Parallel Programming Principle and Practice

Lecture 6 —parallel algorithm design

Outline

- Speedup and overhead
  - Serial/Parallel time
  - Total parallel computation time
  - Overhead t_0
    - communication/synchronization
    - extra computation
    - parallel granularity
    - load balancing
    - memory hierarchy
  - Speedup

Speedup and overhead

Serial/Parallel time (串行/并行计算时间)

Serial/Parallel time

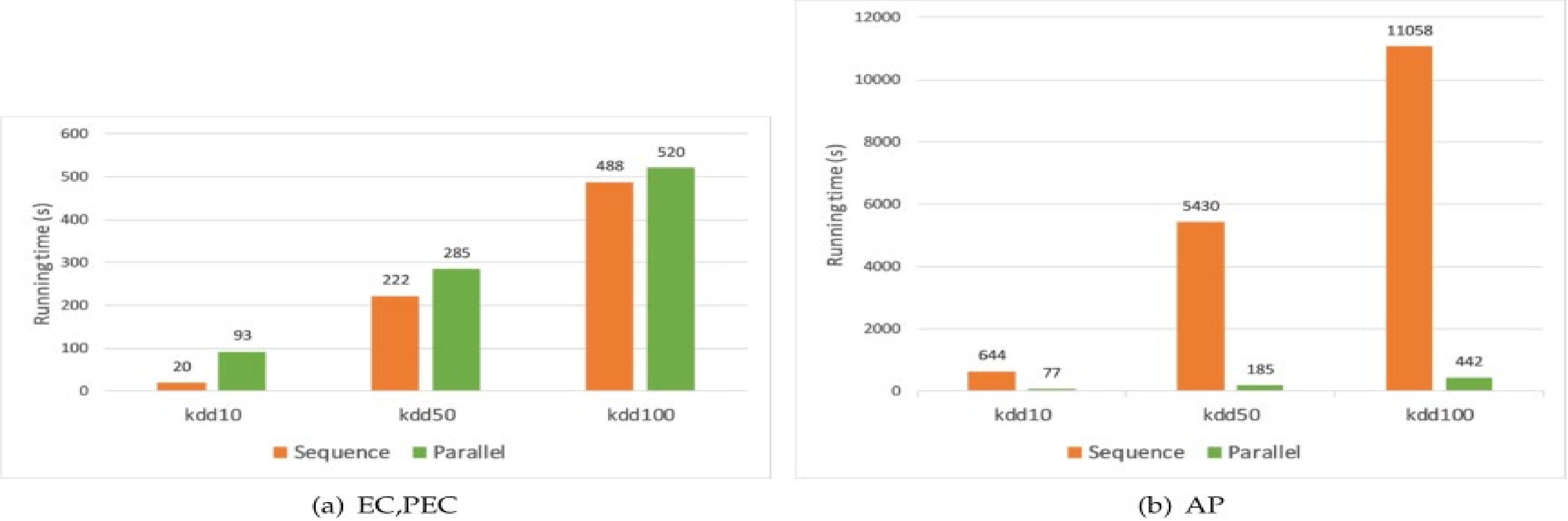


Figure 2. Compare execution time of each step between sequential and parallel algorithms:

(a) Step 1 of the sequential algorithm and EC, PEC steps of the parallel algorithm

(b) Step 2 of the sequential algorithm and AP step of the parallel algorithm

Cao T, Yamada K, Unehara M, et al. Parallel Computation of Rough Set Approximations in Information Systems with Missing Decision Data[J]. Computers, 2018, 7(3): 44.

Serial/Parallel time

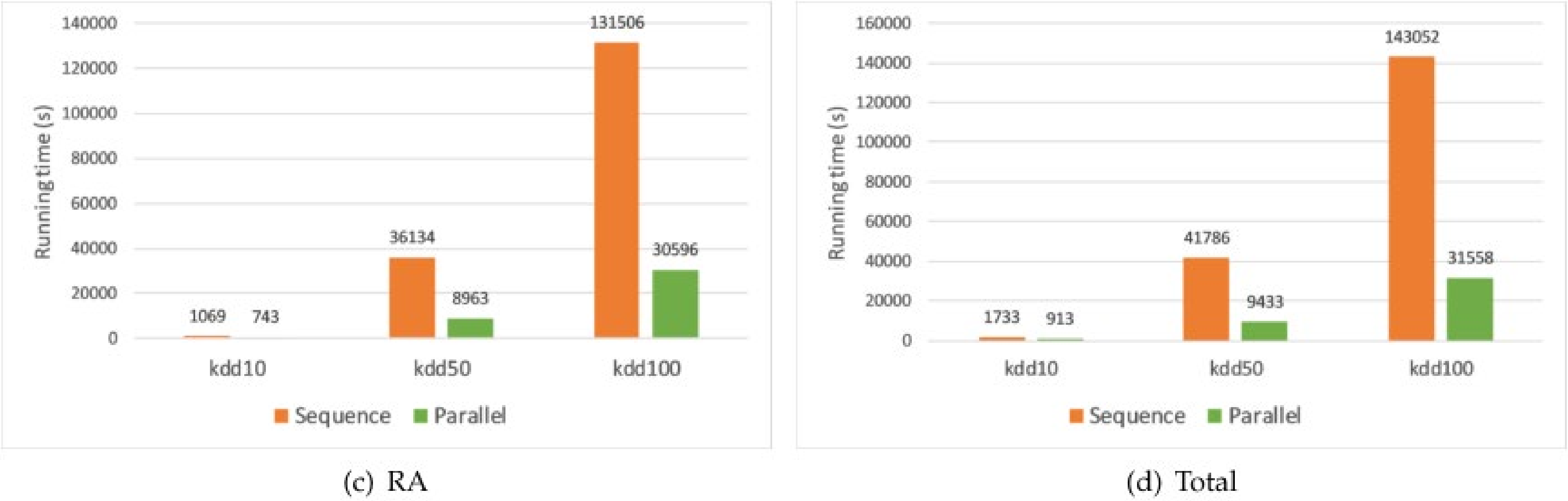


Figure 2. Compare execution time of each step between sequential and parallel algorithms:
(c) Steps 3,4 of the sequential algorithm and RA stepof the parallel algorithm
(d) total steps of the sequential and parallel algorithms

Cao T, Yamada K, Unehara M, et al. Parallel Computation of Rough Set Approximations in Information Systems with Missing Decision Data[J]. Computers, 2018, 7(3): 44.

華中科技大學

Serial/Parallel time

- **Serial execution time**
  - $T(n, 1) = \sigma(n) + \varphi(n)$
- Assume that the parallel portion of the computation that can be executed in parallel divides up perfectly among p processors
- **Parallel execution time**
  - $T(n, p) \geq \sigma(n) + \varphi(n)/p + \kappa(n, p)$

> - n problem size(问题大小)
> - p number of processors
> - $\sigma(n)$ inherently serial portion of computation
> - $\varphi(n)$ portion of parallelizable computation
> - $\kappa(n, p)$ parallelization overhead(并行开销)

Speedup and overhead

Total Parallel time (总并行计算时间)

Total Parallel time

Number of Processors Used

- The number of processors used is an important factor in analyzing the efficiency of a parallel algorithm
- The cost to buy, maintain, and run the computers are calculated
- Larger the number of processors used by an algorithm to solve a problem, more costly becomes the obtained result

Total cost of a parallel algorithm is the product of time complexity and the number of processors used in that particular algorithm.

- **Total Parallel time = Time complexity × Number of processors used**

Speedup and overhead

Overhead t_0(额外开销问题)

Overhead t_0(额外开销问题)

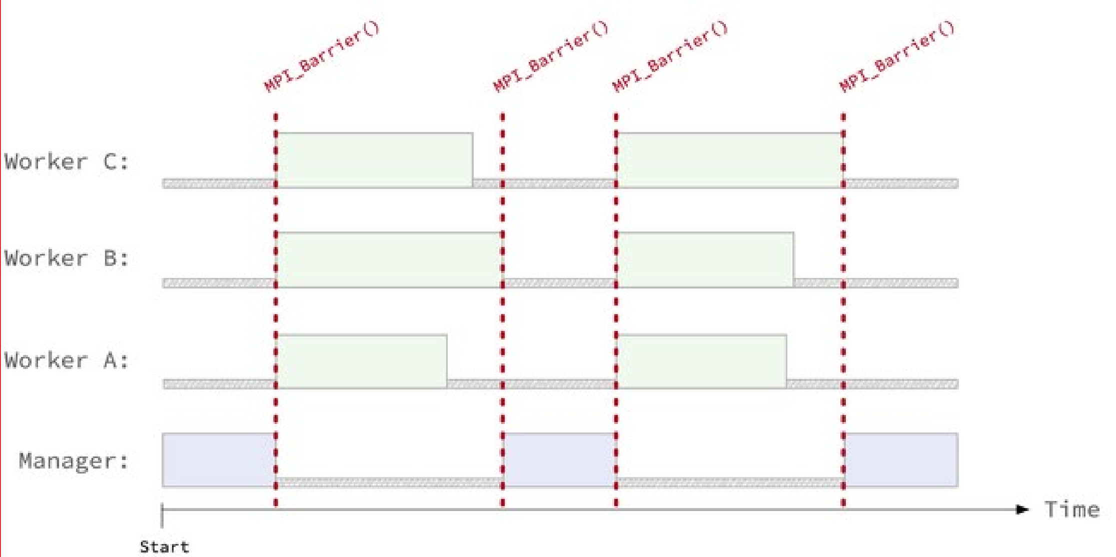
- communication/synchronization
- extra computation
- parallel granularity
- load balancing
- memory hierarchy

communication/synchronization

- The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently
  - Data must then be transferred between tasks so as to allow computation to proceed
- Communication/synchronization is then required to manage the data transfer and/or coordinate the execution of tasks
- Organizing this communication in an efficient manner can be challenging

Synchronization overhead example

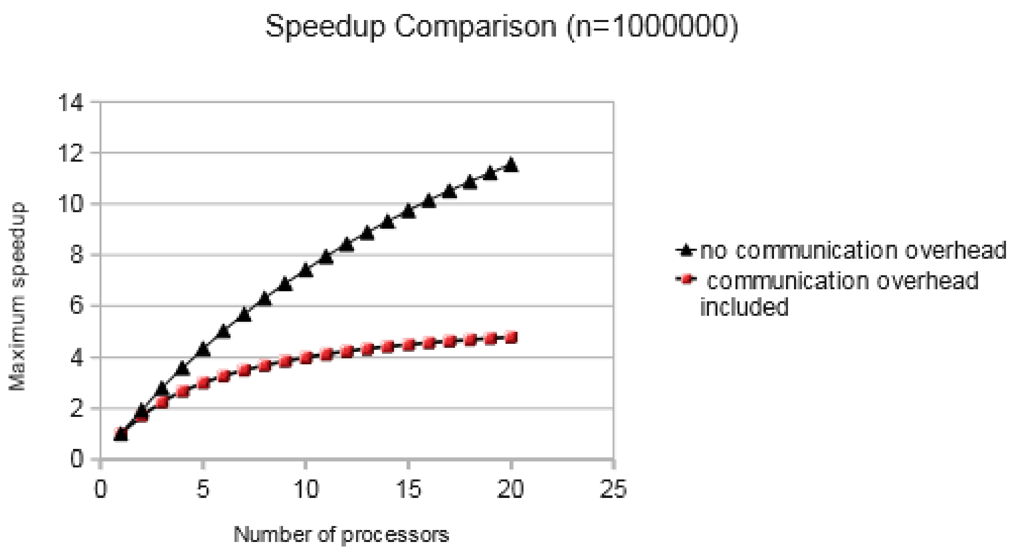
- 一个任务等待另一个任务的时间都被认为是同步开销
  - 任务可能在一个显式的障碍上同步：在更新共享数据和计算下一个时间步之前，它们都完成了一个模拟时间步的计算
  - 最慢的任务决定了整个计算的速度



- ✓ 三个worker和一个manager
- ✓ manager定期收集数据并重新分发给worker
- ✓ 一旦一个任务调用了MPI_Barrier()，它必须等待，直到所有其他工作调用MPI_Barrier()，它才能继续

Communication overhead example

- 在网络传输过程中，由于对信号的传输，需要变换数据格式，难免要加入一些冗余的数据，这些冗余数据又是传输所必须的，而这些冗余数据在源数据中占有的比例叫做开销



Extra computation(额外计算)

- Sometime the best sequential algorithm is not easily parallelizable and one is forced to
  - use a parallel algorithm based on a poorer but easily parallelizable sequential algorithm
- Sometimes repetitive work is done on each of the N processors instead of send/receive, which leads to extra computation

Parallel granularity

Parallel granularity Definition

- In parallel computing, granularity (or grain size) of a task is a measure of the amount of work (or computation) which is performed by that task
- Another definition of granularity takes into account the communication overhead between multiple processors or processing elements
  - It defines granularity as the ratio of computation time to communication time
  - computation time is the time required to perform the computation of a task
  - communication time is the time required to exchange data between processors
  - If T_{comp} is the computation time and T_{comm} denotes the communication time, then the Granularity G of a task can be calculated as

$$G = \frac{T_{\text{comp}}}{T_{\text{comm}}}$$

Parallel granularity

Types of parallelism

- Depending on the amount of work which is performed by a parallel task, parallelism can be classified into three categories
  - Fine-grained parallelism
  - Coarse-grained parallelism
  - Medium-grained parallelism

Fine-grained parallelism

- In fine-grained parallelism, a program is broken down to a large number of small tasks
  - These tasks are assigned individually to many processors
  - The amount of work associated with a parallel task is low and the work is evenly distributed among the processors
  - ✓ Hence, fine-grained parallelism facilitates load balancing
- As each task processes less data, the number of processors required to perform the complete processing is high
  - ✓ This in turn, increases the communication and synchronization overhead

Fine-grained parallelism

- Fine-grained parallelism is best exploited **in architectures which support fast communication**
  - ✓ Shared memory architecture which has a low communication overhead is most suitable for fine-grained parallelism
- It is difficult for programmers to detect parallelism in a program, therefore, it is usually the compilers' responsibility to detect fine-grained parallelism

Fine-grained parallelism

- An example of a fine-grained system (from outside the parallel computing domain) is **the system of neurons in our brain**
- Connection Machine (CM-2) and J-Machine are examples of fine-grain parallel computers that have grain size in the range of 4-5 μs.

Thinking Machines CM-2 at the Computer History Museum in Mountain View, California.

Coarse-grained parallelism

- In coarse-grained parallelism, a program is split into large tasks
  - Due to this, a large amount of computation takes place in processors
  - This might **result in load imbalance**, wherein certain tasks process the bulk of the data while others might be idle
  - Further, coarse-grained parallelism fails to exploit the parallelism in the program as most of the computation is performed sequentially on a processor.
- The advantage of this type of parallelism is **low communication and synchronization overhead**

Coarse-grained parallelism

- Message-passing architecture takes a long time to communicate data among processes which makes it suitable for coarse-grained parallelism
- **Cray Y-MP** is an example of coarse-grained parallel computer which has a grain size of about 20s

Cray Y-MP Model D at NASA Center for Computational Sciences, GSFC

Cray Y-MP M90 (Ziegler) at the US National Cryptologic Museum

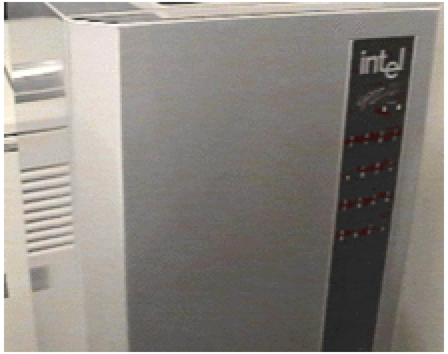
Medium-grained parallelism

- Medium-grained parallelism is used relatively to fine-grained and coarse-grained parallelism
- Medium-grained parallelism is a compromise between fine-grained and coarse-grained parallelism, where we have <u>task size and communication time greater than fine-grained parallelism and lower than coarse-grained parallelism</u>
- Most general-purpose parallel computers fall in this category

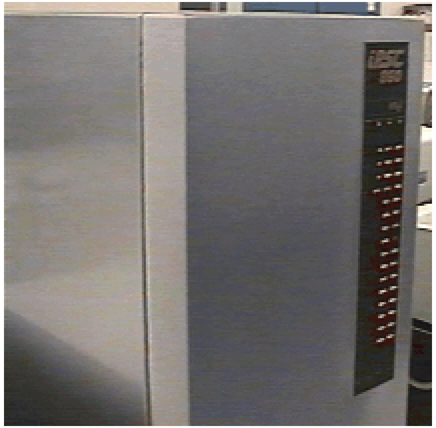
Medium-grained parallelism

- Intel iPSC is an example of medium-grained parallel computer which has a grain size of about 10ms

Intel iPSC-1 (1985) at Computer History Museum.



Intel iPSC/2 16-node parallel computer



Intel iPSC/860 32-node parallel computer front panel

Example

- Consider a 10*10 image that needs to be processed, given that, processing of the 100 pixels is independent of each other

<table>
<tr><th>Fine-grain : Pseudocode for 100 processors</th><th>Medium-grain : Pseudocode for 25 processors</th><th>Coarse-grain : Pseudocode for 2 processors</th></tr>
<tr><td><pre>
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute element 1; break;
    case 2: Compute element 2; break;
    case 3: Compute element 3; break;
    .
    .
    .
    .
    case 100: Compute element 100;
              break;
  }
}
</pre></td><td><pre>
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute elements 1-4; break;
    case 2: Compute elements 5-8; break;
    case 3: Compute elements 9-12; break;
    .
    .
    case 25: Compute elements 97-100;
             break;
  }
}
</pre></td><td><pre>
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute elements 1-50;
            break;
    case 2: Compute elements 51-100;
            break;
  }
}
</pre></td></tr>
<tr><td>Computation time - 1 clock cycle</td><td>Computation time - 4 clock cycles</td><td>Computation time - 50 clock cycles</td></tr>
</table>

Example

Fine-grain : Pseudocode for 100 processors

```
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute element 1; break;
    case 2: Compute element 2; break;
    case 3: Compute element 3; break;
    .
    .
    .
    .
    case 100: Compute element 100;
              break;
  }
}
```

Computation time - 1 clock cycle

Medium-grain : Pseudocode for 25 processors

```
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute elements 1-4; break;
    case 2: Compute elements 5-8; break;
    case 3: Compute elements 9-12; break;
    .
    .
    case 25: Compute elements 97-100;
             break;
  }
}
```

Computation time - 4 clock cycles

Coarse-grain : Pseudocode for 2 processors

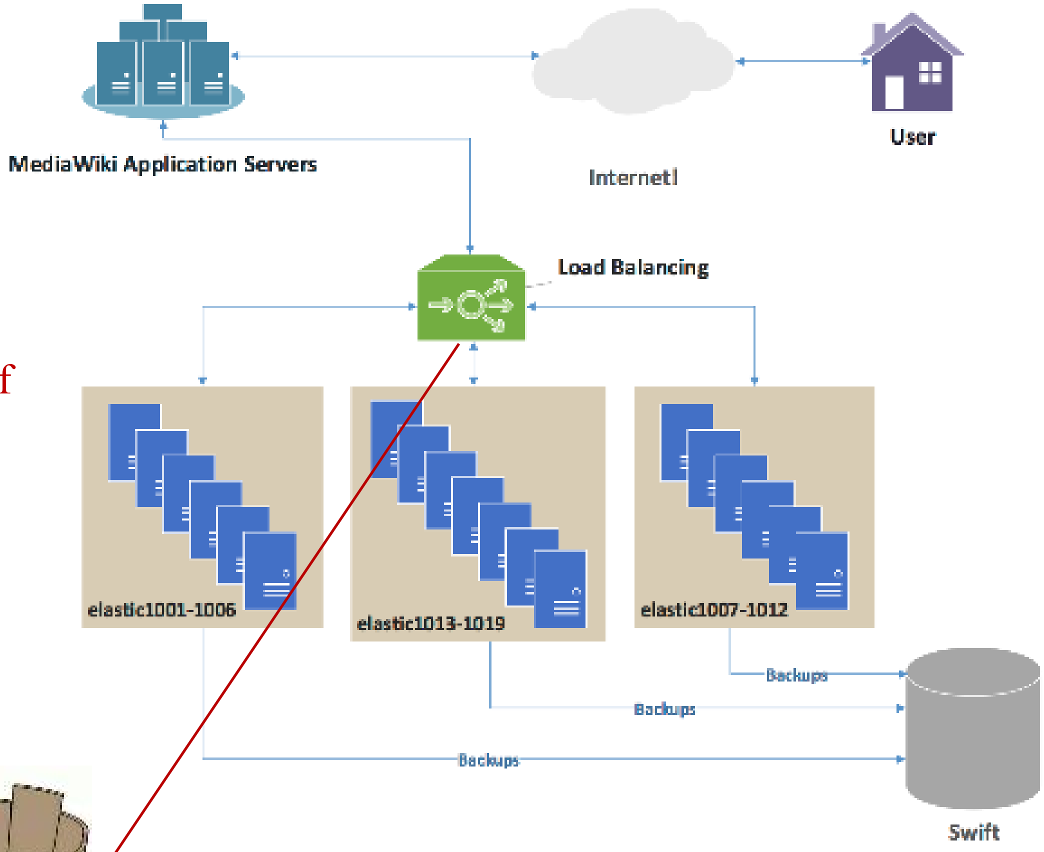
```
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute elements 1-50;
            break;
    case 2: Compute elements 51-100;
            break;
  }
}
```

Computation time - 50 clock cycles

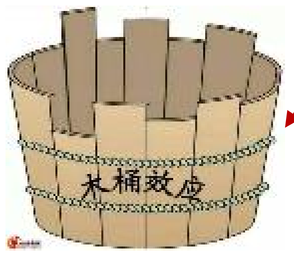
Example

- **Fine-grained parallelism**
  - ✓ Assume there are 100 processors that are responsible for processing the 10*10 image
  - ✓ Ignoring the communication overhead, the 100 processors can process the 10*10 image in 1 clock cycle
  - ✓ Each processor is working on 1 pixel of the image and then communicates the output to other processors
- **Medium-grained parallelism**
  - ✓ Consider that there are 25 processors processing the 10*10 image
  - ✓ The processing of the image will now take 4 clock cycles
- **Coarse-grained parallelism**
  - ✓ If we reduce the processors to 2, then the processing will take 50 clock cycles
  - ✓ Each processor need to process 50 elements which increases the computation time, but the communication overhead decreases as the number of processors which share data decreases

Load balancing



- load balancing refers to the process of distributing a set of tasks over a set of resources (computing units), with the aim of making their overall processing more efficient
- Load balancing can optimize the response time and avoid unevenly overloading some compute nodes while other compute nodes are left idle



Load balancing

- Load balancing is the subject of research in the field of parallel computers
- Two main approaches exist
  - static algorithms, which do not take into account the state of the different machines
  - dynamic algorithms, which are usually more general and more efficient, but require exchanges of information between the different computing units, at the risk of a loss of efficiency

轮询、随机、hash、加权轮询、加权随机、最小连接数、……

Load balancing

- Nature of tasks
  - A load balancing algorithm always tries to answer a specific problem
    - ✓ Take into account: the nature of the tasks, the algorithmic complexity, the hardware architecture, required error tolerance, ……
    - ✓ best meet application-specific requirements
  - The efficiency of load balancing algorithms critically depends on the nature of the tasks
    - ✓ the more information about the tasks is available at the time of decision making, the greater the potential for optimization

Size of tasks

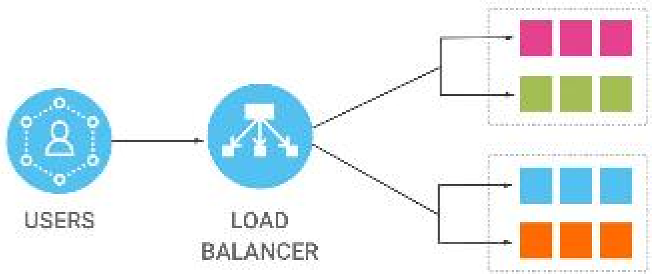
- **the execution time of** each of the tasks
  - ✓ A perfect knowledge of the execution time of each of the tasks allows to reach an optimal load distribution
  - ✓ Knowing the exact execution time of each task is an extremely rare situation
- For this reason, there are several techniques to get an idea of the different execution times
  - ➢ in the fortunate scenario of having tasks of relatively homogeneous size, it is possible to consider that each of them will require approximately the average execution time
  - ➢ add some metadata to each task, depending on the previous execution time for similar metadata, it is possible to make inferences for a future task based on statistics
  - ➢ ……?

Static and dynamic algorithms

- Static
  - A load balancing algorithm is "static“: when it does not take into account the state of the system for the distribution of tasks
  - Measures assumptions beforehand: the arrival times, resource requirements of incoming tasks, the number of processors, their respective power and communication speeds, ……
  - Therefore, static load balancing aims to associate a known set of tasks with the available processors in order to **minimize a certain performance function**

Example

- Static distribution with full knowledge of the tasks: **prefix sum**
  - If the tasks are independent of each other, and if their respective execution time and the tasks can be subdivided:
    - ✓ dividing the tasks in such a way as to give the same amount of computation to each processor, all that remains to be done is to group the results together
    - ✓ Using a prefix sum algorithm, this division can be calculated in logarithmic time with respect to the number of processors



Example

- Prefix sum:

  for array: a[4] = {1,   2,   3,   4};

  prefix sum:

  S[0] = a[0] = 1;

  S[1] = a[0] + a[1] = 1+ 2 = 3;

  S[2] = a[0] + a[1] + a[2] = 1 + 2 + 3 = 6;

  S[3] = a[0] + a[1] + a[2] + a[3] = 1 + 2 + 3 + 4 = 10;

Example

- For serial:

  S[i+1]=S[i]+a[i+1]

- For parallel:

For 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Step1：divide into four arrays and give the same amount of computation to each processor and then calculate their prefix sum

（1 2 3 4） （5 6 7 8） （9 10 11 12） （13 14 15 16）

（1 3 6 10） （5 11 18 26） （9 19 30 42） （13 27 42 58）

Step2: Choose the last number of the result of Step2 and calculate the sum:

（1 3 6 10） （5 11 18 36） （9 19 30 78） （13 27 42 136）

36=10+26

78=10+26+42

Static and dynamic algorithms

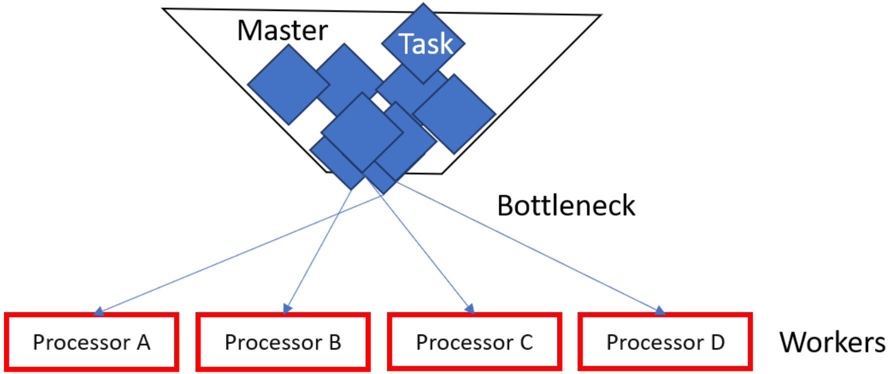
- Dynamic
  - dynamic algorithms: take into account the current load of each of the computing units (also called nodes) in the system
  - tasks can be moved dynamically from an overloaded node to an underloaded node in order to receive faster processing
  - While these algorithms are much more complicated to design, they can produce excellent results, in particular, when the execution time varies greatly from one task to another

Static and dynamic algorithms

- Dynamic
  - Dynamic load balancing architecture can be more modular since it is not mandatory (强制) to have a specific node dedicated to the distribution of work
  - **Unique assignment**: tasks are uniquely assigned to a processor according to its state at a given moment
  - **dynamic assignment**: tasks can be permanently redistributed according to the state of the system and its evolution
  - a load balancing algorithm that **requires too much communication in order to reach its decisions runs the risk of slowing down the resolution of the overall problem**

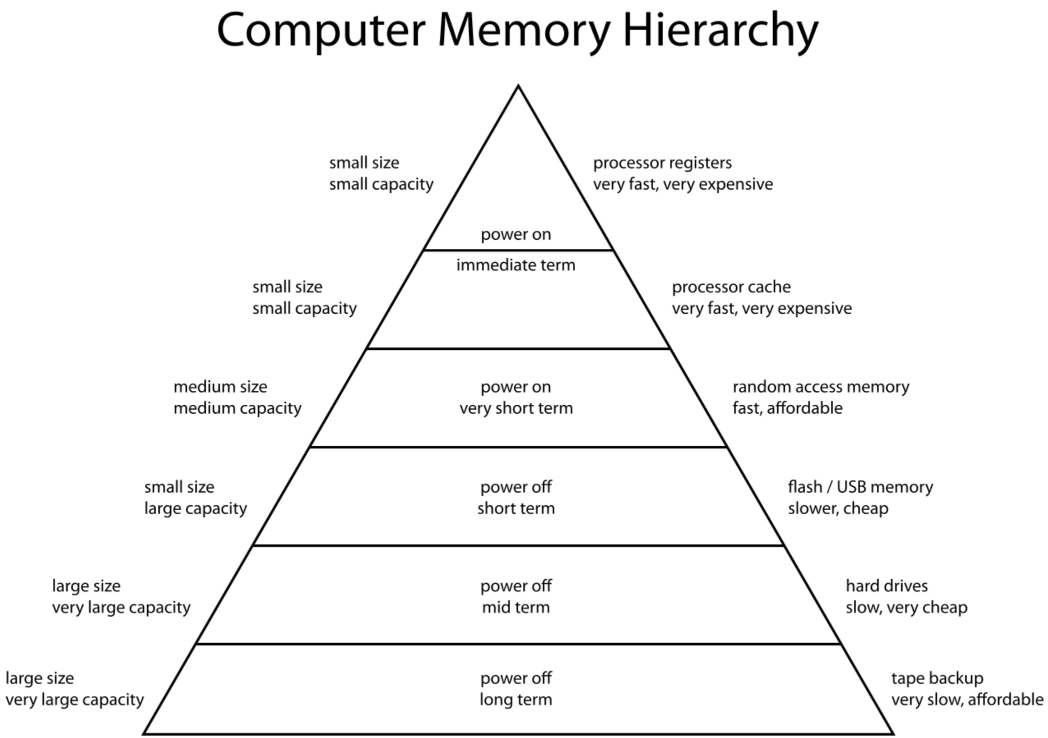
Example:Master-Worker Scheme

- A master distributes the workload to all workers (also referred to as "slaves")
  - ✓ Initially, all workers are idle and report this to the master
  - ✓ The master answers worker requests and distributes the tasks to them.
  - ✓ When he has no more tasks to give, he informs the workers so that they stop asking for tasks.
- The advantage of this system is that it distributes the burden very fairly
  - ✓ if one does not take into account the time needed for the assignment, the execution time would be comparable to the prefix sum seen above **(how to prove?)**



Memory hierarchy

- the memory hierarchy separates computer storage into a hierarchy based on response time
  - Since response time, complexity, and capacity are related, the levels may also be distinguished by their performance and controlling technology
  - Memory hierarchy affects performance in computer architectural design, algorithm predictions, and lower level programming constructs involving locality of reference



Memory hierarchy

- **Designing for high performance** requires considering the restrictions of the memory hierarchy, i.e. the size and capabilities of each component
  - Each of the various components can be viewed as part of a hierarchy of memories (m_1, m_2, ..., m_n) in which each member m_i is typically smaller and faster than the next highest member m_i+1 of the hierarchy
- To limit waiting by higher levels, a lower level will respond by filling a buffer (这是啥？) and then signaling for activating the transfer

Memory hierarchy

- There are four major storage levels
  - Internal – Processor registers and cache
  - Main – the system RAM and controller cards
  - On-line mass storage – Secondary storage
  - Off-line bulk storage – Tertiary and Off-line storage
- Other memory hierarchy structure
  - ✓ a paging algorithm is a virtual memory when designing a computer architecture
  - ✓ a nearline storage between online and offline storage

Example

- the memory hierarchy of an Intel Haswell Mobile processor circa 2013 is:
  - Processor registers – the fastest possible access (usually **1 CPU cycle**). A few thousand bytes in size
  - Cache
    - Level 0 (L0) Micro operations cache – 6,144 bytes/6 KiB in size
    - Level 1 (L1) Instruction cache – 128 KiB in size
    - Level 1 (L1) Data cache – 128 KiB in size. Best access speed is around **700 GB/s**
    - Level 2 (L2) Instruction and data (shared) – 1 MiB in size. Best access speed is around **200 GB/s**
    - Level 3 (L3) Shared cache – 6 MiB in size. Best access speed is around **100 GB/s**
    - Level 4 (L4) Shared cache – 128 MiB in size. Best access speed is around **40 GB/s**
  - Main memory (Primary storage) – GiB in size. Best access speed is around **10 GB/s**. In the case of a NUMA machine, access times may not be uniform
  - Disk storage (Secondary storage) – Terabytes in size. As of 2017, best access speed is from a consumer solid state drive is about **2000 MB/s**
  - Nearline storage (Tertiary storage) – Up to exabytes in size. As of 2013, best access speed is about **160 MB/s**
  - Offline storage

Modern programming

- Taking optimal advantage of the memory hierarchy requires the cooperation of programmers, hardware, and compilers (as well as underlying support from the operating system):
  - **Programmers** are responsible for moving data between disk and memory through file I/O
  - **Hardware** is responsible for moving data between memory and caches
  - **Optimizing compilers** are responsible for generating code that, when executed, will cause the hardware to use caches and registers efficiently

- Modern programming languages mainly assume two levels of memory, main memory and disk storage
- The memory hierarchy will be assessed during code refactoring

Speedup and overhead

Speedup function

Speedup and efficiency

- The performance of a parallel algorithm is determined by calculating its **speedup** and **efficiency**
- **Speedup** is defined as the ratio of the worst-case execution time of the fastest known serial algorithm for a particular problem to the worst-case execution time of the parallel algorithm

➢ **Speedup =** $\dfrac{\textbf{Worst case execution time of sequential algorithm}}{\textbf{Worst case execution time of the parallel algorithm}}$

➢ **Efficiency= Speedup/*p***

Speedup and Scalability

- For p processors

$$\text{Speedup} = \frac{\text{serial time}}{\text{parallel time}} = S(p) \to p$$

$$\text{Efficiency} = \frac{\text{Speedup}}{p} = \frac{S(p)}{p} = E(p) \to 1$$

- Let T_s denote the serial time, T_p the parallel time, and T_0 the overhead, then

$$pT_p = T_s + T_0.$$

$$E(p) = \frac{T_s}{pT_p} = \frac{T_s}{T_s + T_0} = \frac{1}{1 + T_0/T_s}$$

- The scalability analysis of a parallel algorithm measures its capacity to effectively <u>utilize an increasing number of processors</u>

Speedup

- Relating efficiency to work and overhead
- Let W be the problem size

The overhead T_0 depends on W and p: $T_0 = T_0(W, p)$.

The parallel time equals $T_p = \dfrac{W + T_0(W, p)}{p}$

Speedup $S(p) = \dfrac{W}{T_p} = \dfrac{Wp}{W + T_0(W, p)}$.

Efficiency $E(p) = \dfrac{S(p)}{p} = \dfrac{W}{W + T_0(W, p)} = \dfrac{1}{1 + T_0(W, p)/W}$.

- The algorithm scales badly if W must grow exponentially to keep efficiency from dropping
- If W needs to grow only moderately to keep the overhead in check, then the algorithm scales well

Conclusion

- Speedup and overhead
  - serial/parallel time
  - Total parallel computation time
  - overhead t_0
    - communication/synchronization
    - extra computation
    - parallel granularity
    - load balancing
    - memory hierarchy
  - speedup