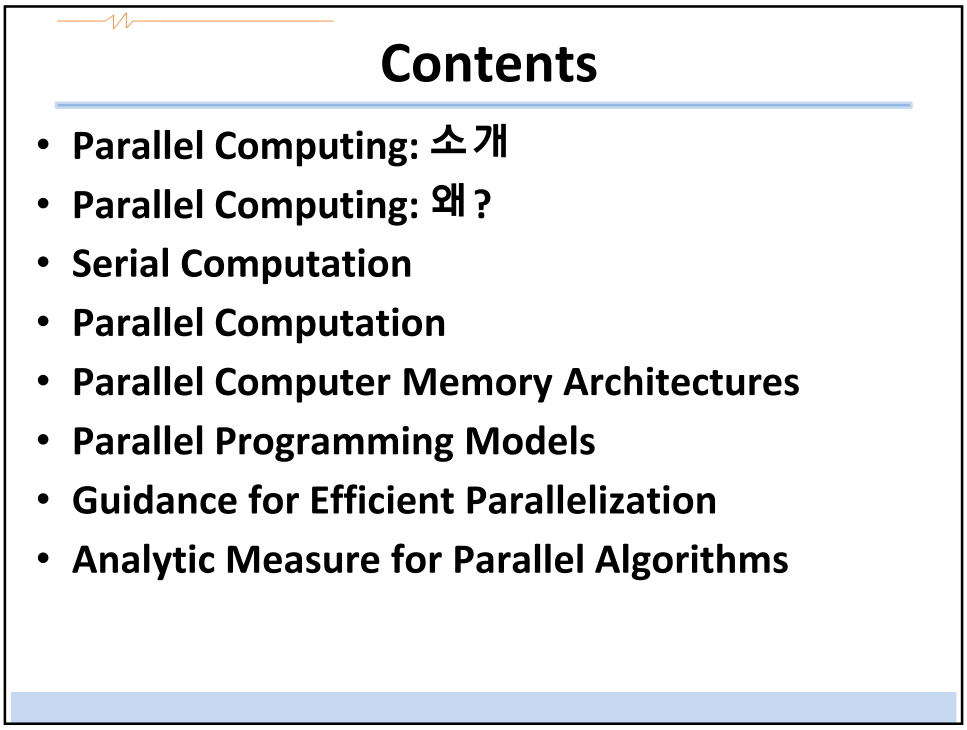
Introduction To Parallel Computing

이 정 근 (Jeong-Gun Lee)

임베디드 SoC 연구실, 한림대학교 소프트웨어융합대학

www.onchip.net

Email: **Jeonggun.Lee@hallym.ac.kr**



Contents

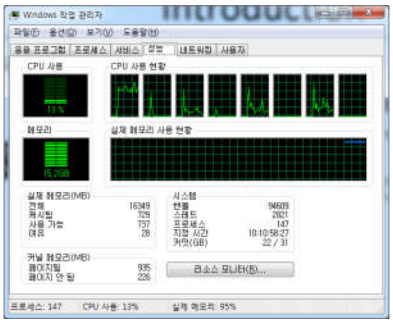
Parallel Computing

- Multicore/Manycore and GPU

인텔 코어i7-7세대 7700 (카비레이크) (정품) 표준PC

인텔(소켓1151) / 쿼드 코어 / 쓰레드 8개 / 64비트 / 3.6GHz / 8MB / 인텔 HD 630 / 350MHz / 65W / 하이퍼스레딩

관련기사 인텔 공인대리점, 7세대 코어 프로세서 봄맞이 퀴즈 이벤트
사용기 [똑똑한 리뷰씨] 게임에 강한 프로세서! 7세대 인텔 코어 프로세서



실제 메모리(MB)

전체	16349
캐시됨	729
사용 가능	737
여유	28

커널 메모리(MB)

페이지됨	935
페이지 안 됨	226

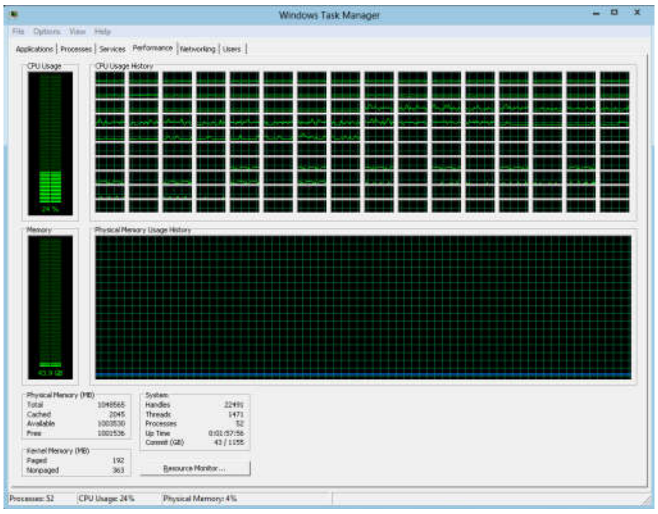
시스템

핸들	94609
스레드	2821
프로세스	147
작동 시간	10:10:58:27
커밋(GB)	22 / 31

프로세스: 147 CPU 사용: 13% 실제 메모리: 95%

Parallel Computing

- Multicore/Manycore and GPU



Physical Memory (MB)

Total	1048565
Cached	2045
Available	1003530
Free	1002536

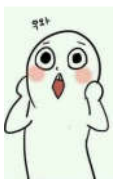
Kernel Memory (MB)

Paged	192
Nonpaged	363

System

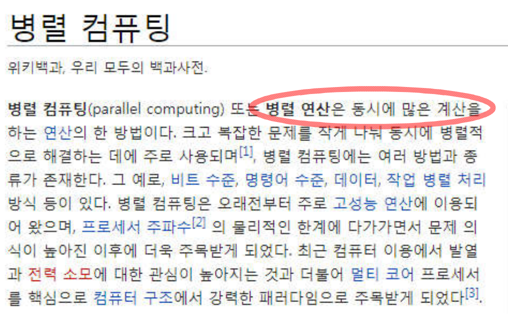
Handles	22491
Threads	1471
Processes	52
Up Time	0:01:57:56
Commit (GB)	43 / 1155

Processes: 52 CPU Usage: 24% Physical Memory: 4%



Parallel Computing

- **First, Wiki Definition …**



병렬 컴퓨팅

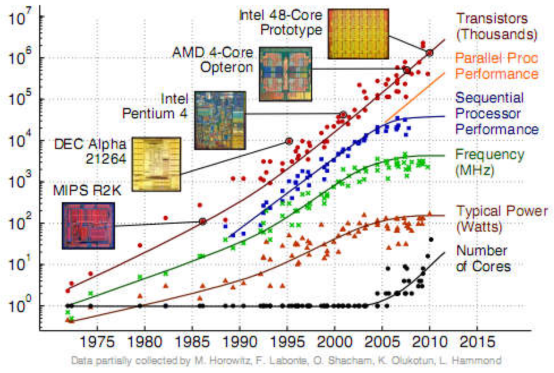
위키백과, 우리 모두의 백과사전.

병렬 컴퓨팅(parallel computing) 또는 **병렬 연산**은 동시에 많은 계산을 하는 연산의 한 방법이다. 크고 복잡한 문제를 작게 나눠 동시에 병렬적으로 해결하는 데에 주로 사용되며[1], 병렬 컴퓨팅에는 여러 방법과 종류가 존재한다. 그 예로, 비트 수준, 명령어 수준, 데이터, 작업 병렬 처리 방식 등이 있다. 병렬 컴퓨팅은 오래전부터 주로 고성능 연산에 이용되어 왔으며, 프로세서 주파수[2] 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목받게 되었다[3].

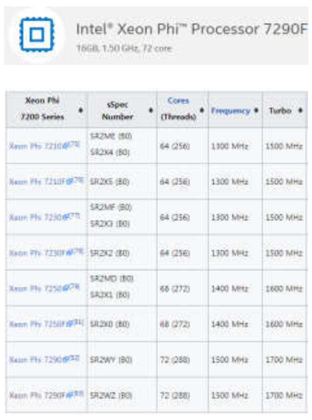
프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목



Prepared by C. Batten - School of Electrical and Computer Engineering - Cornell University - 2005 - retrieved Dec 12 2012 - http://www.csl.cornell.edu/courses/ece5950/handouts/ece5950-overview.pdf



Intel® Xeon Phi™ Processor 7290F

16GB, 1.50 GHz, 72 core

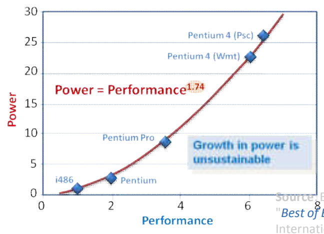
Xeon Phi 7200 Series	sSpec Number	Cores (Threads)	Frequency	Turbo
Xeon Phi 7210[75]	SR2ME (B0) SR2X4 (B0)	64 (256)	1300 MHz	1500 MHz
Xeon Phi 7210F[76]	SR2X5 (B0)	64 (256)	1300 MHz	1500 MHz
Xeon Phi 7230[77]	SR2MF (B0) SR2X3 (B0)	64 (256)	1300 MHz	1500 MHz
Xeon Phi 7230F[78]	SR2X2 (B0)	64 (256)	1300 MHz	1500 MHz
Xeon Phi 7250[79]	SR2MD (B0) SR2X1 (B0)	68 (272)	1400 MHz	1600 MHz
Xeon Phi 7250F[81]	SR2X0 (B0)	68 (272)	1400 MHz	1600 MHz
Xeon Phi 7290[82]	SR2WY (B0)	72 (288)	1500 MHz	1700 MHz
Xeon Phi 7290F[83]	SR2WZ (B0)	72 (288)	1500 MHz	1700 MHz

Each core will have two 512-bit vector units and will support AVX-512 SIMD instructions

Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

- Worse news: Power (normalized to i486) trends



Source: Ed Grochowski, Ronny Ronen, John Shen, Hong Wang, "*Best of Both Latency and Throughput*," Computer Design, International Conference on, pp. 236-243, 2004

- Multi-core
  - **Parallelism** is an **energy-efficient way to achieve performance** [Chandrakasan et al 1992]
  - A larger number of smaller processing elements allows a finer-grained ability to perform dynamic voltage scaling and power down

Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

- Worse news: Power (normalized to i486) trends

Small is Beautiful !

That does not immediately imply that "smallest is best" !

  - **Parallelism** is an **energy-efficient way to achieve performance** [Chandrakasan et al 1992]
  - A larger number of smaller processing elements allows a finer-grained ability to perform dynamic voltage scaling and power down

Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

- Multi-core Challenges: The **3 P**'s
  - Performance challenge
  - **Power efficiency challenge**

Processor	Power	Perf.	Power Efficiency
Itanium 2	100W	1	1
RISC*	1/2W	1/8X**	**25X**

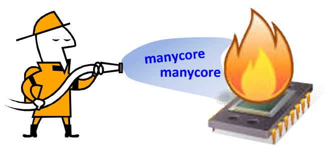
Assuming 130nm
* 90's RISC at 405 MHz
** e.g., Timberwolf (SpecInt)

  - Programming challenge

Anant Agarwal (professor at MIT), "Going Multi-core: Opportunities, Challenges and Dreams" in his keynote talk at MULTI-CORE EXPO, Mar. 2006

Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

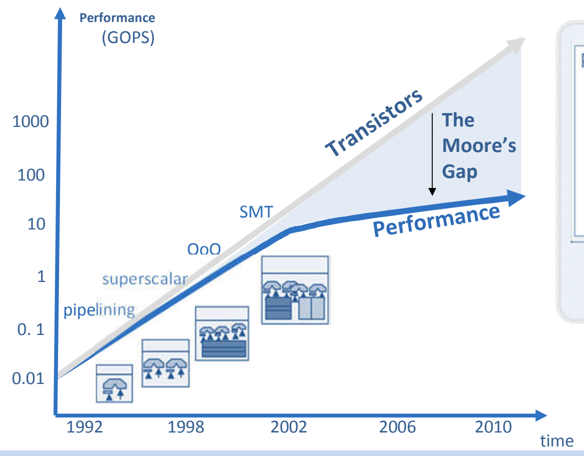


+ Diminishing Return

Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

Diminishing Return

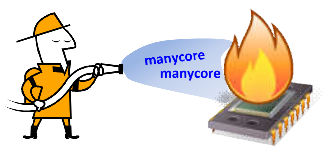


Pentium 3	Pentium 4
- 1 GHz	- 1.4 GHz
- Year 2000	- Year 2000
- 0.18 um	- 0.18 um
- 28M Tr.	- 42M Tr. 50%↑
343 (SpecInt 2000)	393 15%↑ (SpecInt 2000)

1. ***Diminishing returns*** from single CPU
2. ***Wire*** delays
3. ***Power*** envelopes

Parallel Computing: Why ?

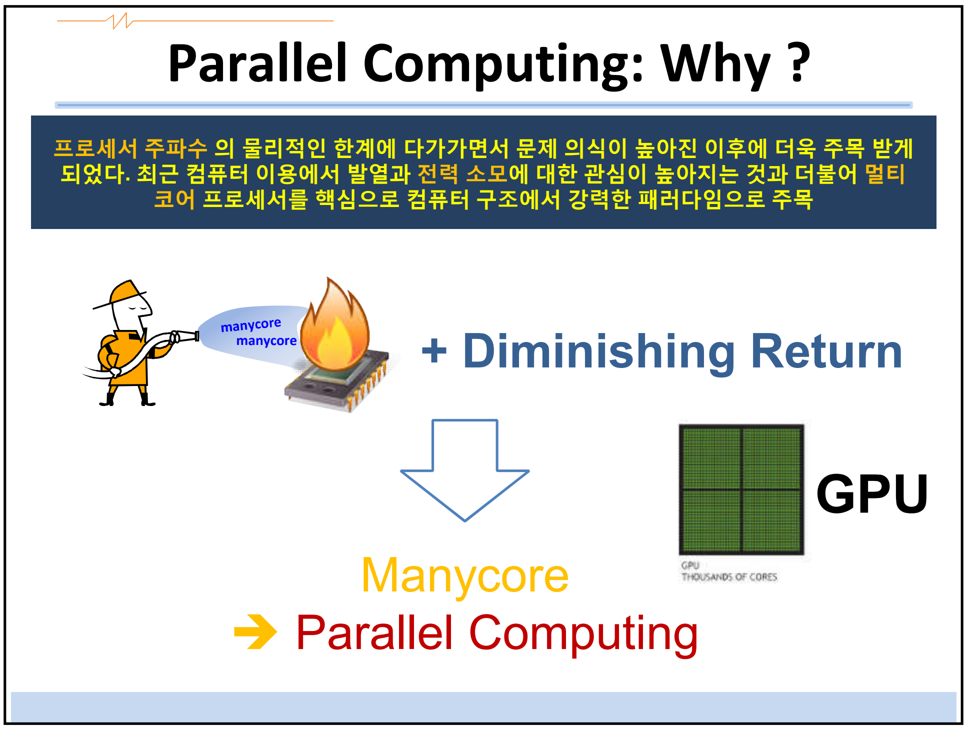
프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목



+ Diminishing Return

Manycore

➔ Parallel Computing



Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

manycore
manycore

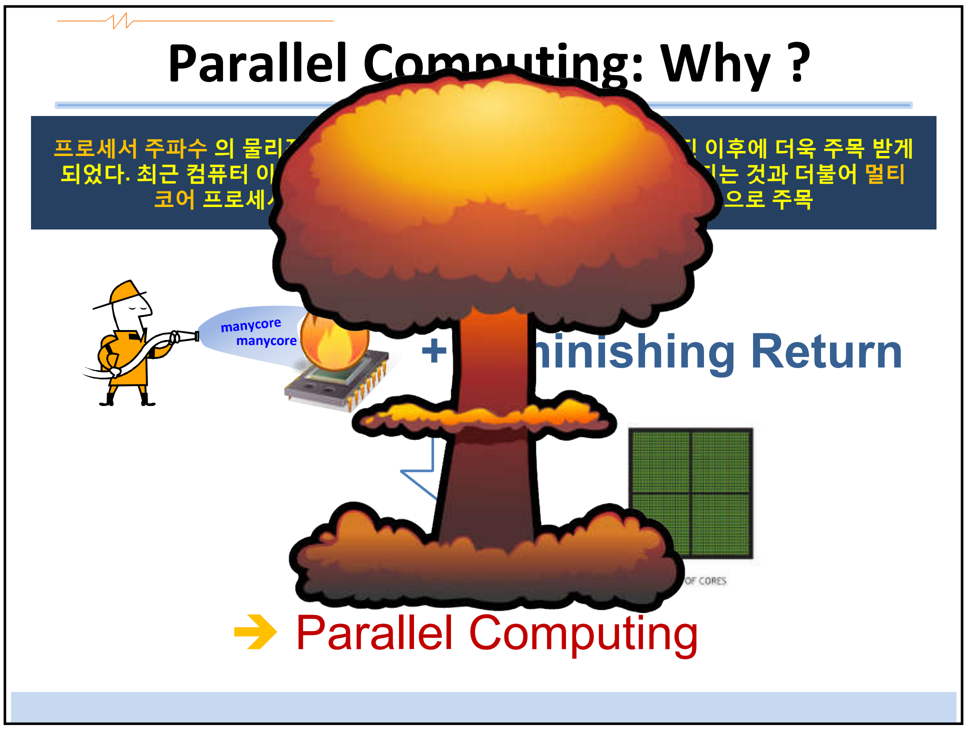
+ Diminishing Return

GPU

GPU
THOUSANDS OF CORES

Manycore

➔ Parallel Computing



Parallel Computing: Why ?

프로세서 주파수 의 물리 … 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이 … 지는 것과 더불어 멀티 코어 프로세 … 으로 주목

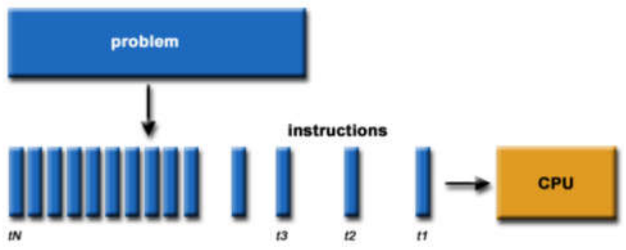
+ … inishing Return

➔ Parallel Computing

manycore
manycore
OF CORES

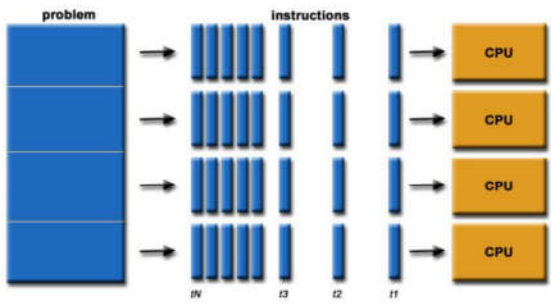
Serial Computation

- Traditionally software has been written for serial computations:
  - A problem is broken into a discrete set of instructions
  - Instructions are executed <u>one after another</u>
  - Only one instruction can be executed at any moment in time



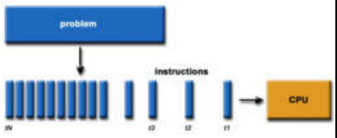
Parallel Computing

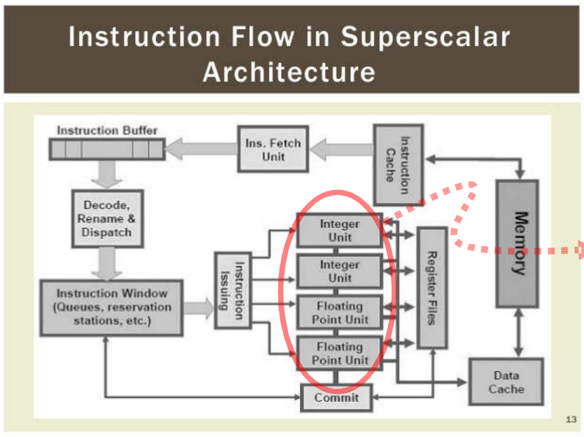
- Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:
  - To be run using multiple CPUs
    - A problem is broken into discrete parts that can be solved concurrently (Task-Level or Data-Level Parallelism)
    - Each part is further broken down to a series of instructions
    - Instructions from each part execute simultaneously on different CPUs



Single CPU → No Parallelism ?

- If a CPU has multiple functional units ?




Instruction Flow in Superscalar Architecture

Instruction Level Parallelism (ILP)

Instruction Level Parallelism

- **ILP** = Instruction Level Parallelism
  - Even in a single CPU, more than one instruction can be executed at a time with advanced microarchitecture techniques

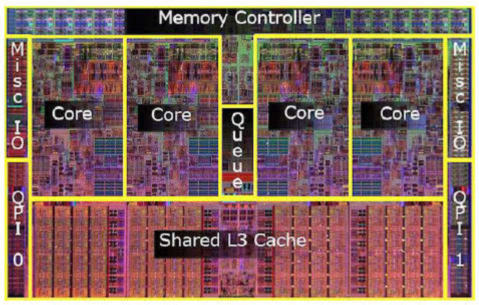
Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

$\downarrow i$ $t \rightarrow$

IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Parallel Computers

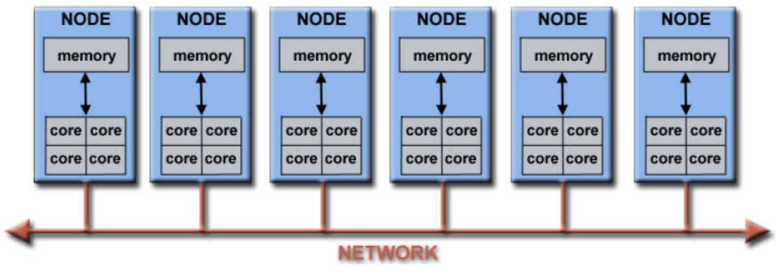
- Virtually all stand-alone computers today are parallel machines from a hardware perspective:
  - Multiple functional units (floating point, integer, GPU, etc.)
  - Multiple execution units / cores
  - Multiple hardware threads



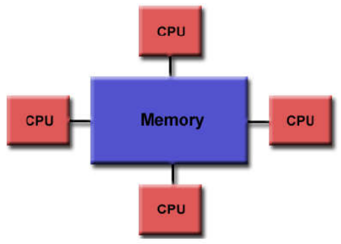
Intel Core i7 CPU and its major components

Parallel Computers

- Networks connect multiple stand-alone computers (nodes) to create larger parallel computer clusters
  - Each compute node is a multi-processor parallel computer in itself
  - Multiple compute nodes are networked together with an InfiniBand network
  - Special purpose nodes, also multi-processor, are used for other purposes



Parallel Computer Memory Architectures

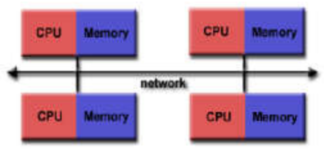


- ***Shared Memory (공유메모리):***
  - Multiple processors can operate independently, but share the same memory resources
  - Changes in a memory location caused by one CPU are visible to all processors

Advantages:
- Global address space provides a user-friendly programming perspective to memory
- Fast and uniform data sharing due to proximity of memory to CPUs

Disadvantages:
- Lack of scalability between memory and CPUs. Adding more CPUs increases traffic on the shared memory-CPU path
- Programmer responsibility for "correct" access to global memory

Parallel Computer Memory Architectures



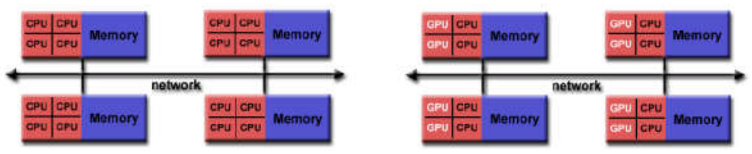
- ***Distributed Memory(분산메모리):***
  - Requires a communication network to connect inter-processor memory
  - Processors have their own local memory.
  - Changes made by one CPU have no effect on others
  - Requires communication to exchange data among processors

Advantages:
- Memory is scalable with the number of CPUs
- Each CPU can rapidly access its own memory without overhead incurred with trying to maintain global cache coherency

Disadvantages:
- Programmer is responsible for many of the details associated with data communication between processors

Parallel Computer Memory Architectures

- *Hybrid Distributed-Shared Memory(분산공유메모리)*:
  - The largest and fastest computers in the world today employ both shared and distributed memory architectures.



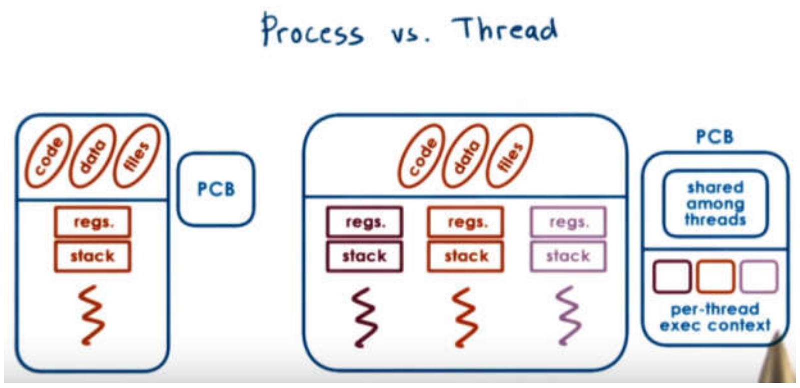
- Shared memory component can be a shared memory machine and/or GPU
- Processors on a compute node share same memory space
- Requires communication to exchange data between compute nodes

Parallel Programming Models

- Parallel Programming Models exist as an abstraction above hardware and memory architectures
  - Shared Threads Models (Pthreads, OpenMP)
  - Distributed Memory / Message Passing (MPI)
  - Single Instruction/Program Multiple Data (SIMD, SPMD)
    - Vector Processing (MMX, SSE, AVX, …)
    - Single Instruction Multiple Thread (SIMT for GPU)
  - …

Shared Threads Models

- **POSIX Threads**



POSIX (**포직스**, /ˈpɒzɪks/)는 **이식 가능 운영 체제 인터페이스**(移植可能運營體制 interface, **p**ortable **o**perating **s**ystem **i**nterface)의 약자로, 서로 다른 UNIX OS의 공통 API를 정리하여 이식성이 높은 유닉스 응용 프로그램을 개발하기 위한 목적으로 IEEE가 책정한 애플리케이션 인터페이스 규격이다. POSIX의 마지막 글자 X는 유닉스 호환 운영체제에 보통 X가 붙는 것에서 유래한다.

Shared Threads Models

- **POSIX Threads**

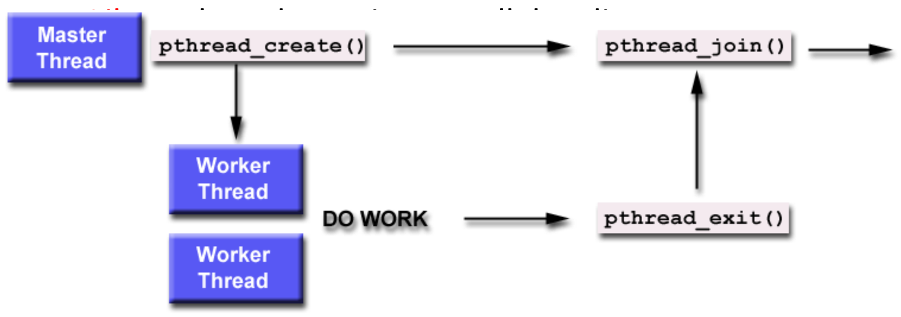
POSIX (**포직스**, /ˈpɒzɪks/)는 **이식 가능 운영 체제 인터페이스**(移植可能運營體制 interface, **p**ortable **o**perating **s**ystem **i**nterface)의 약자로, 서로 다른 UNIX OS의 공통 API를 정리하여 이식성이 높은 유닉스 응용 프로그램을 개발하기 위한 목적으로 IEEE가 책정한 애플리케이션 인터페이스 규격이다. POSIX의 마지막 글자 X는 유닉스 호환 운영체제에 보통 X가 붙는 것에서 유래한다.

- Library based; requires explicit parallel coding
- C Language only; Interfaces for Perl, Python and others exist
- Commonly referred to as **Pthreads**
- Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations
- Very explicit parallelism; requires significant programmer attention to detail

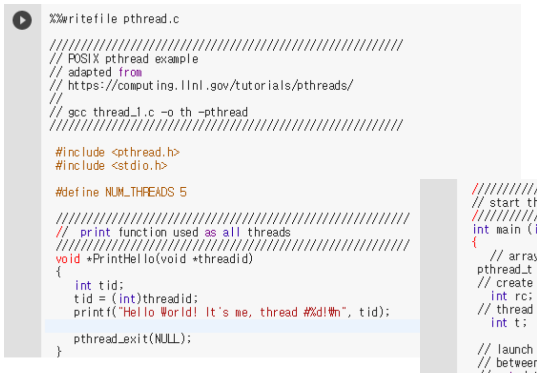
Shared Threads Models

- **POSIX Threads**

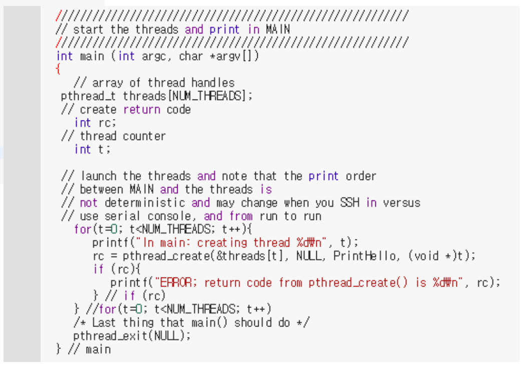
POSIX (**포직스**, /ˈpɒzɪks/)는 **이식 가능 운영 체제 인터페이스**(移植可能運營體制 interface, **p**ortable **o**perating **s**ystem **i**nterface)의 약자로, 서로 다른 UNIX OS의 공통 API를 정리하여 이식성이 높은 유닉스 응용 프로그램을 개발하기 위한 목적으로 IEEE가 책정한 애플리케이션 인터페이스 규격이다. POSIX의 마지막 글자 X는 유닉스 호환 운영체제에 보통 X가 붙는 것에서 유래한다.



Let’s Dive in- pth_hello.c



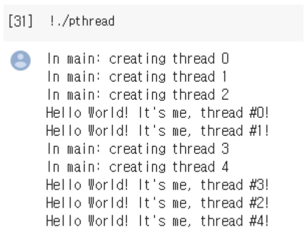
```
%%writefile pthread.c

/////////////////////////////////////////////////////
// POSIX pthread example
// adapted from
// https://computing.llnl.gov/tutorials/pthreads/
//
// gcc thread_1.c -o th -pthread
/////////////////////////////////////////////////////

 #include <pthread.h>
 #include <stdio.h>

 #define NUM_THREADS 5

 /////////////////////////////////////////////////////
 //  print function used as all threads
 /////////////////////////////////////////////////////
 void *PrintHello(void *threadid)
 {
    int tid;
    tid = (int)threadid;
    printf("Hello World! It's me, thread #%d!₩n", tid);

    pthread_exit(NULL);
 }
```



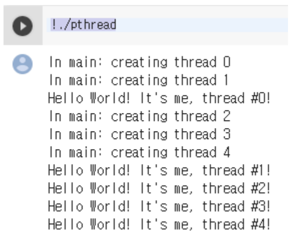
```
/////////////////////////////////////////////////////
// start the threads and print in MAIN
/////////////////////////////////////////////////////
int main (int argc, char *argv[])
{
   // array of thread handles
 pthread_t threads[NUM_THREADS];
 // create return code
   int rc;
 // thread counter
   int t;

 // launch the threads and note that the print order
 // between MAIN and the threads is
 // not deterministic and may change when you SSH in versus
 // use serial console, and from run to run
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %d₩n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d₩n", rc);
      } // if (rc)
   } //for(t=0; t<NUM_THREADS; t++)
   /* Last thing that main() should do */
   pthread_exit(NULL);
} // main
```

https://github.com/jeonggunlee/CUDATeaching/blob/master/01_cuda_lab/01_simple.ipynb

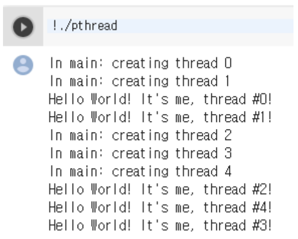
Output



```
[31]  !./pthread

In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #2!
Hello World! It's me, thread #4!
```



```
!./pthread

In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```



```
!./pthread

In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #2!
Hello World! It's me, thread #4!
Hello World! It's me, thread #3!
```

Some disorder is possible

Shared Threads Models

- **OpenMP**
  - Compiler directive based; can use serial code
  - Portable / multi-platform, including Unix and Windows platforms
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for "**incremental parallelism**"

Shared Threads Models

- **OpenMP**
  - Compiler directive based; can use serial code

Sequential Program

```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

Shared Threads Models

- **OpenMP**
  - Single Program Multiple Data (SPMD)

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

Thread 0

```
#include "omp.h"
void main()
{
  int i, k, N
  double A[N]
  lb = 0;
  ub = 250;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 1

```
#include "omp.h"
void main()
{
  int i, k, N
  double A[N]
  lb = 250;
  ub = 500;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 2

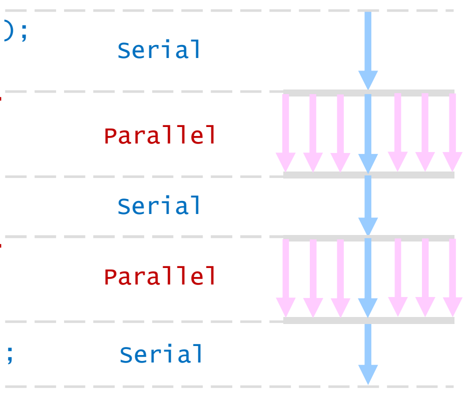
```
#include "omp.h"
void main()
{
  int i, k, N
  double A[N]
  lb = 500;
  ub = 750;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 3

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  lb = 750;
  ub = 1000;
  for (i=lb;i<ub;i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

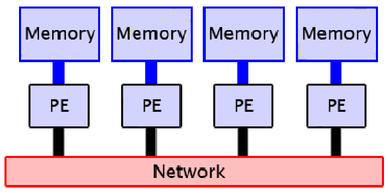
OpenMP Fork-and-Join model

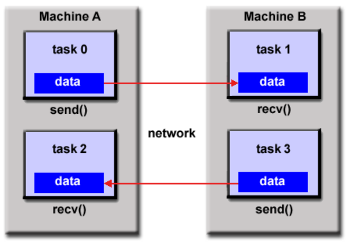
```
printf("program begin\n");
N = 1000;
#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];
M = 500;
#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];
printf("program done\n");
```



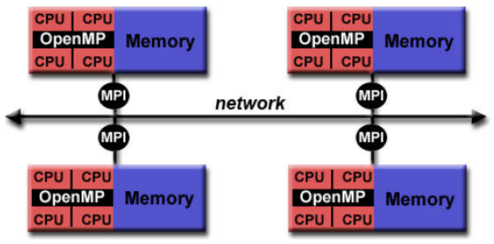
Distributed Memory / Message Passing Models

- Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines
- Tasks exchange data through communications by sending and receiving messages



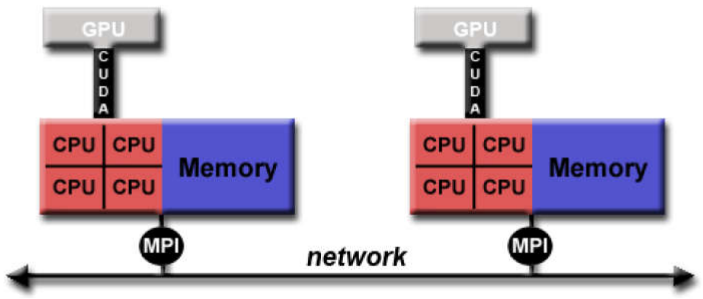


Hybrid Parallel Programming Models



- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP)
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI
- This hybrid model lends itself well to the increasingly common hardware environment of clustered multi/many-core machines

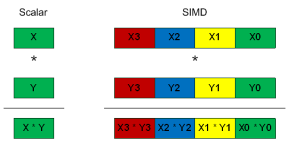
Hybrid Parallel Programming Models

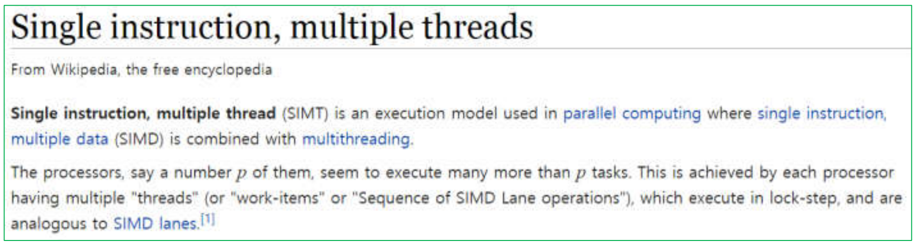


- Another similar and increasingly popular example of a hybrid model is using MPI with GPU (Graphics Processing Unit) programming
  - GPUs perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI

Single Instruction Multiple Data

- Single Instruction/Program Multiple Data (SIMD/SPMD)
  - Vector Processing
    - Intel: MMX, SSE, **AVX**
    - ARM: **NEON**
  - Single Instruction Multiple Thread (SIMT)
    - **GPU** !





Single instruction, multiple threads

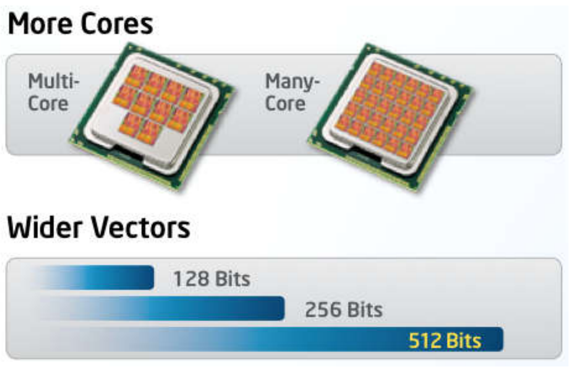
From Wikipedia, the free encyclopedia

Single instruction, multiple thread (SIMT) is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading.

The processors, say a number p of them, seem to execute many more than p tasks. This is achieved by each processor having multiple "threads" (or "work-items" or "Sequence of SIMD Lane operations"), which execute in lock-step, and are analogous to SIMD lanes.[1]

Terms

- SIMD: Single Instruction Multiple Data
- MMX: MultiMedia eXtensions
- SSE: Streaming SIMD Extensions
- AVX: Advanced Vector eXtensions (From Sandy Bridge)



SIMD Vectorization

- To sum the values of 2 arrays, a conventional CPU needs one add operation (“+”) per array index:

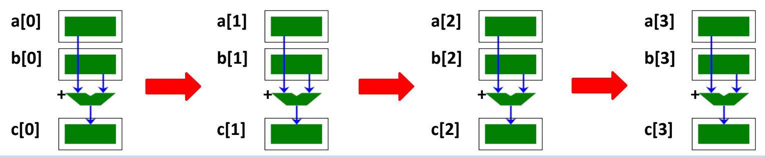
```
double  a[4] = {1.0, 2.0, 3.0, 4.0};
double  b[4] = {1.0, 2.0, 3.0, 4.0};
double   c[4];

c[0] =  a[0] + b[0];
c[1] =  a[1] + b[1];
c[2] =  a[2] + b[2];
c[3] =  a[3] + b[3];
```

sequential array sum

```
double a[4] = {1.0, 2.0, 3.0, 4.0};
double b[4] = {1.0, 2.0, 3.0, 4.0};
              double c[4];
                   int i;

         for(i=0; i < 4; i++) {
            c[i] = a[i] + b[i];
                              }
```

array sum using loop

- Reason: a register of a conventional CPU can only hold only 1 data item at a time (such a register is called a **scalar register**):



SIMD Vectorization (cont.)

- Vector processors have *large registers that can **hold multiple values of the same data type***.

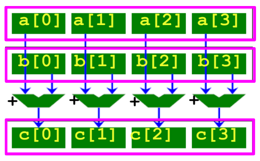
```
double a[4] = {1.0, 2.0, 3.0, 4.0};
double b[4] = {1.0, 2.0, 3.0, 4.0};
double c[4];

 c[0] = a[0] +                b[0];
 c[1] = a[1] +                b[1];
 c[2] = a[2] +                b[2];
 c[3] = a[3] +                b[3];
```

sequential array sum

```
__m256d a = {1.0, 2.0, 3.0, 4.0};
__m256d b = {1.0, 2.0, 3.0, 4.0};
__m256d c;

c = _mm256_add_pd(a,b);
```

data-parallel array sum using vectors

- An Intel AVX register can hold ***4 double values*** at once:
- Called a **vector** of 4 doubles
  - every **double** value is a **vector element**
- **_mm256_add_pd()** operates on vectors
  - In one go, individual elements from vectors a and b are added and stored in vector c (**data parallelism!**)

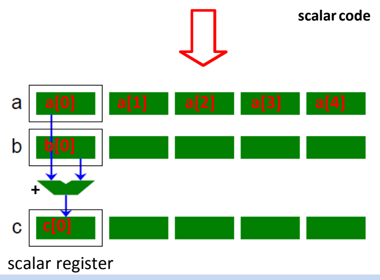


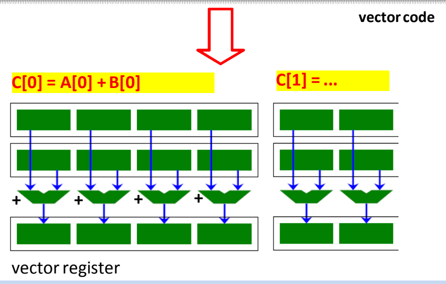
SIMD Vectorization (cont.)

- Instead of arrays of integers, we can define arrays of vectors of integers:

```
double a[4*MAX], b[4*MAX];
double c[4*MAX];
int ctr;

for (ctr = 0; ctr < 4*MAX; ctr++) {
    c[ctr] = a[ctr] + b[ctr];
}
```

```
#define MAX ...
__m256d A[MAX], B[MAX], C[MAX];
int ctr;

for (ctr = 0; ctr < MAX; ctr++) {
   C[ctr] = _mm256_add_pd(A[ctr], B[ctr];
}
```

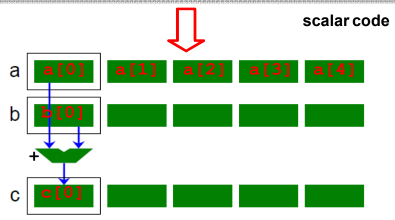


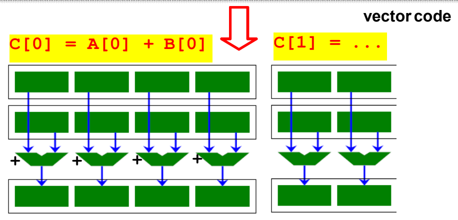


SIMD Vectorization (cont.)

```
double a[4*MAX], b[4*MAX];
double c[4*MAX];
int ctr;

for (ctr = 0; ctr < 4*MAX; ctr++) {
    c[ctr] = a[ctr] + b[ctr];
}
```

```
#define MAX ...
__m256d A[MAX], B[MAX], C[MAX];
int ctr;

for (ctr = 0; ctr < MAX; ctr++) {
   C[ctr] = _mm256_add_pd(A[ctr], B[ctr];
}
```





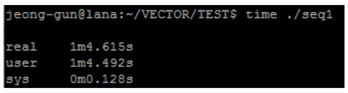
- Vector code uses single instruction, but multiple data items (vector elements)
  - called Single Instruction Multiple Data (SIMD)
- Scalar code operates on 1 data item per operation
  - called Single Instruction Single Data (SISD)

SIMD Vectorization (cont.)

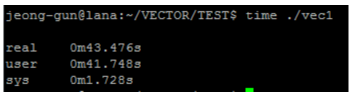
```
double a[4*MAX], b[4*MAX];
double c[4*MAX];
int ctr;

for (ctr = 0; ctr < 4*MAX; ctr++) {
     c[ctr] = a[ctr] + b[ctr];
}
```

```
#define MAX ...
__m256d A[MAX], B[MAX], C[MAX];
int ctr;

for (ctr = 0; ctr < MAX; ctr++) {
    C[ctr] = _mm256_add_pd(A[ctr], B[ctr];
}
```

With "MAX 5000000"



```
jeong-gun@lana:~/VECTOR/TEST$ time ./seq1

real    1m4.615s
user    1m4.492s
sys     0m0.128s
```



```
jeong-gun@lana:~/VECTOR/TEST$ time ./vec1

real    0m43.476s
user    0m41.748s
sys     0m1.728s
```

64 sec **43 sec**

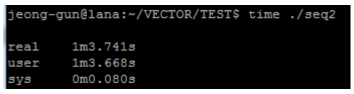
~ 1.48 Speedup < 4

SIMD Vectorization (cont.)

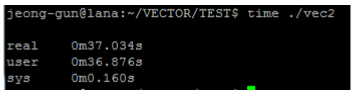
```
float a[4*MAX], b[4*MAX];
float c[4*MAX];
int ctr;

for (ctr = 0; ctr < 4*MAX; ctr++) {
     c[ctr] = a[ctr] + b[ctr];
}
```

```
#define MAX ...
__m256 A[MAX], B[MAX], C[MAX];
int ctr;

for (ctr = 0; ctr < MAX; ctr++) {
    C[ctr] = _mm256_add_ps(A[ctr], B[ctr];
}
```

With "MAX 5000000"



```
jeong-gun@lana:~/VECTOR/TEST$ time ./seq2

real    1m3.741s
user    1m3.668s
sys     0m0.080s
```



```
jeong-gun@lana:~/VECTOR/TEST$ time ./vec2

real    0m37.034s
user    0m36.876s
sys     0m0.160s
```

63 sec **36 sec**

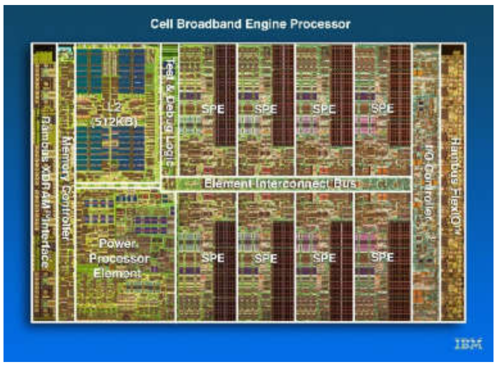
~ 1.75 Speedup < 8

Multiple Program Multiple Data

- Multiple autonomous processors simultaneously operating at least 2 independent programs.
- Typically such systems pick one node to be the "**host**" ("the explicit host/node programming model") or "**manager**" (the "Manager/Worker" strategy), which runs one program that farms out data to all the other nodes which all run a second program.
- Those other nodes then return their results directly to the manager.
  - An example of this would be the Sony PlayStation 3 game console, with its SPU/PPU processor.

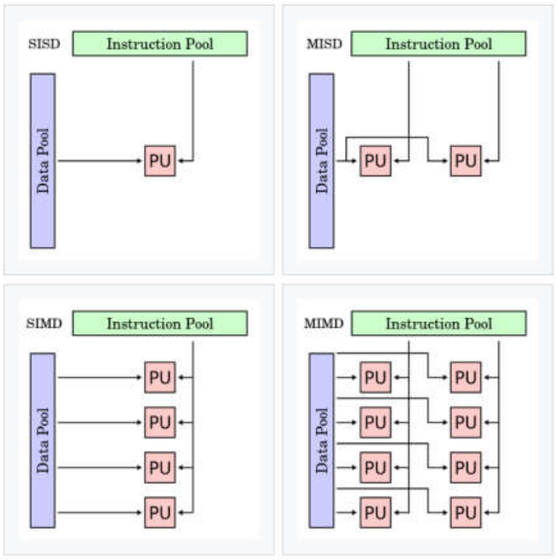
Multiple Program Multiple Data

- Those other nodes then return their results directly to the manager.
  - An example of this would be the Sony PlayStation 3 game console, with its SPU/PPU processor.

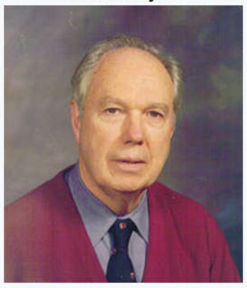


- 1 PPE core @ 3.2GHz
  - 64bit hyperthreaded PowerPC
  - 512KB L2 cache
- 8 SPE cores @ 3.2GHz
  - 128bit SIMD optimized
  - 256KB SRAM

Flynn's Classification



Michael Flynn

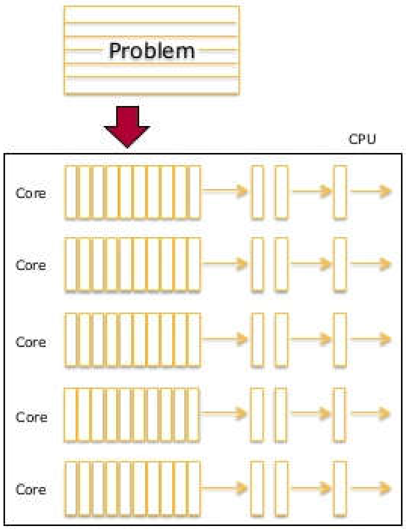


Born	May 20, 1934 (age 82) New York City
Nationality	American
Institutions	Stanford University
Alma mater	Manhattan College Syracuse University Purdue University
Known for	Flynn's taxonomy
Notable awards	Harry H. Goode Memorial Award

Three Ways Of Doing Anything Faster [Pfister]

Hasso Plattner Institut

60

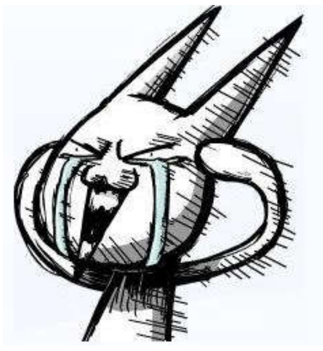


- Work harder (clock speed)
  - Power wall problem
  - Memory wall problem
- Work smarter (optimization, caching)
  - ILP wall problem
  - Memory wall problem
- **Get help** (parallelization)
  - More cores per single CPU
  - Software needs to exploit them in the right way
  - Memory wall problem

Can my code be parallelized?

Interesting Quotes about Parallel Programming

1. "There are 3 rules to follow when parallelizing large codes. Unfortunately, no one knows what these rules are." ~ W. Somerset Maugham, Gary Montry
2. "The wall is there. We probably won't have any more products without multicore processors [but] we see a lot of problems in parallel programming." ~ Alex Bachmutsky
3. "We can solve [the software crisis in parallel computing], but only if we work from the algorithm down to the hardware — not the traditional hardware-first mentality." ~ Tim Mattson
4. "[The processor industry is adding] more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it." ~ Steve Jobs



Can my code be parallelized?

- Does it have large loops that repeat the same operations?
- Does your code do multiple tasks that are not dependent on one another? If not, is the dependency weak?
- Is the amount of communications small?
- Do multiple tasks depend on the same data?
- Does the order of operations matter?

Guidance for Efficient Parallelization

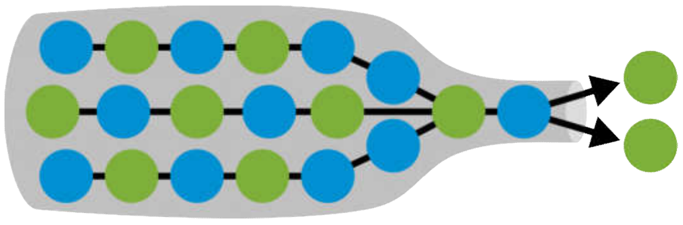
- Is it even worth parallelizing my code?
  - Does your code take an intractably long amount of time to complete?
  - Do you run a single large model or do statistics on multiple small runs?
  - Would the amount of time it take to parallelize your code be worth the gain in speed?

Guidance for Efficient Parallelization

- Parallelizing established code vs. starting from scratch
  - **Established code**: Maybe easier / faster to parallelize, but my not give good performance or scaling (OpenMP)
  - **Start from scratch**: Takes longer, but will give better performance, accuracy, and gives the opportunity to turn a “black box” into a code you understand

Guidance for Efficient Parallelization

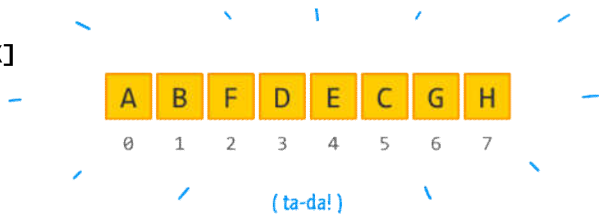
- Increase the fraction of your program that can be parallelized
  - **Identify the most time consuming parts** of your program and parallelize them. This could require modifying your intrinsic algorithm and code's organization

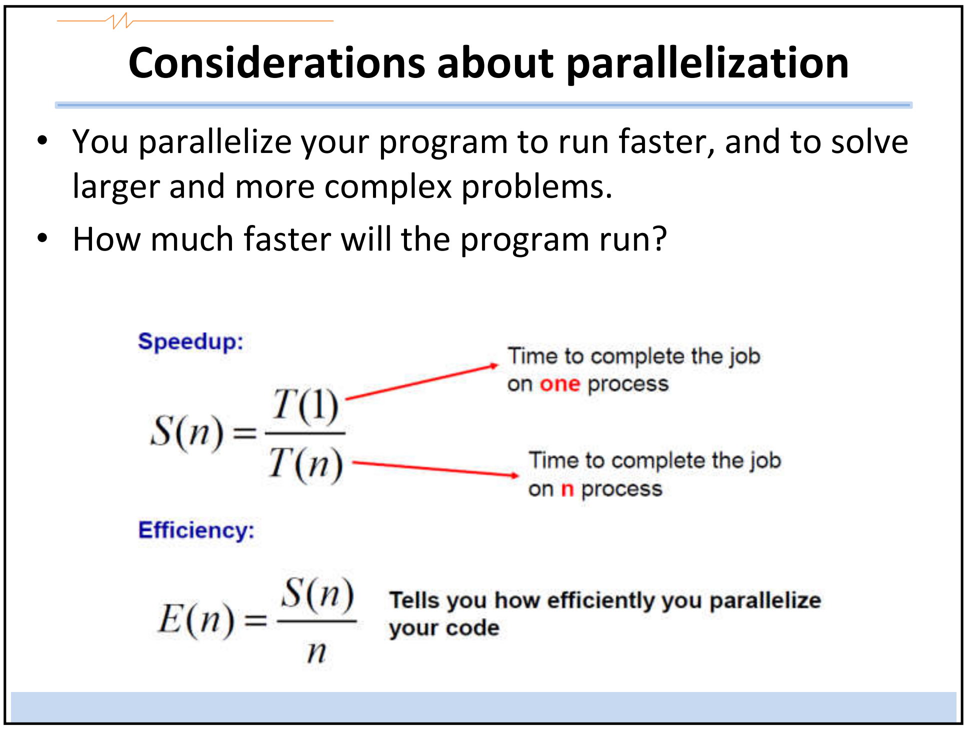


Guidance for Efficient Parallelization

- Balance parallel workload
- Minimize time spent in communication
- Use **simple arrays** instead of user defined derived types

```
int simpleArray[MAX]
```





Considerations about parallelization

- You parallelize your program to run faster, and to solve larger and more complex problems.
- How much faster will the program run?

Speedup:

$$S(n) = \frac{T(1)}{T(n)}$$

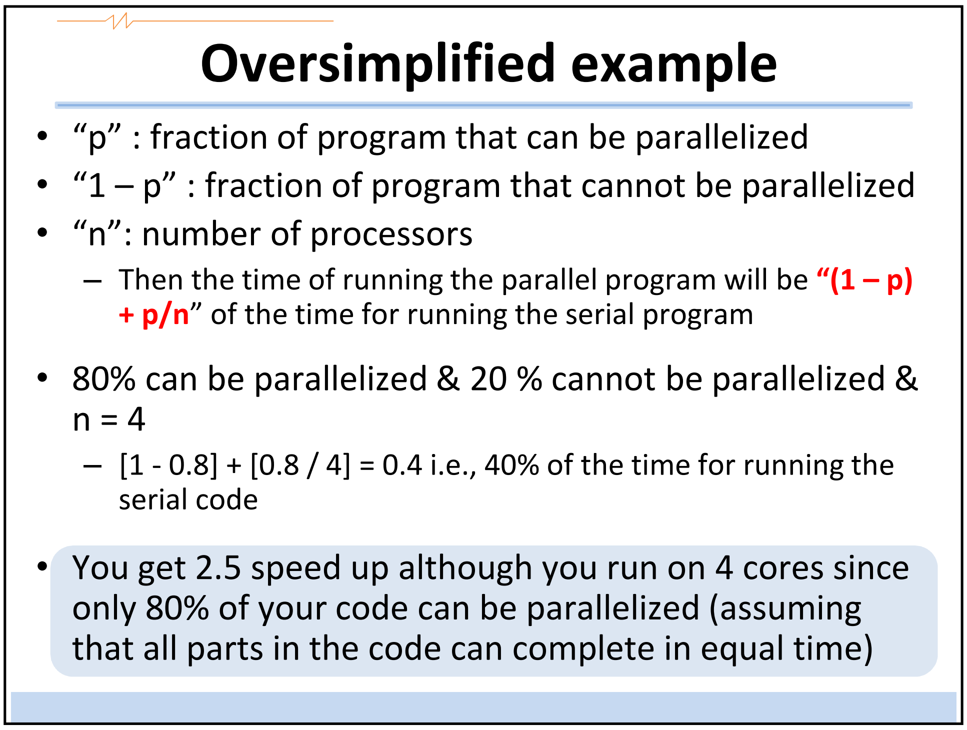
Time to complete the job on one process

Time to complete the job on n process

Efficiency:

$$E(n) = \frac{S(n)}{n}$$

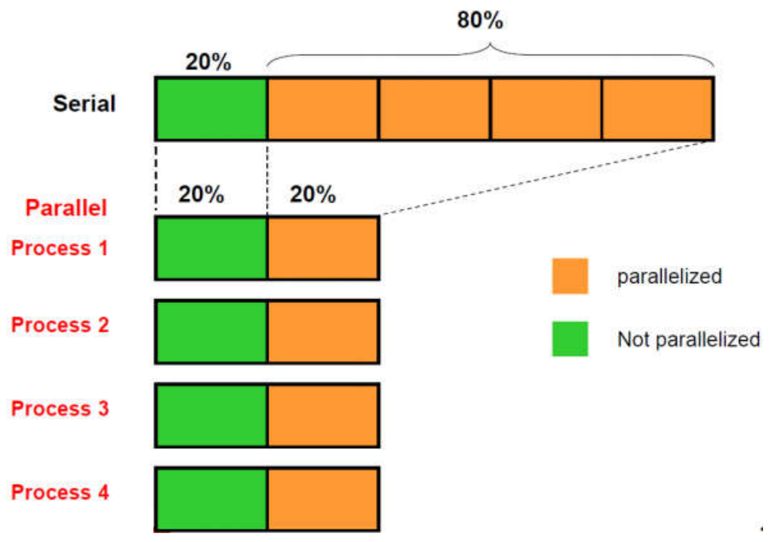
Tells you how efficiently you parallelize your code



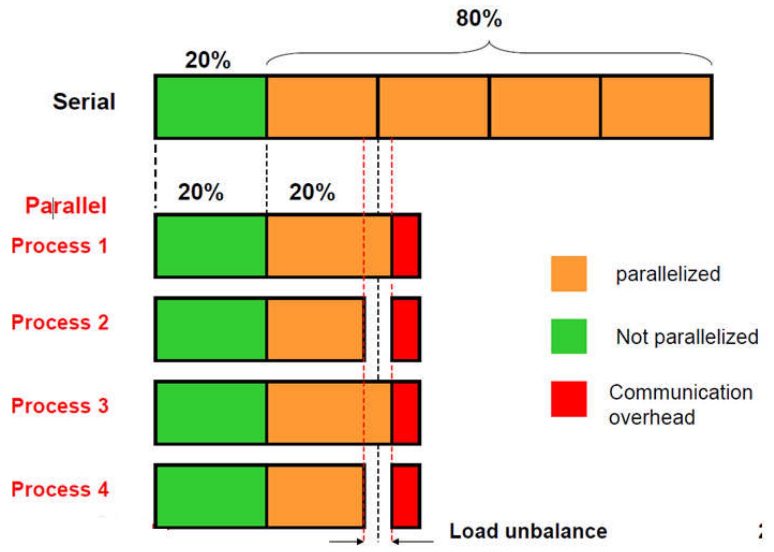
Oversimplified example

- “p” : fraction of program that can be parallelized
- “1 – p” : fraction of program that cannot be parallelized
- “n”: number of processors
  - Then the time of running the parallel program will be **“(1 – p) + p/n**” of the time for running the serial program
- 80% can be parallelized & 20 % cannot be parallelized & n = 4
  - [1 - 0.8] + [0.8 / 4] = 0.4 i.e., 40% of the time for running the serial code
- You get 2.5 speed up although you run on 4 cores since only 80% of your code can be parallelized (assuming that all parts in the code can complete in equal time)

Oversimplified example

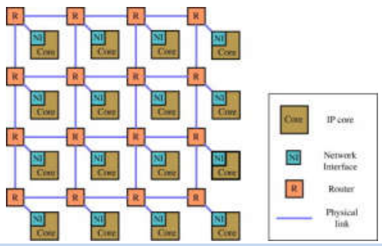


More realistic example:



Designing parallel programs - communication

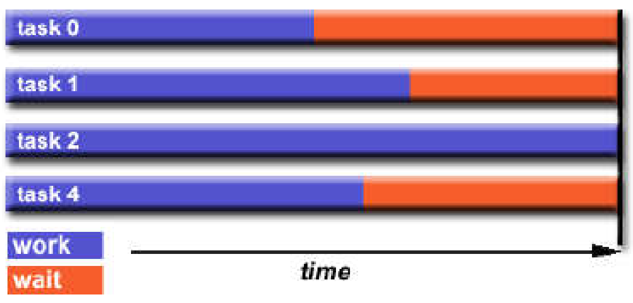
- Most parallel applications require tasks to share data with each other.
  - **Cost of communication**: Computational resources are used to package and transmit data. Requires frequently synchronization – some tasks will wait instead of doing work. Could saturate network bandwidth.
  - **Latency vs. Bandwidth**: Latency is the time it takes to send a minimal message between two tasks. Bandwidth is the amount of data that can be communicated per unit of time. Sending many small messages can cause latency to dominate communication overhead.



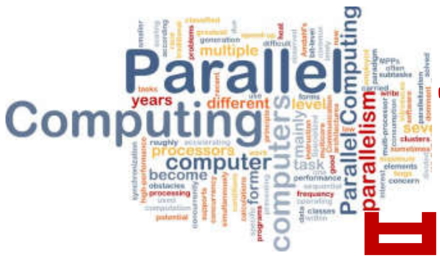
Network on a Chip !

Designing parallel programs – load balancing

- Load balancing is the practice of distributing approximately equal amount of work so that all tasks are kept busy all the time.



Analytic Measure for Parallel Algorithms



? 분석

Performance Metrics for Parallel Applications

- There are a number of metrics, the best known are:
  - **Speedup**
  - **Efficiency**
- Some laws/metrics that try to explain and assert the **potential performance of a parallel application**:
  - **Amdahl Law**
  - **Gustafson-Barsis Law**

Speedup

- Speedup is a measure of performance. It measures the ration between the sequential execution time and the parallel execution time.

$$S(p) = \frac{T(1)}{T(p)}$$

T(1) is the execution time with one processor

T(p) is the execution time with *p* processors

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
T(p)	1000	520	280	160	100
S(p)	1	1,92	3,57	6,25	10,00

Efficiency

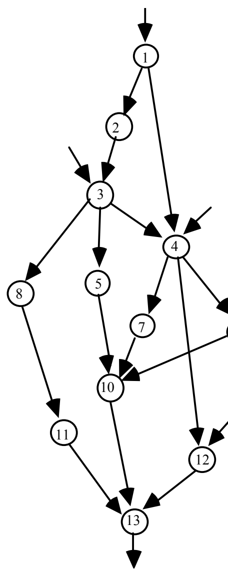
- Efficiency is a measure of the usage of the computational resources. It measures the ration between performance and the resources used to achieve that performance.

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \times T(p)}$$

S(p) is the speedup for *p* processors

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
S(p)	1	1,92	3,57	6,25	10,00
E(p)	1	0,96	0,89	0,78	0,63

Effectiveness of Parallel Processing



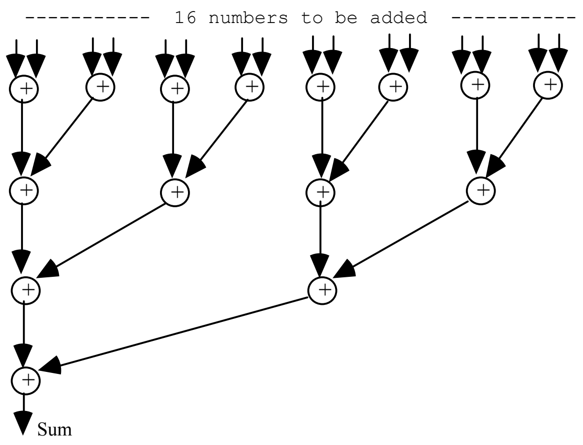
Task graph exhibiting limited inherent parallelism.

$W(1) = 13$
$T(1) = 13$
$T(\infty) = 8$

p	Number of processors
$W(p)$	Work performed by p processors
$T(p)$	Execution time with p processors $T(1) = W(1)$; $T(p) \le W(p)$
$S(p)$	Speedup = $T(1) / T(p)$
$E(p)$	Efficiency = $T(1) / [p\ T(p)]$

Reduction or Fan-in Computation

- Example: Adding 16 numbers, 8 processors, unit-time additions



Zero-time communication

$E(8) = 15 / (8 \times 4) = 47\%$
$S(8) = 15 / 4 = 3.75$

Unit-time communication

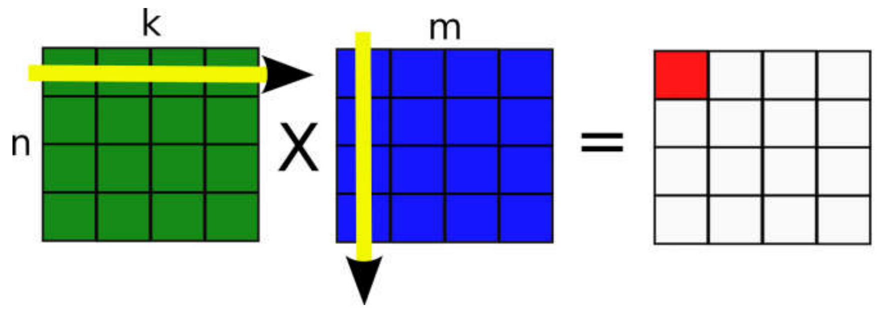
$E(8) = 15 / (8 \times 7) = 27\%$
$S(8) = 15 / 7 = 2.14$

Computation graph for finding the sum of 16 numbers .

Matrix Multiplication ?

- What about n×n matrix multiplication ?
  - Assume “+” or “×” take one time unit.

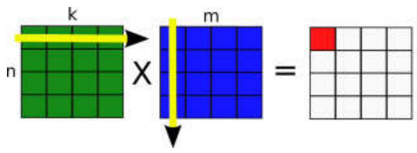
$$\sum_{i=0}^{n}\sum_{j=0}^{n} C[i,j] = \sum_{i=0}^{n}\sum_{j=0}^{n}\sum_{k=0}^{n} A[i,k] * B[k,j]$$



Matrix Multiplication ?

- What about n×n matrix multiplication ?
  - Assume “+” or “×” take one time unit.

$$\sum_{i=0}^{n}\sum_{j=0}^{n} C[i,j] = \sum_{i=0}^{n}\sum_{j=0}^{n}\sum_{k=0}^{n} A[i,k] * B[k,j]$$

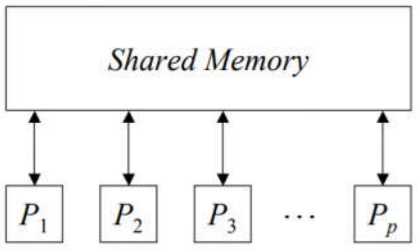


  - If we have 1 processor…
    - n × n ×($n_\times$ +$(n-1)_+$) ops = n^2 ×(2n – 1) = O(n^3)
  - If we have p processors = n ?
    - n ×($n_\times$ +$(n-1)_+$) ops ~ O(n^2)
  - If we have p processors = n^2 ?
  - If we have p processors = n^3 ?

Matrix Multiplication ?

- What about n × n matrix multiplication
  - Assume "+" or "×" take one time unit.
  - What about **binary** matrix multiplication ?

PRAM Architecture / Model



PRAM

- Parallel Random Access Machine (PRAM)
  - Natural extension of RAM: each processor is a RAM
  - Processors operate synchronously
- Model is refined for concurrent read/write capability
  - Exclusive Read Exclusive Write (EREW)
  - Concurrent Read Exclusive Write (CREW)
  - Concurrent Read Concurrent Write (CRCW)
- CRCW PRAM
  - Common CRCW: all processors must write the same value
  - Arbitrary CRCW: one of the processors succeeds in writing
  - Priority CRCW: processor with highest priority succeeds in writing

Matrix Multiplication ?

- What about n × n matrix multiplication
  - Assume "+" or "×" take one time unit.
  - What about **binary** matrix multiplication ?

```
All processor_i_j_k (1≤i, j, k≤n)
  read A[i,k] to a;
  read B[k,j] to b;
  temp = a*b;
  write '1' to C[i,j];
  write temp to C[i,j]; // with CRCW
```

Amdahl Law

- The computations performed by a parallel application are of 3 types:
  - C(seq): computations that can only be realized sequencially.
  - C(par): computations that can be realized in parallel.
  - C(com): computations related to communication / synchronization / initialization.

- Using these 3 classes, the speedup of an application can be defined as:

$$S(p) = \frac{T(1)}{T(p)} = \frac{C(seq) + C(par)}{C(seq) + \boxed{\frac{C(par)}{p}} + C(com)}$$

Amdahl Law

- Since C(com) ≥ 0 then:

$$S(p) \le \frac{C(seq) + C(par)}{C(seq) + \frac{C(par)}{p}}$$

- If 'f' is the ***fraction of the computation*** *that can only be realized* ***sequentially***, then:

$$\boxed{f = \frac{C(seq)}{C(seq) + C(par)}} \quad \text{and} \quad S(p) \le \frac{\frac{C(seq)}{f}}{C(seq) + \frac{C(seq) \times \left(\frac{1}{f} - 1\right)}{p}}$$

Amdahl Law

- Simplifying:

$$S(p) \le \frac{\frac{C(seq)}{f}}{C(seq) + \frac{C(seq) \times \left(\frac{1}{f} - 1\right)}{p}}$$

$$\Rightarrow \quad S(p) \le \frac{\frac{1}{f}}{1 + \frac{\frac{1}{f} - 1}{p}} \quad \Rightarrow \quad S(p) \le \frac{1}{f + \frac{1 - f}{p}}$$

Amdahl Law

- Let 0 ≤ f ≤ 1 be the computation fraction that can only be realized sequentially.
- The Amdahl law tells us that the ***maximum speedup that a parallel application can attain with p processors*** is:

$$S(p) \leq \frac{1}{f + \frac{1-f}{p}}$$

- The Amdahl law can also be used to determine ***the limit of maximum speedup*** that a determined application can achieve regardless of the number of processors used.

Amdahl Law

- Suppose one wants to determine if it is advantageous to develop a parallel version of a certain sequential application. Through experimentation, it was verified that 90% of the execution time is spent in procedures that may be parallelizable. What is the maximum speedup that can be achieved with a parallel version of the problem executing on 8 processors?

$$S(p) \leq \frac{1}{0{,}1 + \frac{1-0{,}1}{8}} \approx 4{,}71$$

- And the limit of the maximum speedup that can be attained?

$$\lim_{p \to \infty} \frac{1}{0{,}1 + \frac{1-0{,}1}{p}} = 10$$

Limitations of the Amdahl Law

- The Amdahl law ignores the cost with communication / synchronization operations associated to the introduction of parallelism in an application. For this reason, the Amdahl law can result in predictions not very realistic for certain problems.
- Consider a parallel application, with complexity $O(n^2)$, whose execution pattern is the following, where n is the size of the problem:
  - Execution time of the sequential part (input and output of data):
    - 18.000 + n
  - Execution time of the parallel part: $n^2 / 100$
  - Total communication/synchronization points per processor: $\lceil \log n \rceil$
  - Execution time due to communication/synchronization (n=10.000) : $1000 * \lceil \log p \rceil + n/p$

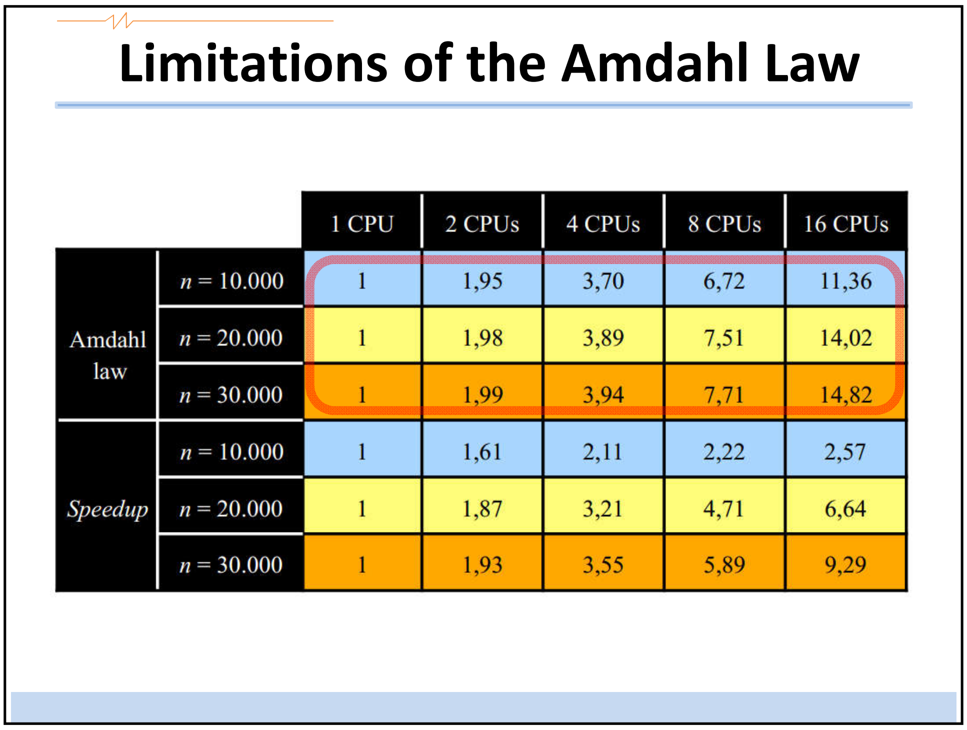
Limitations of the Amdahl Law

- What is the maximum speedup attainable?
- Using Amdahl law:

$$f = \frac{18.000 + n}{18.000 + n + \frac{n^2}{100}} \quad \text{and} \quad S(p) \le \frac{18.000 + n + \frac{n^2}{100}}{18.000 + n + \frac{n^2}{p \times 100}}$$

- Using the speedup measure:

$$S(p) = \frac{18.000 + n + \frac{n^2}{100}}{18.000 + n + \frac{n^2}{p \times 100} + \lceil \log n \rceil \times \left(10.000 \times \lceil \log p \rceil + \frac{n}{10} \right)}$$



Limitations of the Amdahl Law

		1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
Amdahl law	n = 10.000	1	1,95	3,70	6,72	11,36
	n = 20.000	1	1,98	3,89	7,51	14,02
	n = 30.000	1	1,99	3,94	7,71	14,82
Speedup	n = 10.000	1	1,61	2,11	2,22	2,57
	n = 20.000	1	1,87	3,21	4,71	6,64
	n = 30.000	1	1,93	3,55	5,89	9,29