Parallel & Distributed Systems

Dr. Omar Usman Khan
omar.khan@nu.edu.pk

Department of Computer Science
National University of Computer & Emerging Sciences, Peshawar

October 4, 2021

Syllabus

1. Introduction
   - Brief Overview
   - Hardware Models

About the Course

Learning Outcomes

CLO1	Have a good understanding of concurrent and distributed systems
CLO2	Be able to write programs for Concurrent systems
CLO3	Be able to write programs for Distributed systems
CLO4	Be able to write programs for specialized Performance Hardwares

Marks Breakdown

- Sessional I: 15%
- Sessional II: 15%
- Final Examination: 50%
- Assignments: 20% (including Takehome Lab Activities)

About the Course (cont.)

Assignment Instructions

- Will accept assignments after last date, provided said assignment for entire class is not marked.
- Parts of assignment may appear in examinations. If not, then similarity checks may be applied on submitted assignments.

- HPC Setup with support for various parallel and distributed computing frameworks will be made available to all students for duration of semester (and beyond upon request)

Generating Public Key for Assignments

- On Windows, Use https://winscp.net/eng/docs/ui_puttygen
- On Linux, use **ssh-keygen** command
- Share generated key with me by email.

Where are We

- Clock rates 40 MHz (MIPS R3000 1988) $\rightarrow$ 4.0 GHz (Intel Core-i7-4790K 2015), 4.4 GHz (Intel Xeon X5698 2015)
- Higher processor speed $\propto$ More **heat dissipated**
- Transistor has reached size of 32 nm (Generation 3 Core-i7). **Size limit**: How much more smaller can it get?
- What is a single clock cycle worth?
  - **xchg**: Exchange values in two registers
  - **push**: Push operation using registers
  - **pop**: Pop operation using registers
  - **add**,**sub**: 3 additions/subtractions involving registers
  - **cmp**: 3 comparisons involving registers
  - **mul**: half a fp multiplication involving registers
- Can we get a 8 GHz clock frequency?
- What will happen if we reach a 8 GHz clock frequency?
- Support for executing multiple instructions per clock cycle, fast cache technologies, superscalar architectures ...
- Dramatic increase in Floating Point Operations per Second (FLOPs)

Where are We (cont.)

Figure 1: AMD breaks Guinness World Record for fastest processor frequency at 8.429 gigahertz (2011)

Where are We (cont.)

A question to think about

- Akram has a 12 GHz single core processor. Akbar has a quad-core 3 GHz multiprocessor. Which one is faster?

Premise: Top 10 Supercomputers

Application Areas: Astrophysics, financial markets (portfolio management), scientific research (computational modelling, biology, chemistry, physics, mathematics, geometry, graphics, signal processing, etc.), . . . Will be covered again in detail

Rank	Site	System	Cores	TFLOPs	Power (kWh)
1	Guangzhou, China	Tianhe-2: Intel Xeon	3,120,000	54,902	17,808
2	Oak Ridge Lab, USA	Titan: Cray Opteron, NVIDIA K20	560,640	27,112	8,209
3	DoE, USA	Sequoaia:, IBM BlueGene/Q	1,572,864	20,132	7,890
4	RIKEN Ins., Japan	K-Computer: Fujitsu SPARC64	705,024	11,280	12,660
5	DoE, USA	Mira: IBM BlueGene/Q	786,432	10,066.3	3,945
6	Swiss S.Comp., Switzerland	Cray, Intel Xeon, NVIDIA K20	115,984	7,788	2,325
7	Univ. Texas, USA	Stampede: Intel Xeon	462,462	8,520	4510
8	Forschchung Juelich, Germany	JUQUEEN: IBM BlueGene/Q	458,752	5,872	2,301
9	DoE, USA	Vulcan: IBM BlueGene	393,216	5,033	1,972
10	US Govt	Cray, Intel Xeon, NVIDIA K40	72,800	6,131	1,499
-	Your Home	Intel Core-i7	4	.026	0.3
-	Your Home	NVIDIA K40	2,880	4.29	0.3

Table 1: Top 10 Supercomputers: February 2015, top500.org

Premise: Top 10 Supercomputers (cont.)

Figure 2: China's Tianhe-2 . . . The Heaven River, The Milky Way, developed and operated by 1,300 scientists and engineers. Plans to double the computing power were stopped by the US government when they rejected Intel's application of granting an export license for selling CPU's and motherboards to the chinese government. Total of 3.12 Million Cores, and 1,375 TB RAM, runs Kylin Linux, and Slurm job management

Premise: Top 10 Supercomputers (cont.)

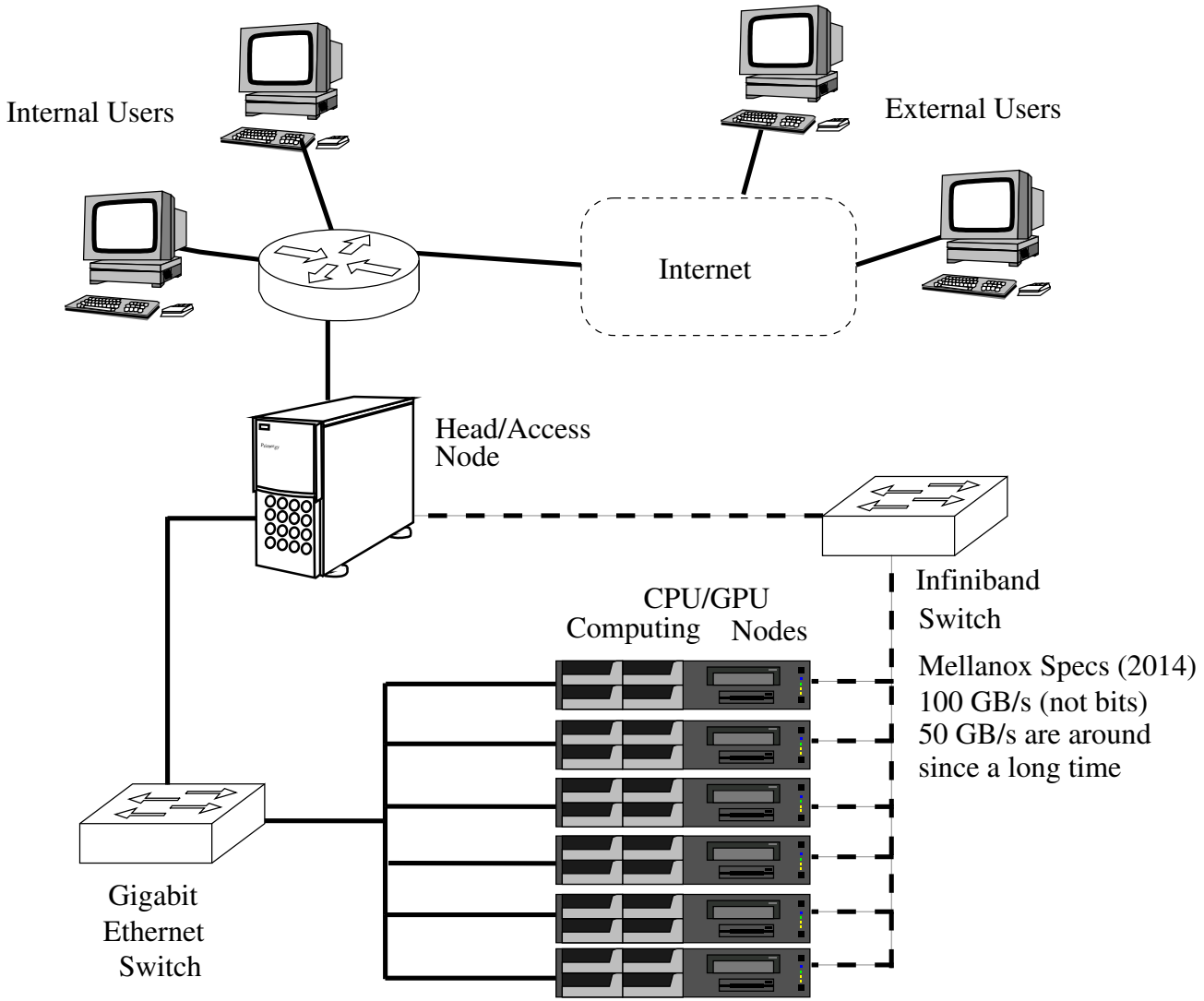


Figure 3: Setup of a typical HPC facility

Top 10 Supercomputers over Time

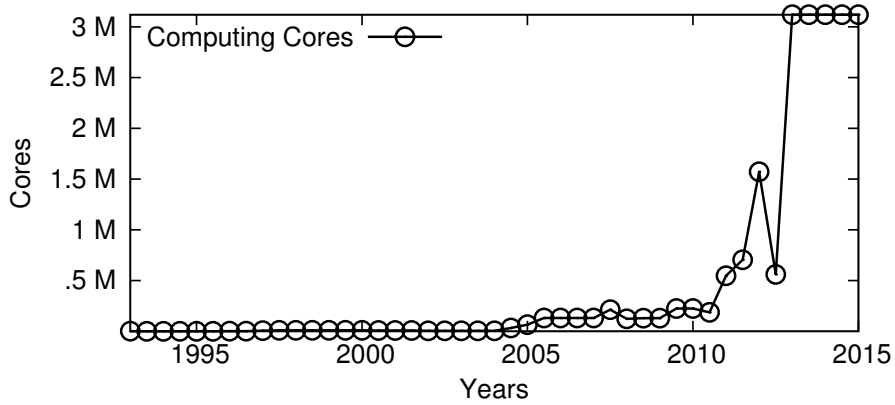
Year	Cores	FLOPs
2006	131,072	367,000,000,000,000 (367 TFLOP)
2007	212,992	596,000,000,000,000 (596 TFLOP)
2008	129,600	1,456,700,000,000,000 (1.4 PFLOP)
2009	224,162	2,331,000,000,000,000 (2.3 PFLOP)
2010	224,162	4,701,000,000,000,000 (4.7 PFLOP)
2011	705,024	11,280,400,000,000,000 (11.2 PFLOP)
2012	1,572,864	27,112,500,000,000,000 (27.1 PFLOP)
2013	3,120,000	54,902,400,000,000,000 (54.9 PFLOP)
2014	3,120,000	54,902,400,000,000,000 (54.9 PFLOP)
2015	3,120,000	54,902,400,000,000,000 (54.9 PFLOP)

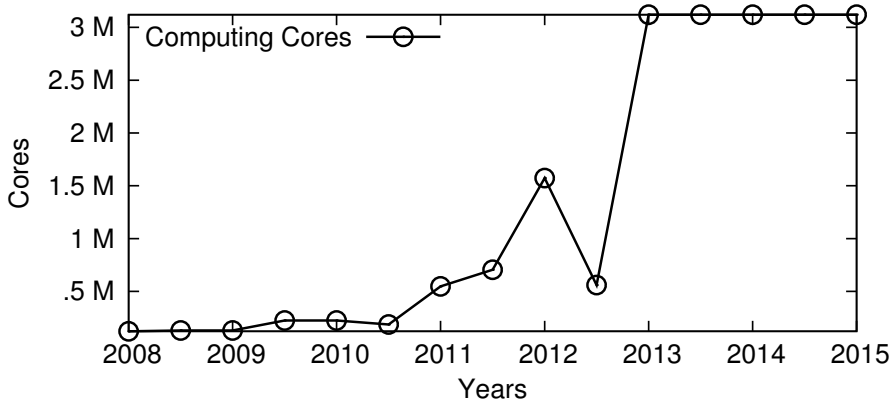
Table 2: Timeline Table of the Top Super-Computer, top500.org

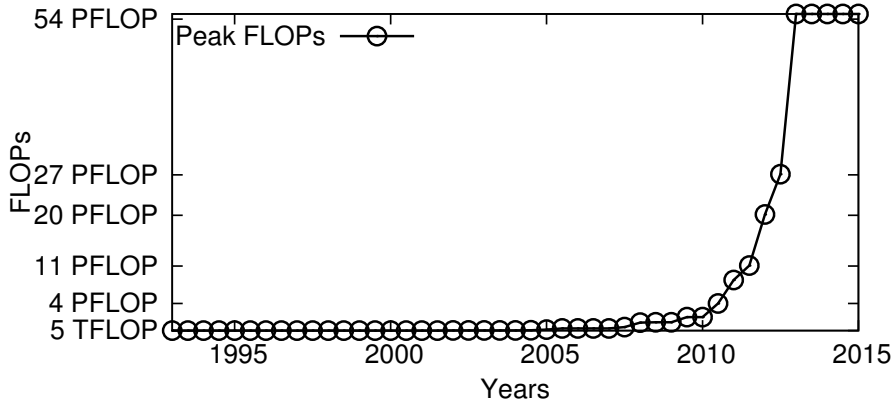
atto	10^{-18}		exa	10^{18}
femto	10^{-15}		peta	10^{15}
pico	10^{-12}		tera	10^{12}
nano	10^{-9}		giga	10^{9}
micro	10^{-6}		mega	10^{6}
milli	10^{-3}		kilo	10^{3}

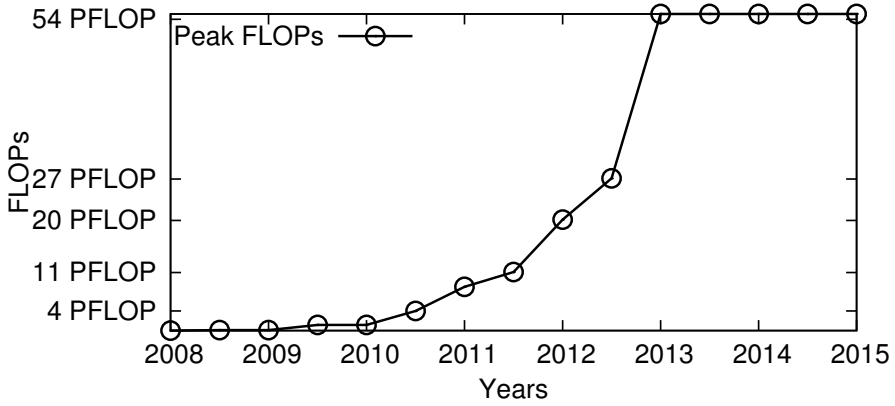
Table 3: Scales

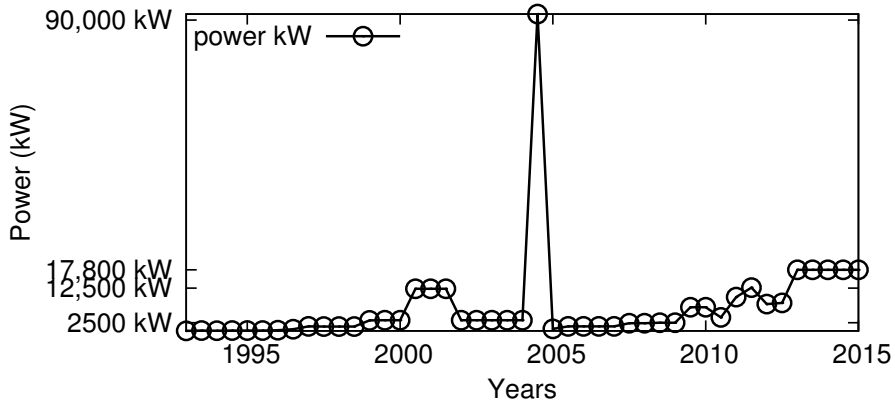
Top 10 Supercomputers over Time (cont.)

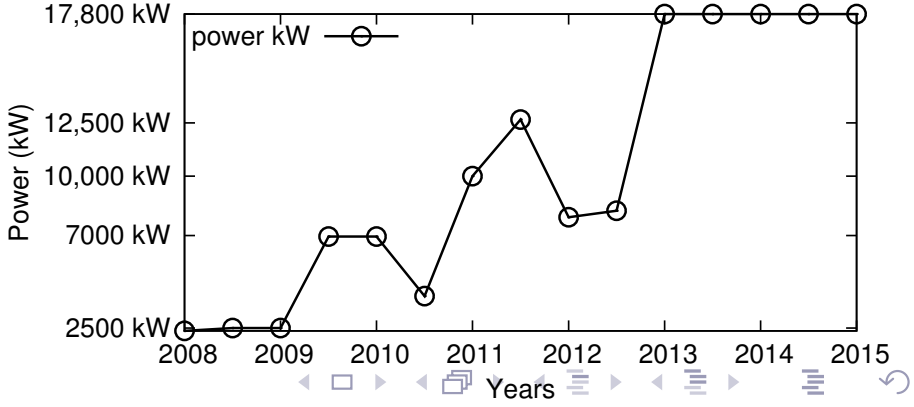












Why do we Want More than 3 Million Cores (or more) ??

- Because some computations are not possible using current hardware. E.g.,
  - Modelling of Global Climate over decades
  - Atomic scale simulations of large materials
  - Aerodynamics simulation of an entire aircraft
  - Modelling of fine resolution air/liquid turbulence

Example Calculations

$10\mu m^3$ volume discretized into cells of size $5nm^3$.

$$\frac{10 \times 10 \times 10\mu m}{5 \times 5 \times 5nm} = \frac{1000\mu m}{125 \times 10^{-9}\mu m} = 8 \times 10^9 \tag{1}$$

RAM required (M) $= 8 \times 10^9 \times sizeof\,(double) = 6.4 \times 10^{10}$ bytes = **59.6 GB**. Assuming each cell takes 1 ms and complexity of $O\,(n)$, Time required (t) = $8 \times 10^9 \times 10^{-3} / (60 \times 60 \times 24 \times 30) =$ **3.08 months**. **This is too much.** Changing cell size to $15nm^3$ gives us.

$$\frac{10 \times 10 \times 10\mu m}{15 \times 15 \times 15nm} = \frac{1000\mu m}{3375 \times 10^{-9}\mu m} = 2.96 \times 10^8 \tag{2}$$

RAM required (M) $= 2.96 \times 10^8 \times sizeof\,(double) =$ **2.2 GB**. Time required (t) = $2.96 \times 10^8 \times 10^{-3} / (60 \times 60 \times 24) =$ **3.42 days**. **This is acceptible**.

Moore's Law, 1965

A prediction that would define the pace of digital revolution. Many interpretations:

- Computing would increase in **power** exponentially
- Computing would decrease in relative **cost** exponentially
- **Transistory density** will double every year (revised to double every 18 months)

Cramming more components onto integrated circuits

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip

By Gordon E. Moore

Director, Research and Development Laboratories, Fairchild division of Fairchild Camera and Instrument Corp.

The future of integrated electronics is the future of electronics itself. The advantages of integration will bring about a proliferation of electronics, pushing this science into many new areas.

Integrated circuits will lead to such wonders as home computers—or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment. The electronic wristwatch needs only a display to be feasible today.

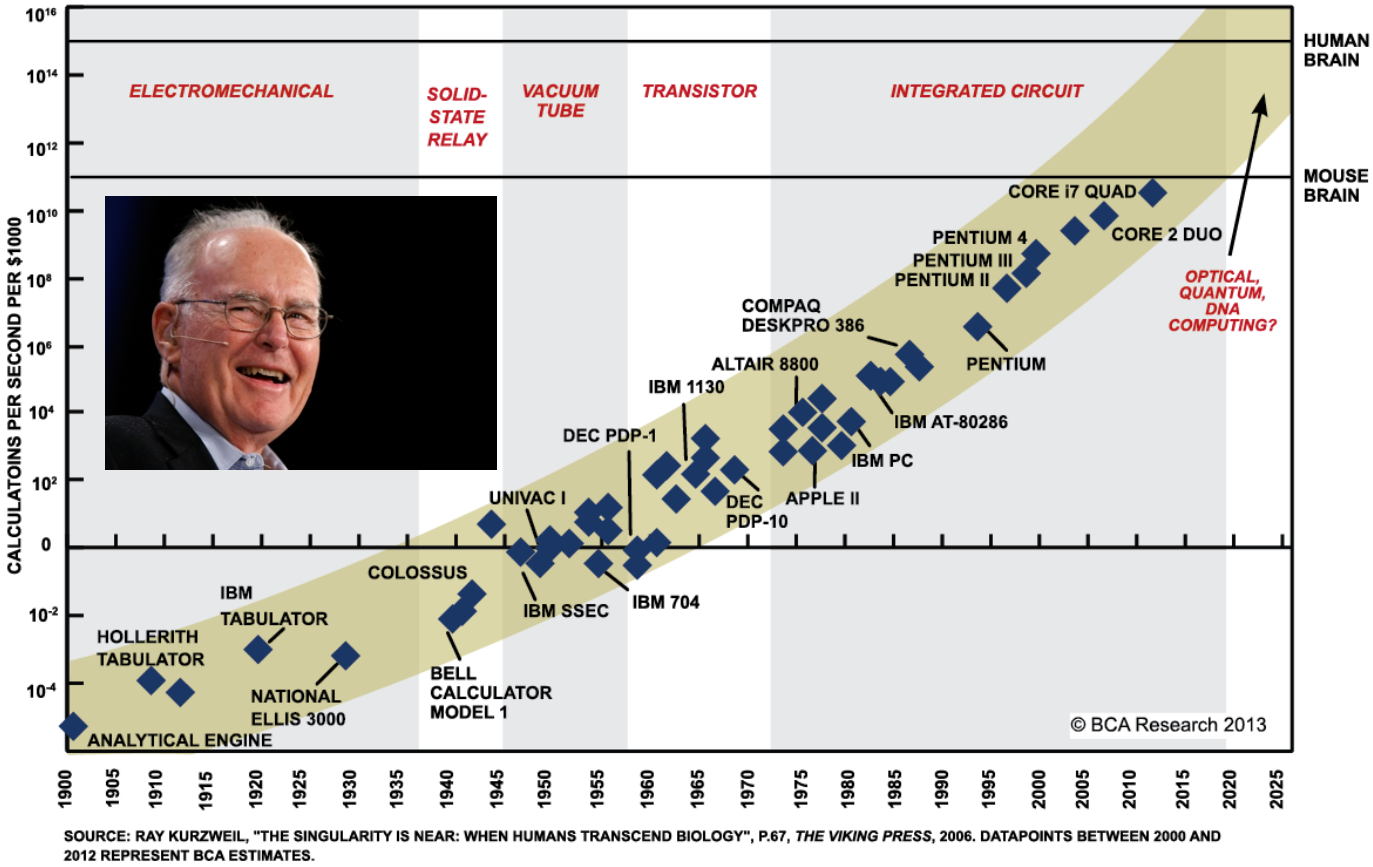
But the biggest potential lies in the production of large systems. In telephone communications, integrated circuits in digital filters will separate channels on multiplex equipment. Integrated circuits will also switch telephone circuits and perform data processing.

Computers will be more powerful, and will be organized in completely different ways. For example, memories built of integrated electronics may be distributed throughout the

The author

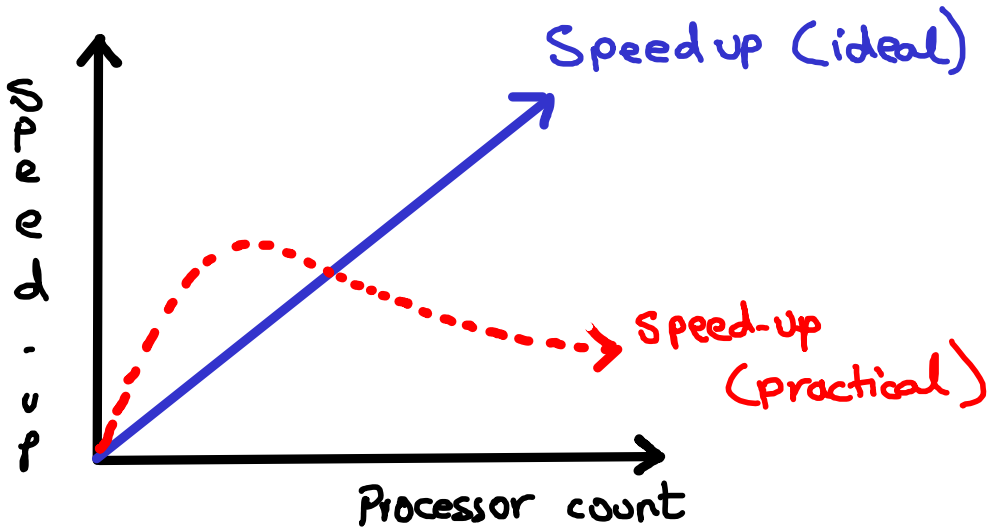
Dr. Gordon E. Moore is one of the new breed of electronic engineers, schooled in the physical sciences rather than in electronics. He earned a B.S. degree in chemistry from the University of California and a Ph.D. degree in physical chemistry from the California Institute of Technology. He was one of the founders of Fairchild Semiconductor and has been director of the research and development laboratories since 1959.

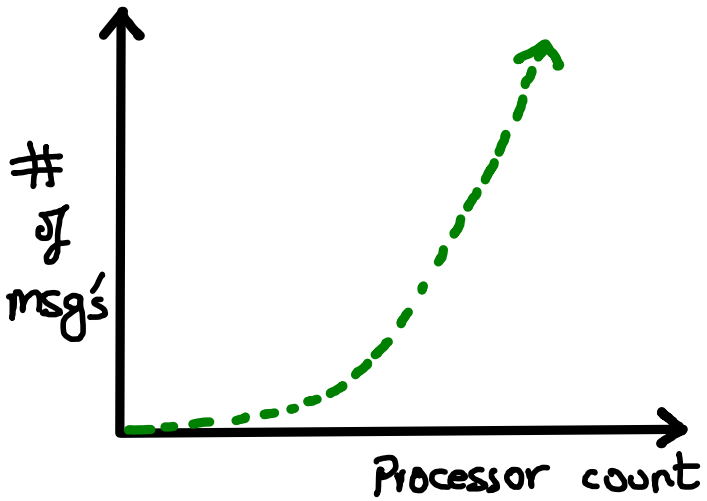
Electronics, Volume 38, Number 8, April 19, 1965



SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

Premise: Can software exploit this hardware for speed??





- Why ideal speedup is not possible: data transfer (through message exchanges), I/O bottlenecks, race conditions, dependencies, contention (critical section), load balancing, deadlocks, synchronization, node failures, **lazy programmers, programmers lacking skills in concurrent and distributed programming** . . .

Theoretical Speedup

n Problem Size

p Processors Quantity

$\sigma(n)$ Sequential portion of computation

$\phi(n)$ Parallelizable portion of computation

$\kappa(n,p)$ Parallelization overhead

$T(n,1)$ Sequential Execution time $T(n,1) = \sigma(n) + \phi(n)$

$T(n,p)$ Parallel Execution time $T(n,p) = \sigma(n) + \kappa(n,p) + \frac{\phi(n)}{p}$

$s(n,p)$ Speedup $s(n,p) = \frac{T(n,1)}{T(n,p)}$

$\epsilon(n,p)$ Efficiency $\epsilon = \frac{T(n,1)}{pT(n,p)}$

Theoretical Speedup (cont.)

Amdahl's Law

Ignoring overhead, speedup is:

$$s(n,p) = \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p}} \tag{3}$$

1. Introduction

- Brief Overview
- Hardware Models

Serial vs Parallel vs Distributed

Serial Computing Systems

- Single control mechanism, determines the next instruction to be executed.
- All data operations are fetched from memory one at a time.
- Speedup can be introduced using:
  - Instruction Caching (large caches with low latency)
  - Pipelining (super-scalar architectures)
  - Overclocking
  - Overlapping of I/O and computation using Direct Memory Access (DMA)
- **Note: Two serial programs can execute concurrently (multi-taskng)**.

- **Multi-Processing**: Multiple CPU cores executing more than 1 program at a given time
- **Multi-Tasking**: A single CPU core "appearing" to be executing more than 1 program at a given time
- **Multi-Threading**: The presence of more than 1 "threads" in a single program.

Serial vs Parallel vs Distributed (cont.)

Parallel Computing Systems

- Several processors that are located within a small distance of each other
- Their main purpose is to perform a computation task jointly
- Communication between processors, if required, is reliable, fast, and predictable.

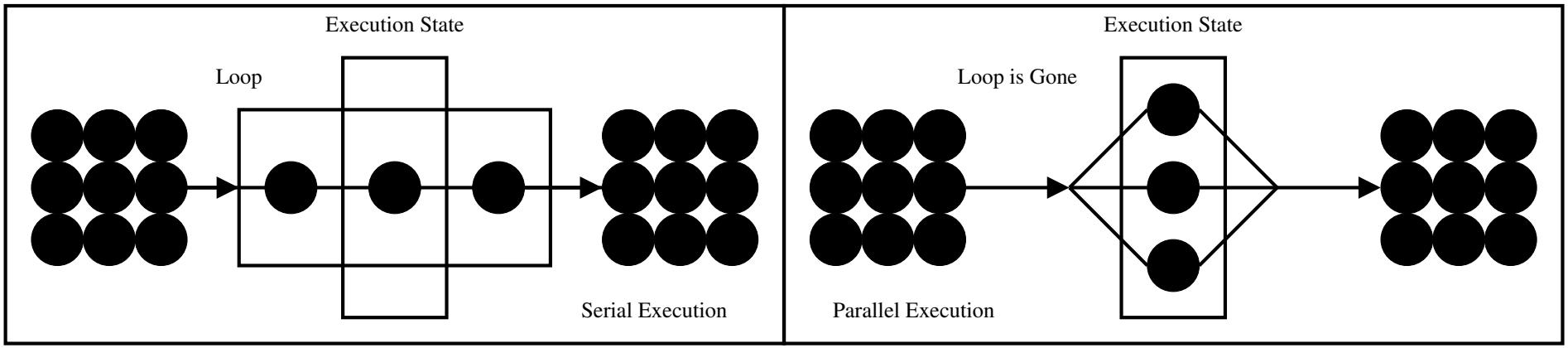


Figure 4: Difference between serial and parallel execution

Serial vs Parallel vs Distributed (cont.)

Distributed Computing Systems

- Processors may be far apart (are loosely coupled and usually independent of others)
- Challenges
  - Communication latency between processors is large and unpredictable
  - Communication links may be unreliable
  - Topology of system may undergo changes.

Serial vs Parallel vs Distributed (cont.)

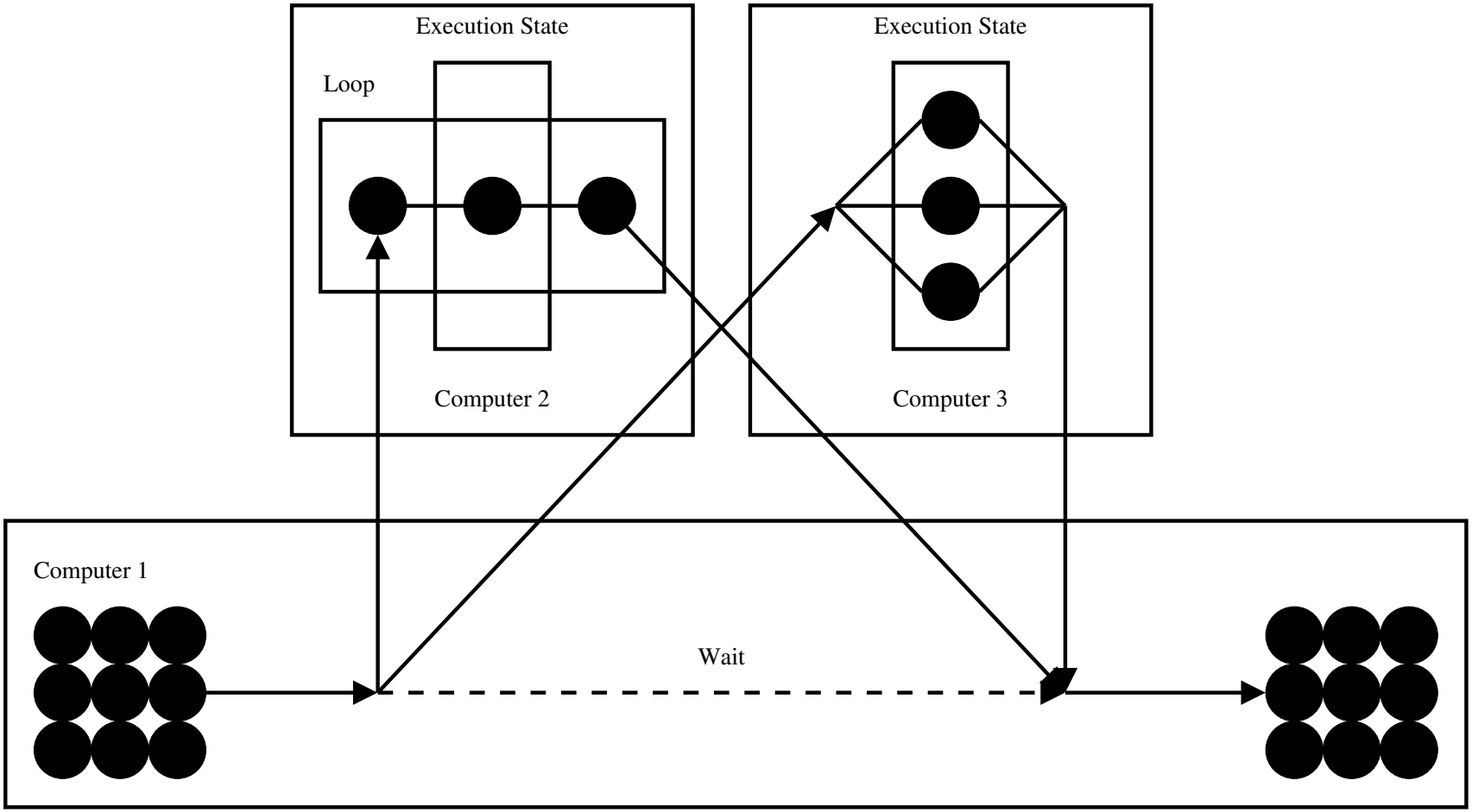


Figure 5: Model of execution on a distributed system (non-client/server). This shows a hybrid. But usually developers don't like these setups.

What is there in Parallel, that is not available in Serial?

Some Examples . . .

- **Task Allocation**: Breakdown of total workload into smaller tasks, such that each smaller task is assigned to a different processor
- **Interim Communication**: Since each processor is working on independent tasks, they may need to communicate outcome of computation with other processors.
  - Is communication automatic? or defined by programmer?
  - Depends on type of multi-threading tool that is being used
- **Synchronization**: A waiting interval by a processor for certain tasks
  - E.g., arrival of data (before computation on it can be performed, or before bring printed).
  - This would have positive effect on accuracy but negative effect on performance.
- . . .

Applications and Examples

General Purpose Programs and Purposes

- Word processor (spell check, auto save, . . .)
- Spreadsheet (automatic background recalculations after cell edits)
- Operating system (User threads, Kernel threads, . . .)
- Server (multi client handling, database servers serving millions of transactions per second)
- GUI's (Event listeners)
- Video games (rendering of animation, taking care of physics, artificial intelligence, network connections, user I/O, . . .))
- Web browser (tabs, multi-segment download accelerators, . . .)
- Search Engines (Information searching, retrieval)
- Encoding/Decoding of video (compression/decompression)
- . . .

Applications and Examples (cont.)

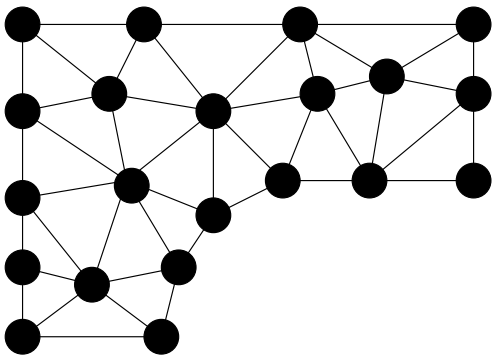
Scientific Purposes

- Mathematical problems (Ordinary differential equations, partial differential equations, linear algebra, numerical analysis)
  - Climate modelling and weather prediction
  - Fluid dynamics (video games, simulation of dam bursts, tsunamies, etc.)
  - Computational Biology, Finance, Physics, Chemistry
  - Machine learning, statistical methods
  - Image processing (medical images, spectral images, satellite images)
- Finding Aliens, Finding cure for Alzheimers, Cancers, Parkinson's, ...

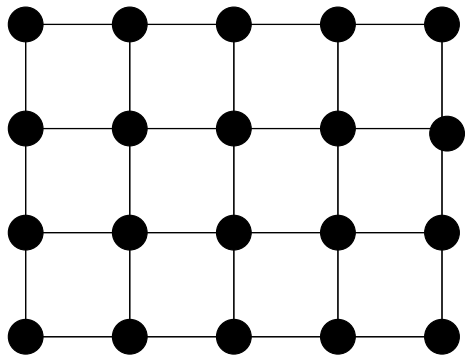
Figure 6: The Folding@Home project by Stanford University

Difference between both types of Applications

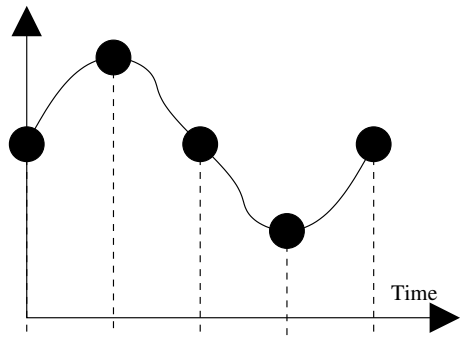
- General Purpose Applications: Threads are usually involved in doing **different things**. This is known as **task-parallelism**.
- Scientific Applications: Threads are usually involved in doing **similar things**, but at different locations. This is known as **data-parallelism**
- Different locations? Take a problem and perform both **spatial decomposition**, as well as **time decomposition**.
- Each decomposed location (involving predominantly local interactions) will be handled ideally by a single processor. If it involves non-local interactions, then processors would need to communicate.
- What level of decomposition would be required ?? A question answered by concerns of cost, speed, storage memory, functional requirements, etc.



Non–Uniform Spatial Decomposition

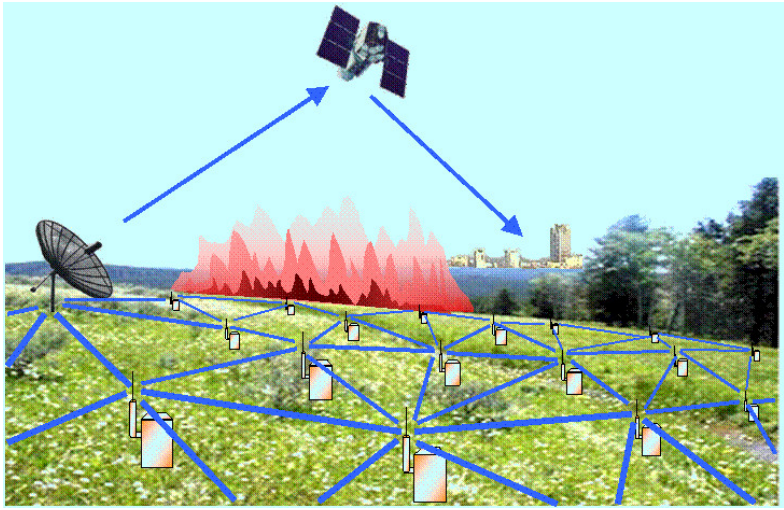


Uniform Spatial Decomposition



Uniform Time Decomposition

Applications



← Fig: Sensor Networks Deployment

- Approach-1: Client/Server Distributed Models
- Approach-2: Decentralized Distributed Models (P2P, WSN, MANET)

- Functions: Information Acquisition, Extraction, Aggregation and Control in geographically distributed systems
- E.g., sensor networks, where geographically distributed sensors **acquire** local information). This information is communicated to a control station through multiple hops. At each hop, information may be **extracted**, **aggregated** (to remove redundancies), and repackaged before forwarded. The control station ultimately recieves the packet and performs **control** operations (e.g., fire control, smart cooling, etc.)
- Concerns: unreliable communication, device failures, cost, speed, deployment, energy constraints

Examples of Processors

- Hundreds of architectures, dozens in main-stream, hundreds now obsolete
- Many ways of classifying these architectures. The most simple classification is those that are "parallel", and those that are "distributed"

Array Processors (Vector Processors)

- Array of coupled processors (with nearest neighbor inter-connects)
- Most array processors perform "combined" or "group" execution.
- Ideal for spatial decomposed problems, mage processing, scientific computing, etc. (not good for task-parallelism)
- A host processor may monitor progress on the processor array and serves as a "control processor"
- Distributed/shared memory array processor possible
- More about this in Hardware Classifications (Flynn)

Examples of Processors (cont.)

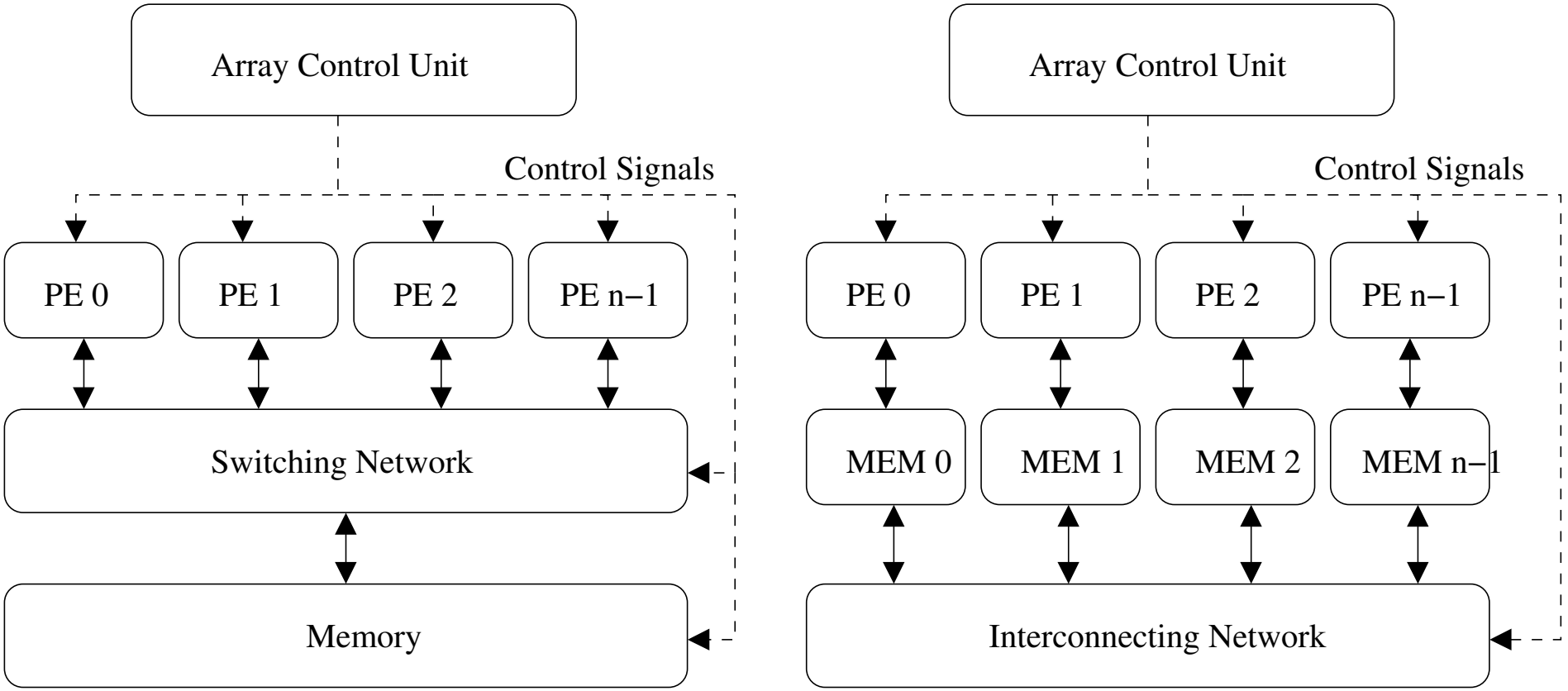


Figure 7: Array Processors (left) Shared Memory (right) distributed memory (IMG: Roland N Ibbett & Nigel P Topham, 1996) (Key: PE = Processing Element)

Examples of Processors (cont.)

Multiprocessors

- Loosely coupled processors, where each processor has more control on what to do and when to do.
- Well suited for task parallel problems (general purpose multi-threaded applications)
- Most common type is known as "symmetric multiprocessor system"

Systolic Arrays

- Makes use of "Very Large Scale Integration" (VLSI) Technology, where all computation elements are placed on a single chip.
- Processors are designed for a single purpose only and show fixed behaviour (code is embedded in system hardware)
- E.g., solving a linear system of equations, performing a fast Fourier transform
- Data "moves" across each processor (must be aligned before execution can commence)

Examples of Processors (cont.)

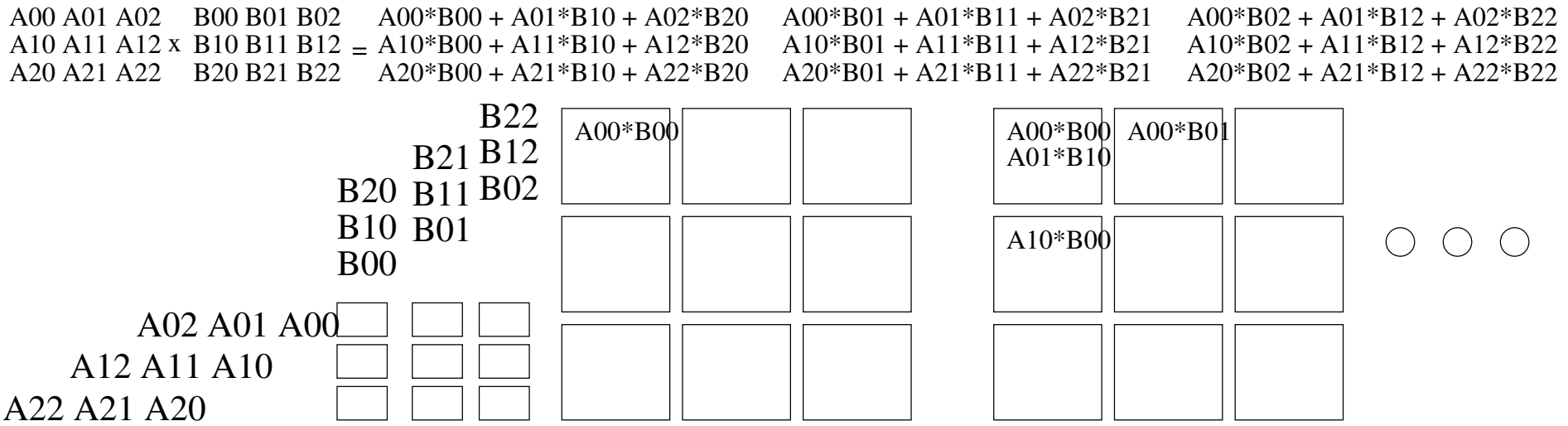


Figure 8: Movement of Data across systolic array for Matrix Multiplication

Parameters Describing Types of Parallel and Distributed Computers

Number of Processors

- Thousands of Processors: Massively Parallel Systems
- Limited No. of Processors: Coarse Grained Parallel Systems
- Clock frequency of massively parallel systems is usually "less" as compared to coarse-grained ones.

Presence/Absence of Global Control Mechanism

- Who is controlling parallel processing ???
- Global control mechanism in Massively Parallel Systems **starts** a program, then **loads** data onto parallel processors, and then **allows** each parallel processor to run on its own, before finally **collecting** data back from each processor
- In coarse grained parallel systems, no central control exists and all processor collaborate in an ad-hoc manner.

Parameters Describing Types of Parallel and Distributed Computers (cont.)

Synchronous vs Asynchronous Operations

- Is there a common global clock available?
- If available, all processors will "synchronize" their execution according to the global clock. (Synchronous Operation)
- Synchronous operations help remove race conditions, but on the other hand, may introduce overhead and delays.

Parameters Describing Types of Parallel and Distributed Computers (cont.)

Processor Interconnects: How Processors Exchange Information

Shared Memory

- Global shared memory accessible by all processors
- Method of Communication: 1 Processor writes to shared memory, the others read from it
- Access latency is very low
- Simultaneous access to shared memory may lead to race conditions (To rectify, use hardware switching Note: software solution like semaphores etc. cannot be used here)

Message Passing

- Processors communicate through messages using an interconnecting network
- Each processor has private local memory
- Ideal interconnecting network would have a link between each and every processor (but this is expensive, hence simpler interconnects such as shared bus can be used)

Parameters Describing Types of Parallel and Distributed Computers (cont.)

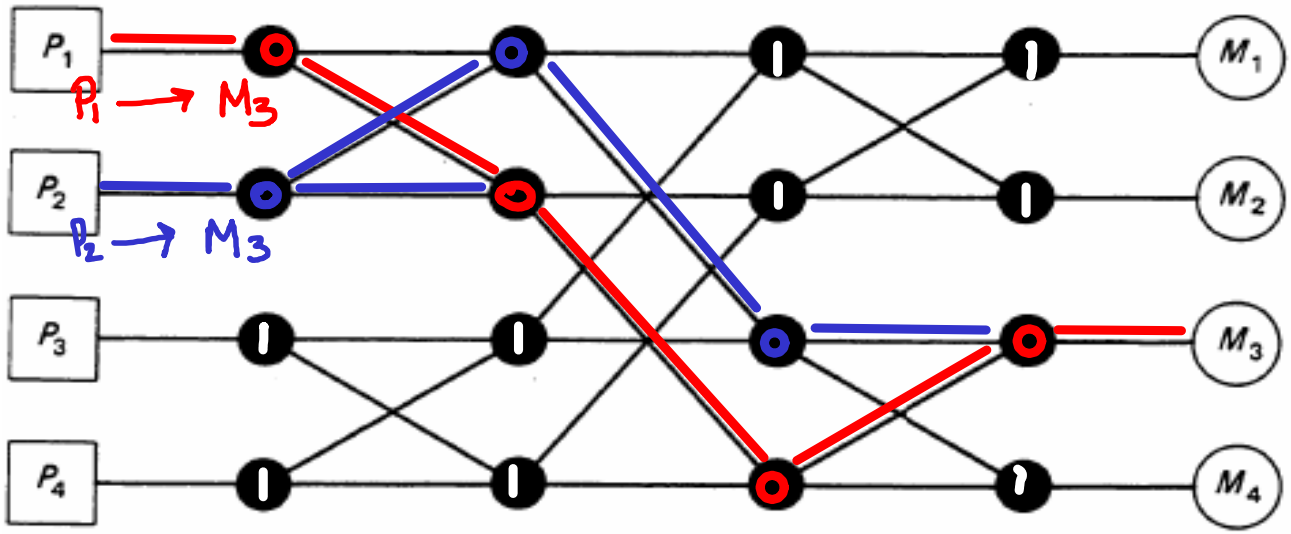


Figure 9: Switching System, with all switches set to 1. p_1 access m_3 and sets the switches to 0 along the access path. p_2 wants to access m_3 but will have to find an access path where the switch is 1. Number of switches depends on number of processors and memory locations, and type of switching network. The depth of switch network would ideally be small (usually at least $\log_2 p$)

Parameters Describing Types of Parallel and Distributed Computers (cont.)

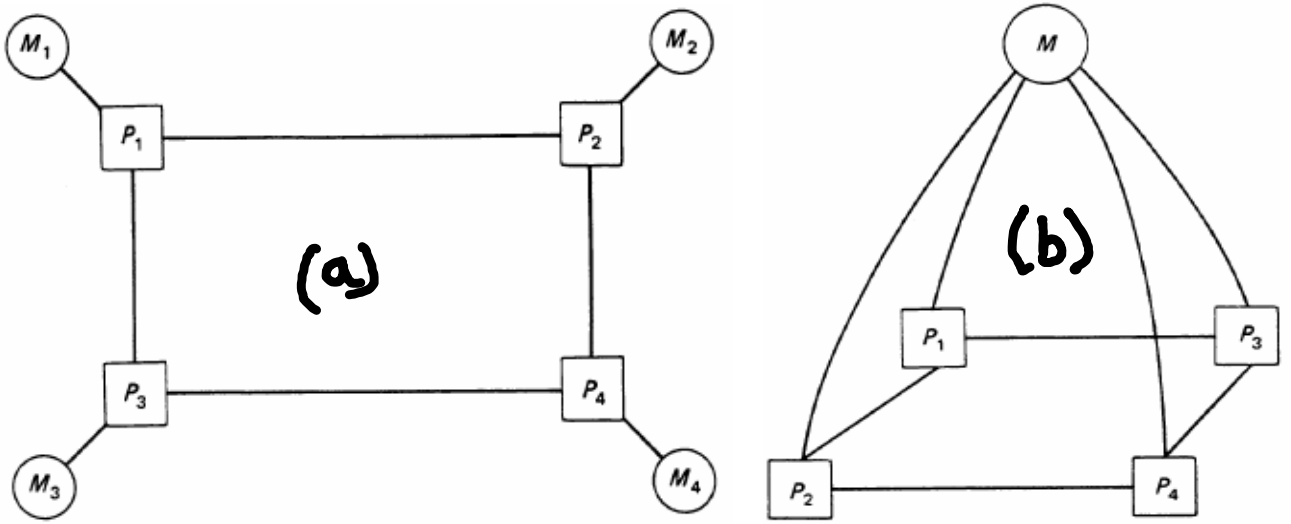


Figure 10: (a) A simple message passing system with local memories assigned to each processor. Processors are interconnected using a ring network. (b) A hybrid system where processors can communicate directly using the ring network, or through the single shared global memory. Global memory contention is ignored in the illustration.

Flynn-Johnson Taxonomy (Classification)

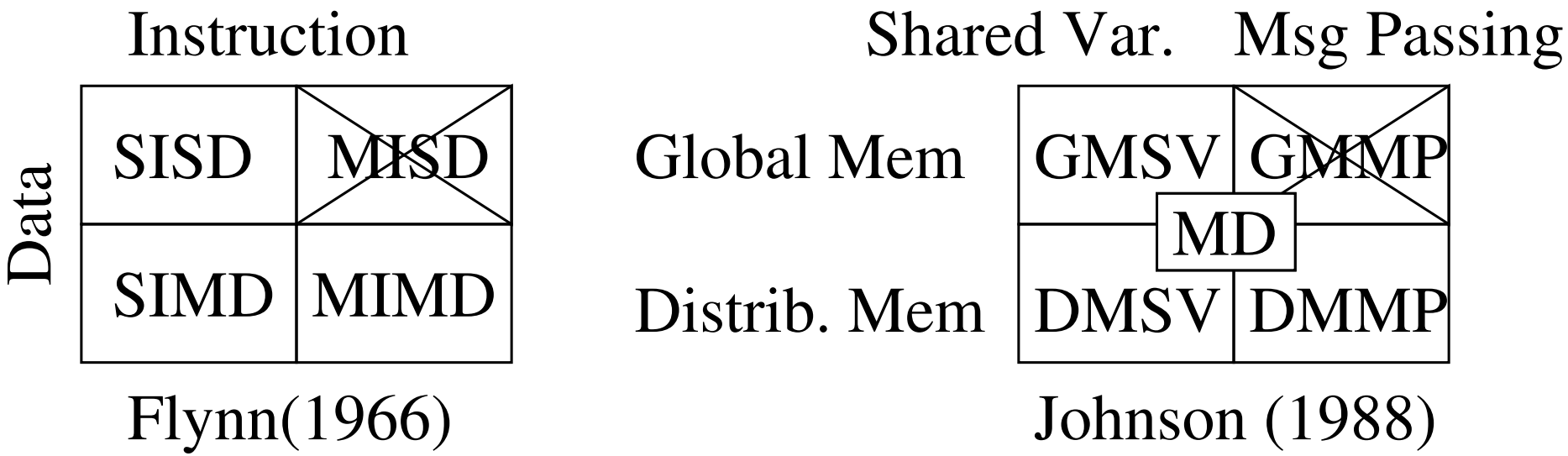


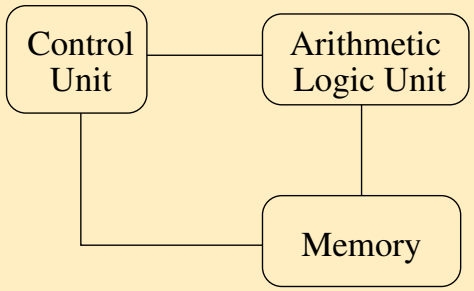
Figure 11: (l) Flynn's Classification (1966) based on instruction and data streams, and (r) Johnson's Classification (1988) based on memory and communication structures

- **Instruction Stream**: Sequence of Instructions
- **Data Stream**: Sequence of Data

Flynn-Johnson Taxonomy (Classification) (cont.)

SISD

- Uni-Processor architectures
- Logic of program is sequential (sequential flow of instructions)
- Concurrent execution using context-switching can be possible
- *It's still SISD if you follow sequential programming on multi-core machines*



Typical Fetch/Execute Cycle: (1) Instruction Fetch, (2) Ins. Decode, (3) Operand Address Calculation, (4) Operand Fetch, (5) Execute, (6) Store Result

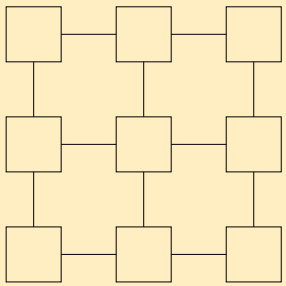
MISD

- Not commercially implemented
- E.g. multiple processors working on decrypting a single encrypted object (using different decrypting algorithms.

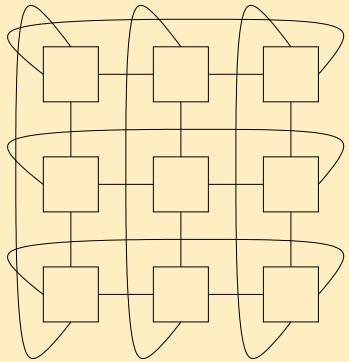
Flynn-Johnson Taxonomy (Classification) (cont.)

SIMD/MIMD

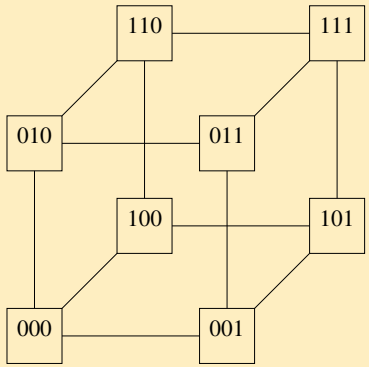
- Array/Streaming/Vector Processors interconnected using nearest-neighbour inter-connects.
- Processing elements contain only "Arithmetic Unit" instead of "Arithmetic and Logic Unit" (hence combined execution)
- Possible to disable some processing elements, if not needed
- Possible Issues
  - Data Alignment Issues
  - Conditional execution



Simple Nearest Neighbor Inter–connects



3 by 3 Nearest Neighbor Inter–connects



3–cube Inter–connects

Flynn-Johnson Taxonomy (Classification)

Johnson's Taxonomy

- **GMSV** Symmetric Multi-Processors (OpenMP, PThreads)
- **DMMP** Distributed systems (If simulated: Omnet++, NS2/3, etc.), NUMA based Cluster systems (OpenMPI, RPC), GPU Computing (OpenCL, CUDA), HPC based systems
- **DMSV** Cloud Model: Memory is distributed but access is through same address space (Hadoop, MapReduce)

Threads

- Smallest execution path
- Runs within scope/address space of a process
- No. of threads = No. of processors ??
- Fork-Join Model

```
main()
{
  // serial region
  fork();
  // parallel region
  join();
  // serial region
}
```

Overview

- Open Multi Processing
- Portable (Windows, Linux), shared-memory threading API for C/C++/Fortran (around 60 methods)
- Combines serial and parallel code in single source file
- Current specification: OpenMP 4.0
  - Compiler directives (Native to C/C++/Fortran)
  - Runtime library routines (e.g. increase/decrease threads as required)
  - Environment variables

Compiler Directives

```
#pragma omp construct clauses
```

- **#pragma omp** Sentinel
- **construct** The construct, or directive Name
- **clauses** e.g., shared, schedule, etc.

Overview (cont.)

Runtime Routines

- 61 routines
- Perform tasks such as Control number of threads, Query thread information, lock management, wall clock time monitoring, etc.
- Example:

```
omp_set_num_threads(8);
```

Environment Variables

- Alternative to runtime routines

```
export OMP_NUM_THREADS=8 && source /etc/profile
```

Posix Threads **pthreads**

- Before there was OpenMP, common approach to support parallel programming was(is) **pthreads**
- **P**ortable **O**perating **S**ystem **I**nterface for UNI**X**
- Originally for UNIX and Linux, but meant for all operating systems that are POSIX standard compliant (Windows did not fall down this way)

```
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}

int main() {
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0;t < NUM_THREADS;t++) {
      printf("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);
   }
   pthread_exit(NULL);
}
```

Posix Threads **pthreads** (cont.)

- Compiled as

  `gcc filename.c -lpthread`

- **Joining a Thread**: Making one thread wait for another (e.g., calling thread waiting for called thread)

```
void *foo() {
   printf("Hello Thread\n");
}

int main() {
   pthread_t tid;
   pthread_create(&tid, NULL, foo, NULL);
   pthread_join(tid, NULL);
   printf("Hello Process\n");
   exit(0);
}
```

- Many programmers find posix to be hard, cumbersome
  - Function pointers
  - Crypted functions calls such as:

    `pthread_create()`, `pthread_exit()`, `pthread_join()`

- Low chances that a compiler may optimize automatically for the above code
- Code is dependent on Posix compatible platforms (operating systems) only.
- Not designed for data-parallelism (scientific computing)

The Code (No. of Threads)

- Master thread spawns a team of threads as needed. (all thread creation work done transparently, developer saved from details)

```
#include <omp.h>

int main(int args, char *argv) {
  #pragma omp parallel num_threads(4)
  {
    tid = omp_get_thread_num();
    printf("Hello from %d\n", tid);
  }
}
```

- Compilation (Intel)

```
icc file.c -openmp
```

- Compilation (GNU)

```
gcc file.c -fopenmp
```

- Common Runtime Routines:

```
omp_set_num_threads(int);
/* Who am I */
omp_get_thread_num();
omp_get_max_threads();
/* Are we in a parallel region?*/
omp_in_parallel();
/* How many processing cores*/
omp_get_num_procs();
/* Lock a thread */
omp_set_lock();
```

- Shared vs Private (What will happen below?)

```
int tid;
#pragma omp parallel private(tid)
{
   tid = omp_get_thread_num()
}
#pragma omp parallel shared(tid)
{
   tid = omp_get_thread_num()
}
```

The Code (Conditional Parallelism)

```c
#include <omp.h>
#include <stdio.h>

int main(int args, char *argv)
{
  int tid, doIt = 0;

  #pragma omp parallel if(doIt == 1) private(tid)
  { /* Parallel Region Construct Starts here */
    tid = omp_get_thread_num();
    printf("Hello from %d\n", tid);

    if (tid == 0)
    {
      printf("Hello %d threads\n", omp_get_num_threads());
    }
  } /* Parallel Region Construct Ends here */
}
```

The Code (Reduction)

```c
#include <omp.h>
#include <stdio.h>

int main(int args, char *argv)
{
 int tid, N = 1000, sum = 0;
 int array[N];

 for (i = 0; i < N; i++) {
  array[i] = i+1;
 }

 #pragma omp parallel num_threads(N) private(tid) reduction(+:sum)
 {
  tid = omp_get_thread_num();
  sum += array[tid];
 }

 printf("Sum: %d\n", sum);
}
```

Profiling

Profiling

An analysis that may measure usage of instructions in terms of their frequency or duration of execution. The goal of profiling is to aid program **optimization**.

Clocks (Wall, User, System)

```
time ./a.out
# output of my program, if any.
real    0m0.003s # Wall clock time
user    0m0.001s # Userspace (accumulated processors)
sys     0m0.001s # Kernelspace
```

General Tips

- Run code for sufficiently long period of times
- Allow relaxation period in code for better approximation

Profiling (cont.)

Event based Methods for Profiling

- HW-based performance counters: Special purpose registers that store counts of HW-related activities (or execution of special instructions)
- Operating system hooks (e.g., function/system call interception)
- Userspace libraries (e.g., sys/times.h)

```
struct timespec start, finish; /* Defined in time.h */
/* struct timespec contains tv_sec, and tv_nsec */
/* struct timeval  contains tv_sec, and tv_usec */
clock_gettime(CLOCK_MONOTONIC, &start); /* vs CLOCK_REALTIME (NTP dependent) */
  /* Code Here */
clock_gettime(CLOCK_MONOTONIC, &finish);
double elapsed = (finish.tv_sec - start.tv_sec);
    elapsed += (finish.tv_nsec - start.tv_nsec) / 1e9;
```

Profiling (cont.)

Statistical Tools for Profiling (only percentages, no absolute time

- System-Level: (profiling through kernel using OProfile)
- User-Level: (Using tools like **gprof**, kcachegrind, valgrind, google performance tools)

Flat Profile of GPROF

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
39.47	12.01	12.01	1	12.01	21.38	func1
30.79	21.38	9.37	1	9.37	9.37	func2
30.79	30.75	9.37	1	9.37	9.37	new_func1
0.13	30.79	0.04				main

Profiling (cont.)

Call Graph of GPROF

```
index % time    self  children    called     name
[1]    100.0    0.04   30.75                 main [1]
               12.01    9.37       1/1           func1 [2]
                9.37    0.00       1/1           func2 [3]
-----------------------------------------------
               12.01    9.37       1/1           main [1]
[2]     69.4   12.01    9.37       1         func1 [2]
                9.37    0.00       1/1           new_func1 [4]
-----------------------------------------------
                9.37    0.00       1/1           main [1]
[3]     30.4    9.37    0.00       1         func2 [3]
-----------------------------------------------
                9.37    0.00       1/1           func1 [2]
[4]     30.4    9.37    0.00       1         new_func1 [4]
```

Profiling (cont.)

```
#include <omp.h>

double start_time = omp_get_wtime();
#pragma omp parallel [...]
   // code
double time = omp_get_wtime() - start_time;
```

Work Sharing

- **Work Sharing**: General term describing distribution of work across threads
- Can be performed using three constructs:
  - **for** construct (for data parallelism)
  - **sections** construct (for task parallelism)
  - **tasks** construct (for irregular problems, e.g. unbounded loops, recursive codes)

Data Parallelism

- Assuming that there is data independence across loop iterations, try:

```
#pragma omp parallel [clauses]
{
   #pragma omp for [for clauses]
   for (loop control) {
      // statements
   }
}
```

- OpenMP (or it's compilers) cannot (always) automatically identify data dependencies
- Threads "share" the iterations of the for loop
- Equivalent Code:

```
#pragma omp parallel for [for clauses]
for (loop control) {
   // statements
}
```

Data Parallelism (cont.)

```
#pragma omp parallel num_threads(2) private(tid)
{
   tid = omp_get_thread_num();
   #pragma omp for
   for (i = 0; i < 20; i++) {
      printf("%3d by %3d\n", i, tid);
   }
}
```

Data Parallelism (cont.)

Schedule Clauses (How loop iterations are mapped to threads)

Static Scheduling

- Low-overhead
- Load imbalance
- threads assigned "chunks" of iterations

Dynamic Scheduling

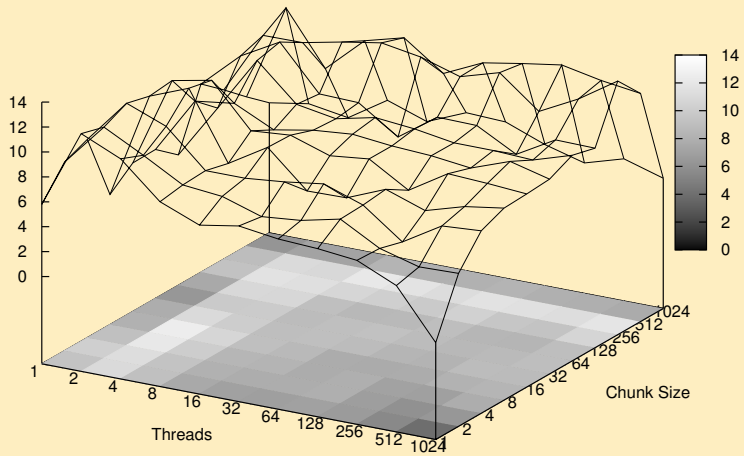
- High-overhead
- Reduces load imbalance
- Threads grab "chunks" of iterations

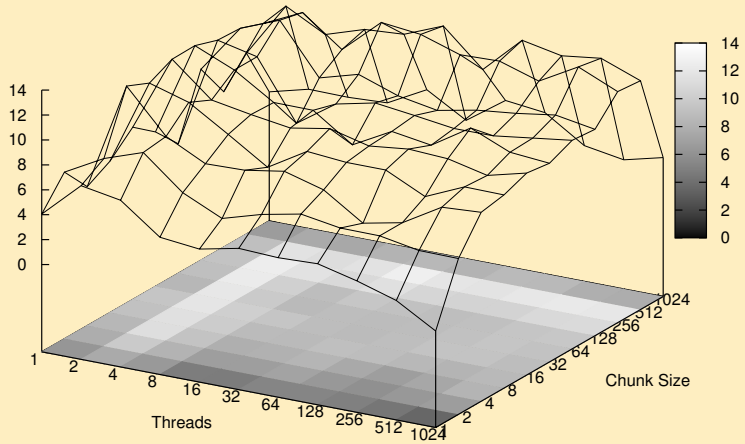
- `#pragma omp for schedule(static,chunksize)`
- `#pragma omp for schedule(dynamic,chunksize)`

Data Parallelism (cont.)

Effect of Chunk Size and Thread Quantities

- Computed 1024×1024 matrix multiplication using static and dynamic scheduling
- (Left) Static Scheduling (Right) Dynamic Scheduling





Data Parallelism (cont.)

Is Schedule Class Really Necessary?

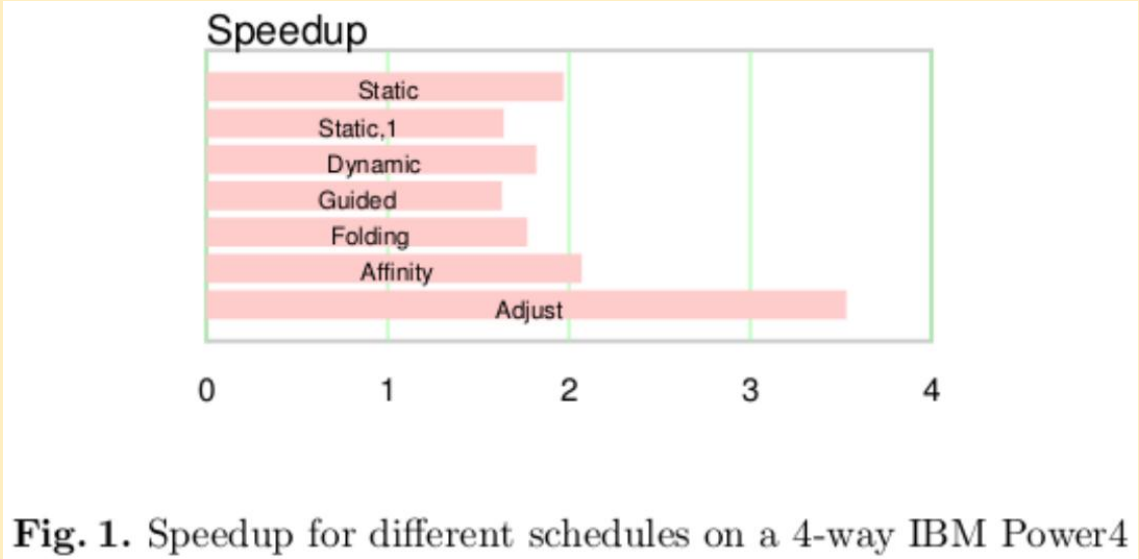


Fig. 1. Speedup for different schedules on a 4-way IBM Power4

- Ayguade et. al., Is the schedule clause really necessary in OpenMP?, Technical Report, Springer Verlag, 2003

Task Parallelism

- Considering following scenario:

```
p = pcibus();
n = networkCard(p);
w = wifiCard(p);
s = ssh(n,w);
h = http(n,w);
f = ftp(n,w);
```

- n, w can be executed in parallel
- s, h, and f can be executed in parallel
- Task Parallelism using **sections** in OpenMP:

```
#pragma omp parallel [clauses]
{
   #pragma omp sections
   {
      #pragma omp section
      // Code of first task

      #pragma omp section
      // Code of second task
   }
}
```

Task Parallelism (cont.)

- **Sections** must be inside a parallel region. **Sections** itself provides enclosure for an individual **omp section**

```
p = pcibus();
#pragma omp parallel sections num_threads(2)
{
   #pragma omp section
   n = networkCard(p);
   #pragma omp section
   w = wifiCard(p);
}
#pragma omp parallel sections num_threads(3)
{
   #pragma omp section
   s = ssh(n,w);
   #pragma omp section
   h = http(n,w);
   #pragma omp section
   f = ftp(n,w);
}
```

- If no. of threads is $<$ no. of tasks, threads first attempt the beginning tasks before jumping to next ones.
- If no. of threads is $>$ no. of tasks, each task is performed by one thread, while the remaining remain idle

OpenMP Tasks for Task Parallelism

- OpenMP Tasks suitable for irregular problems (problems without loops, unbounded loops, recursive algorithms, etc.)
- Example: Summation across loop:

```
int summation(int *array, int N)
{
   int i, sum = 0;
   for (i = 0; i < N; i++)
      sum += array[i];
   return sum;
}

int main()
{
   int N = 1000;
   int array[N];

   for (i = 0; i < N; i++)
      array[i] = i+1;

   int sum = summation(array, N);

   printf("Sum: %d\n", sum);
}
```

- Example: Summation without loop (recursive):

OpenMP Tasks for Task Parallelism (cont.)

```
int summation(int *array, int N)
{
   if (N == 0)       return 0;
   else if (N == 1)   return *array;

   int half = N / 2;
   return summation(array, half) + summation(array+half, N - half);
}
```

- Replacing recursive call with OpenMP Tasks:

```
#pragma omp parallel
#pragma omp single
{
   #pragma omp task shared(x)
   x = summation(array, half);
   #pragma omp task shared(y)
   y = summation(array+half, N - half);

   #pragma omp taskwait

   x += y;
}
return x;
```

- Tryout: Fibonacci Series $f(1) = 1, f(2) = 1, f(n) = f(n-1) + f(n-2)$
- Tryout: Traversing Linked List (unbounded loop)

OpenMP Tasks for Task Parallelism (cont.)

```
struct LL {
   int x;
   struct LL *next;
};

int main()
{  int i, random = 10;    // rand() % 10;

   struct LL *head, *tail, *current;
   head = malloc(sizeof(struct LL));
   head->x = 0;
   head->next = NULL;

   tail = head;
   for (i = 1; i < random; i++) {            // populating the linked list
      current     = malloc(sizeof(struct LL)); // allocate space for new node
      current->x    = i;                   // set its value to i
      current->next = NULL;                  // next is obviously null
      tail->next    = current;               // set tail of last node
      tail        = current;               // set new tail
   }
   current = head;
   while(current) {                        // this is the actual work
      printf("%d\n", current->x);             // task which we modify
      current = current->next;               // on the next slide
   }
}
```

So modifying the work into tasks from previous slide, we have:

OpenMP Tasks for Task Parallelism (cont.)

```
#pragma omp parallel
#pragma omp single
{
   current = head;
   printf("single is: %d\n", omp_get_thread_num());
   while(current) {
      #pragma omp task
      printf("%d by %d\n", current->x, omp_get_thread_num());
      current = current->next;
   }
}
```

- **Properties of Tasks**
  - Tasks are independent units of work (but if dependencies exist, then their execution may be deferred)
  - One thread assigned to perform work of a given task
  - Tasks can be nested (e.g. task within task)
  - All threads cooperate in execution of each other.
  - Tasks may be deferred, or executed immediately (decided by runtime automatically)
  - Task threads can suspend the execution of a task and start/resume another
- Why is **single** construct being used?
  - One thread enters single construct and creates all tasks (you don't want all threads to enter construct and create duplicate tasks)

OpenMP Tasks for Task Parallelism (cont.)

- This single thread will create a pipeline (task queue) on which other tasks will be placed.
- By default, all threads will start execution of tasks **only when** single construct (thread) finishes its job. This can be overridden by passing **nowait** clause to the single construct.

- How are both segments and tasks executed?
  - All **segment** based threads will be "alive" together. (In other words, all threads will implicitly wait at a termination barrier). (Or, in other words, no thread will exit unless the other ones are ready to exit.)
  - In tasks, many possibilities can exist. **Default is:** Worker threads will wait for thread running single construct to create tasks. As soon as the tasks are created, they, including the thread running single construct may pick any task and execute it.

Synchronization

- **Synchronization**: A technique that (a) imposes order of execution, and/or (b) provides protected access to shared data
  - **High Level Synchronization**: barrier, taskwait, critical, atomic
  - **Low Level Synchronization**: locks (simple, nested)

Synchronization (cont.)

High Level Synchronization

- **Barriers** (All work being performed by any thread is guaranteed to be completed at barrier exit). Usage:

  ```
  #pragma omp barrier
  ```

- For OpenMP Tasks, the barrier is the **taskwait** clause. Usage:

  ```
  #pragma omp taskwait
  ```

- **Critical Section** provides for mutual exclusion: only one thread can enter critical section at a time

  ```
  int myValue = 0;
  #pragma omp parallel num_threads(500) shared(myValue)
  {
     #pragma omp critical
     myValue++;
  }
  printf("Value: %d\n", myValue);
  ```

- **atomic** also provides mutual exclusion by making sure read/write to memory location is protected and singular. (Different from critical clause, which can be any sequence/type of code)

  ```
  #pragma omp atomic
  ```

Synchronization (cont.)

Low Level Synchronization

- OpenMP **Simple Locks**: A lock is available if it is **unset**

```
omp_lock_t myLock;          // the lock object
omp_init_lock(&myLock);     // initialize to unset
#pragma omp parallel
{
   omp_set_lock(&myLock);   // set lock (wait if already locked)
   // do some work
   omp_unset_lock(&myLock); // unset lock
}
omp_destroy_lock(&myLock);  // destroy/deallocate lock
```

  - Can also test if a lock is set or not (without blocking) by calling:

```
omp_test_lock(&myLock);  // returns true if set, false otherwise
```

- OpenMP **Nested Locks**:
  - A set is an increment, an unset is a decrement.
  - Can be locked a number of times. Doesn't unlock until you have unset it as many times it was locked.

```
omp_nest_lock_t   myLock;
omp_init_nest_lock(&myLock);
omp_set_nest_lock(&myLock);
omp_unset_nest_lock(&myLock);
omp_destroy_nest_lock(&myLock);
```