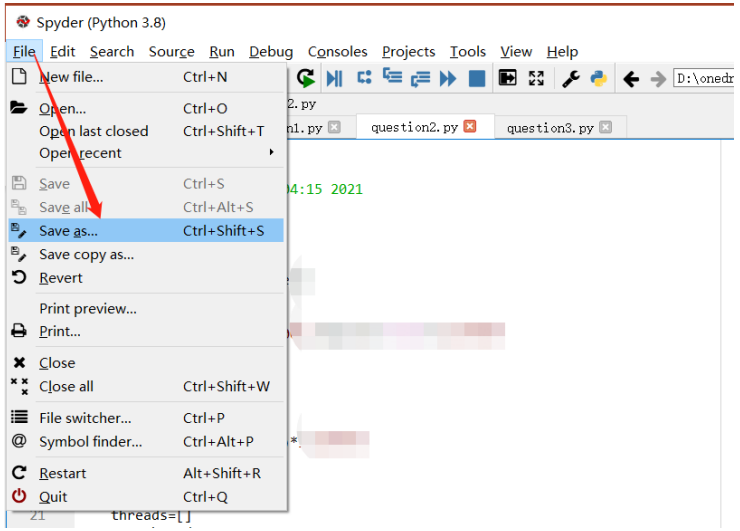
Run the file using MPI

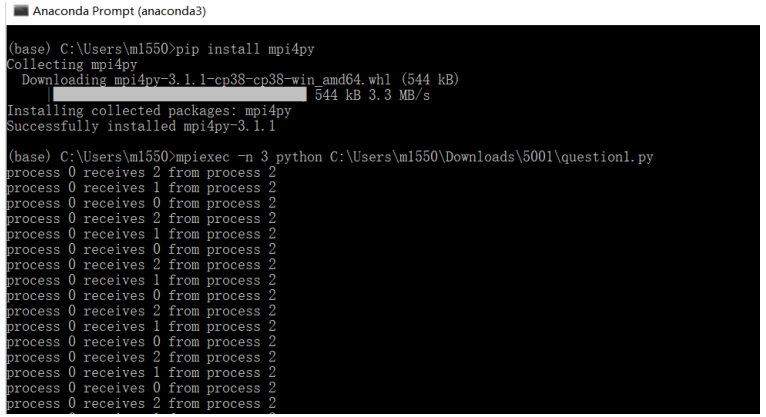
1 Write codes

2 Save as a directory (no space and all English characters)

3 mpiexec –n np python filepath (np: number of processors)







E.g., mpiexec –n 5 python C:\Users\m1550\Downloads\5001\question1.py

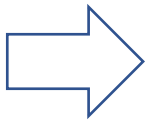
From LIU, Ming

For Windows

Run the file-without using MPI

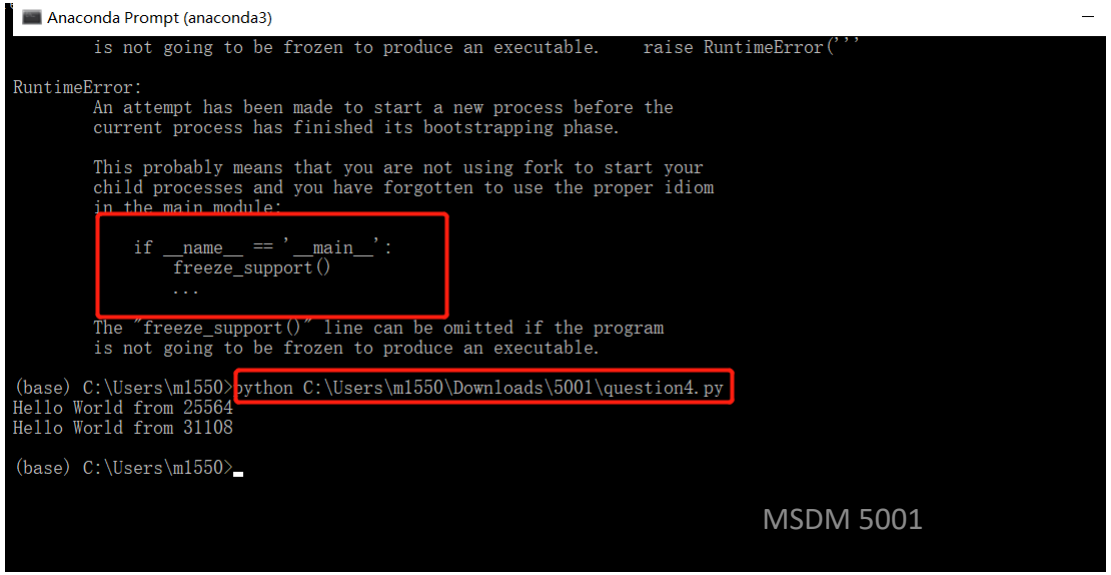
1 Put the code under if __name__ == "__main__"

```
rt multiprocessing
from multiprocessing import Process
import os
def do_something():
    print(f"Hello World from {os.getpid()}")

p1 = multiprocessing.Process(target=do_something)
p2 = multiprocessing.Process(target=do_something)
p1.start()
p2.start()
p1.join()
p2.join()
```



```
import multiprocessing
from multiprocessing import Process
import os
def do_something():
    print(f"Hello World from {os.getpid()}")
if __name__ == '__main__':
    p1 = multiprocessing.Process(target=do_something)
    p2 = multiprocessing.Process(target=do_something)
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

2 python filepath



e.g., python C:\Users\m1550\Downloads\5001\question4.py

From LIU, Ming

For Mac OSX Users to Install mpi4py

- Method 1: install anaconda; in command line, type

conda install mpi4py

- Method 2: use homebrew: https://formulae.brew.sh/formula/mpi4py

- Linux users should be able to use similar package manager to install mpi4py.

The First Parallel Code - *"Hello World"*

- Link: https://colab.research.google.com

```
import multiprocessing
from multiprocessing import Process
import os

def do_something():
  print(f"Hello World from {os.getpid()}")

p1 = multiprocessing.Process(target=do_something)
p2 = multiprocessing.Process(target=do_something)

p1.start()
p2.start()

p1.join()
p2.join()
```

Multiprocessing module allows the programmer to fully leverage multiple processors on a given machine.
Process objects represent activity that is run in a separate process.
start(): Start the process's activity.
join(): the method blocks until the process whose join() method is called terminates.

To run, hold *shift* and return.

A possible outcome:

Hello World from 609
Hello World from 610

The First Parallel Code - *"Hello World"* (Continued)

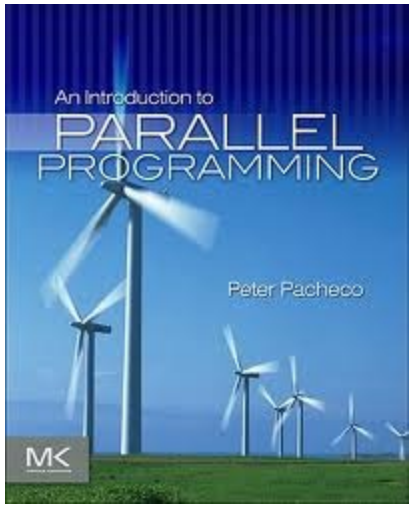
```
import multiprocessing
from multiprocessing import Process
import os

def do_something():
  print('Hello')
  print(f"World from {os.getpid()}")

p1 = multiprocessing.Process(target=do_something)
p2 = multiprocessing.Process(target=do_something)

p1.start()
p2.start()

p1.join()
p2.join()
```

Possible Outcome 1:
Hello
World from 609
Hello
World from 610

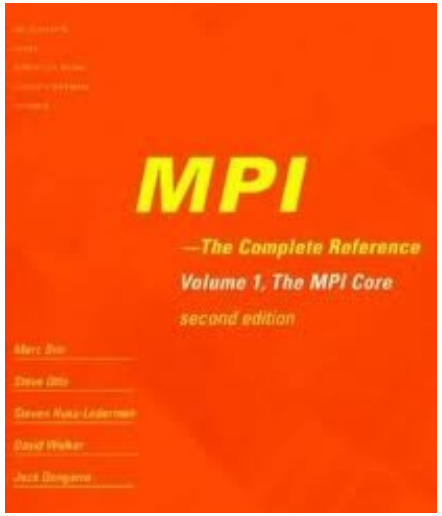
Possible Outcome 2:
Hello
Hello
World from 609
World from 610

Q: how to enforce a sequential execution of multiple processes?
A: exchange p2.start() and p1.join().

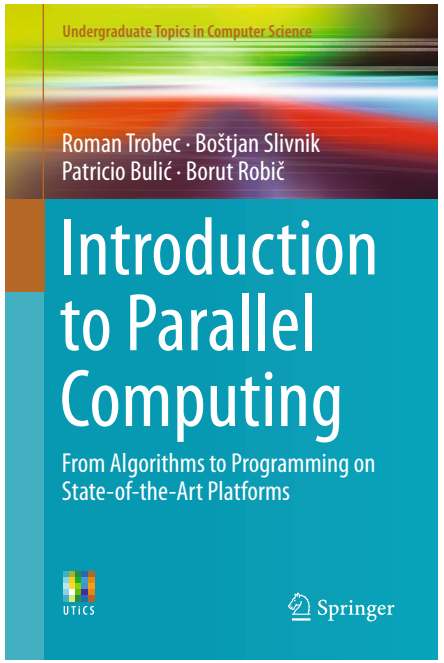
References



An Introduction to Parallel Programming -Pacheco



MPI: The Complete Reference -Snir, Otto, Huss-Lederman, Walker, Dongarra



Introduction to Parallel Computing -Trobec, Silvnik, Bulic and Robic

Summary of Today's Lecture

- Definition & History
- Architecture & Platform
- Time Complexity
- Analysis of Speedup & Efficiency
- Design Strategy
- MPI for Python installation guidance
- "*Hello world*" example

Future Parallel Computing Sections

- There is **no class** on next Saturday, Sep. 30th.
- Please bring your own laptop in the next two classes to enjoy the best learning experience.
- 5th class on Oct. 7th.
- 6th class on Oct. 14th.