Introduction

us-gaap:RealizedGainLossOnInvestments numericContext="usd_flow_2002"> 2007146000</us-gaap:RealizedGainLossOnInvestments> <us-gaap:RealizedGainLossOnInvestments numericContext="usd_flow_2002
us-gaap:RealizedGainLossOnSalesOfInvestmentsInAffiliatedIssuers numericContext="usd_flow_2002">637000</us-gaap:RealizedGainLossOnSalesOfInvestmentsInAffiliatedIssuers> <us-gaap:RealizedGainLo
us-gaap:RealizedGainLossOnForeignCurrencyTransaction numericContext="usd_flow_2002"> 434000</us-gaap:RealizedGainLossOnForeignCurrencyTransaction> <us-gaap:RealizedGainLossOnForeignCurrency
us-gaap:RealizedGainLossOnFutureContracts numericContext="usd_flow_2002"> 34969000</us-gaap:RealizedGainLossOnFutureContracts> <us-gaap:RealizedGainLossOnFutureContracts numericContext="usd_
us-gaap:NetRealizedGainLoss numericContext="usd_flow_2002"> 2042548000</us-gaap:NetRealizedGainLoss> <us-gaap:NetRealizedGainLoss numericContext="usd_flow_2002"> 2042548000</us-gaap:NetReali

Overview

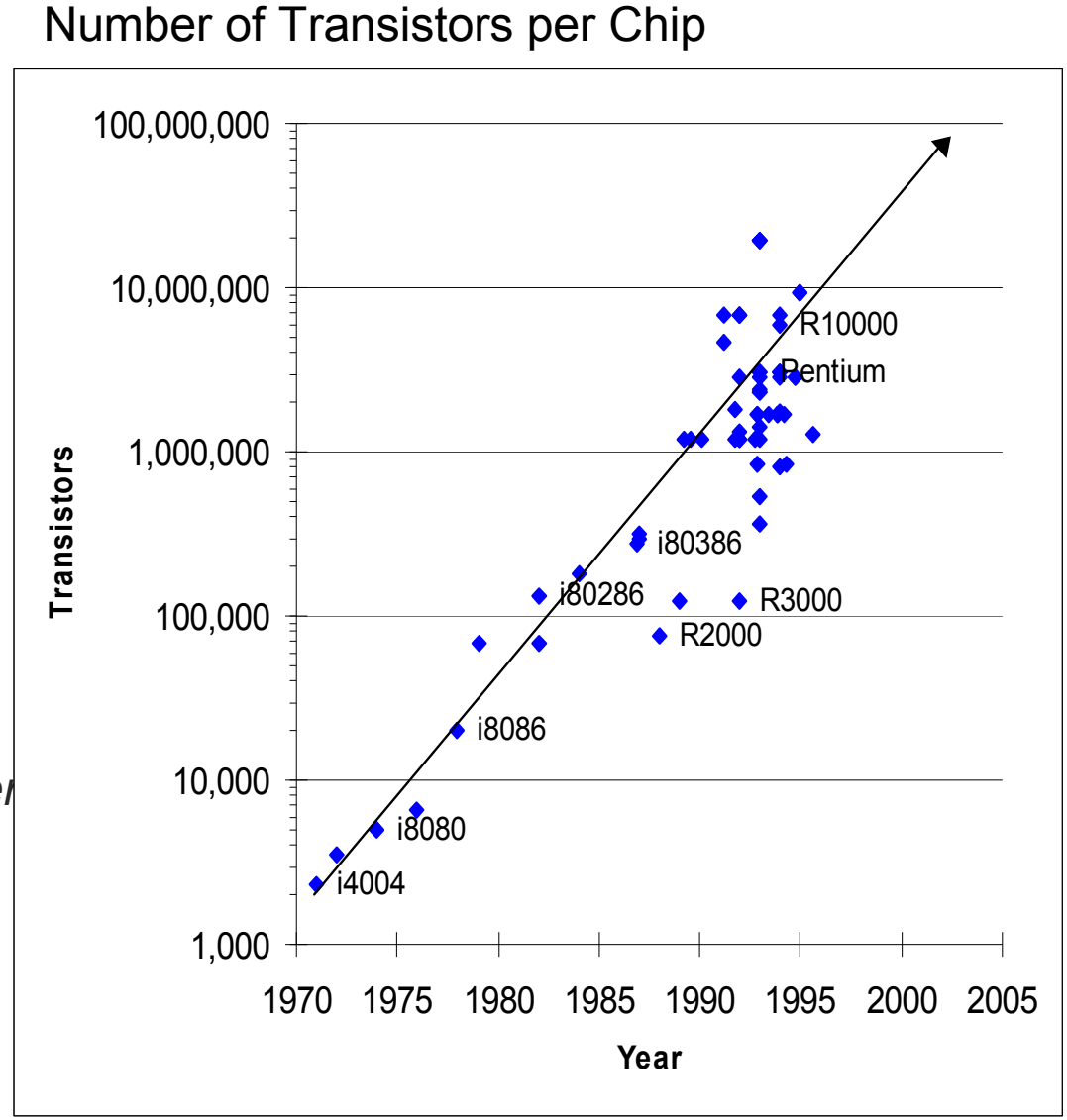
- Parallelism
- Parallel Architectures
- Parallel Programming
  - *General Purpose Mechanism*
  - *Why Java*
- Example: Parallel Sum

Why Parallel

- Some tasks are parallel by nature
  - *Data mining, Search, Medical imaging, Pharmaceutical design, Engineering Design, …*
  - *Some of largest applications of parallel computing: 3D Animation, Financial Applications*
- Parallel can be faster
  - *Two is always faster than one (really?)*
- Can solve larger problems given same amount of time

Why Parallel: Moore's Law

» Moore's law
- *Performance doubles every 18 months*
- *Can't deliver via higher clock frequency*
  - *Clock frequency hasn't been increased for almost three years, record holder of Pentium 4's 3.8Ghz*
  - *Limits to miniaturization, processor technology hits the wall*
  - *Economic limitations - it is increasingly expensive to make a single processor faster Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.*



Why Parallel

- Performance has to come from parallelism from now on
- With 2 cores, you can afford not to be parallel with a word processor. With 16 cores, you have to be parallel. Parallel computing is not just for HPC anymore, it's for everybody. Desktop supercomputing.

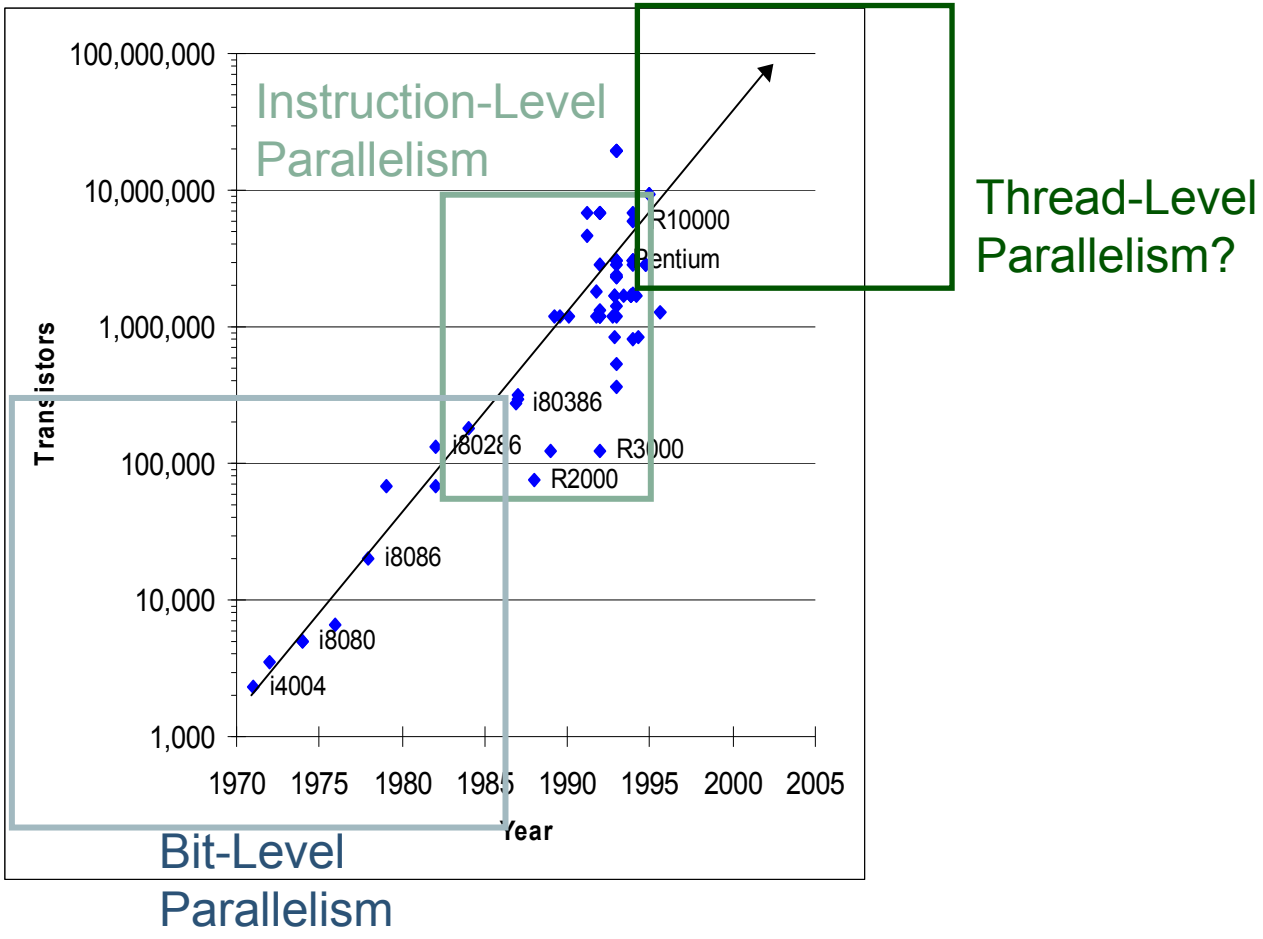
Parallelism in Computing

» Parallelism in Computer Programs

- *Different levels of parallelism*
- *Bit Level Parallelism*
- *Instruction Level Parallelism (ILP)*
- *Thread Level Parallelism, shared-memory, multi-core processors*
- *Heterogeneous Parallelism, distributed computing vs parallel computing*
- *Task Level Parallelism:*
  - *High throughput is the goal instead of high performance*
  - *Multiple tasks with multiple cores vs single task with multiple cores*
  - *“Can handle multiple requests to a server” as opposed to “Sort an array with a parallel machine”.*
  - *Subtle difference between a server and a parallel computer. Some multi-core processors are more optimized for server instead of executing a parallelized task*

Number of Transistors per Chip

- Parallelism in Computer Programs



Instruction Level Parallelism (ILP)

- Example:
  - `(a+b)*(c+d)`, *suppose* `a,b,c` *and* `d` *are independent of each other. So two subexpressions* `(a+b)` *and* `(c+d)` *can be computed in parallel*
- Modern processors have multiple execution units
  - *Single core Pentiums have 2 integer ALUs and 1 FPU*
- Compilers are smart enough to discover ILP
  - *Out-of-order executing*
  - *Branching prediction*
- Programmers do not have to capture ILP conceptually with programming languages

Hidden Parallelism

- Traditionally, programmers depend on advances in silicon and compilers to improve performance
- Programmers no need to concern with hardware performance and changed their techniques and methodologies little over the years even though the underlying processors become more and more parallel
  - *Pipeline, multiple execution units, instruction level parallelism*
  - *Compilers are doing reasonably good job*
- Algorithm improvements mostly focused on serial/sequential algorithms
  - *Some theoretical results in parallel algorithms when there's no popular hardware platform to test*

Explicit Parallelism

- With Multi-core architecture, programmers have to face the parallelism directly
- Multi-core parallelism (thread-level) parallelism can not be exploited with existing tools. Some research has been conducted. But in reality, taking advantage of parallelism is becoming programmers' job
- Programmers need to understand a new parallel programming model,
  - *a shared memory, multi-core model is the focus*
- Think in Parallel
  - *Programmers need to understand parallel algorithms on the new model*

Explicit Parallelism: Heterogeneous

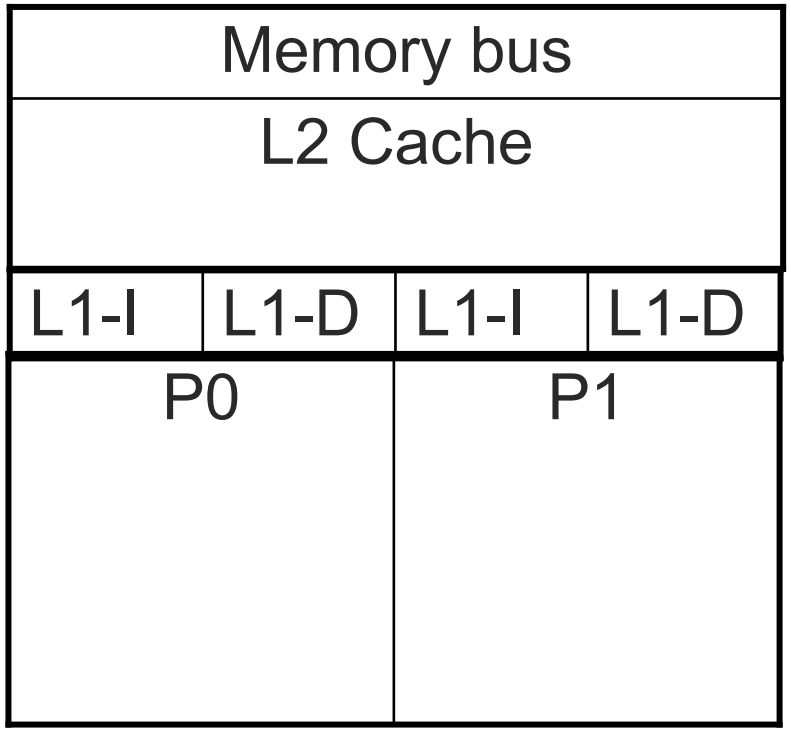
- Heterogeneous parallelism can be very complicated
  - *Cores with different architectures inside one chip*
  - *Co-processors*
    - *Special purpose processors, GPU as general purpose processors GPGPU for high performance float point computing*
  - *Clusters (heterogeneous distributed memory computing)*
  - *System-level parallelism*
    - *Not necessarily single task, multiple tasks, high performance vs high throughput computing*
- Programmers need some understanding of underlying hardware architecture

Overview

- Parallelism
- ***Parallel Architectures***
- Parallel Programming
  - *General Purpose Mechanism*
  - *Why Java*
- Example: Parallel Sum

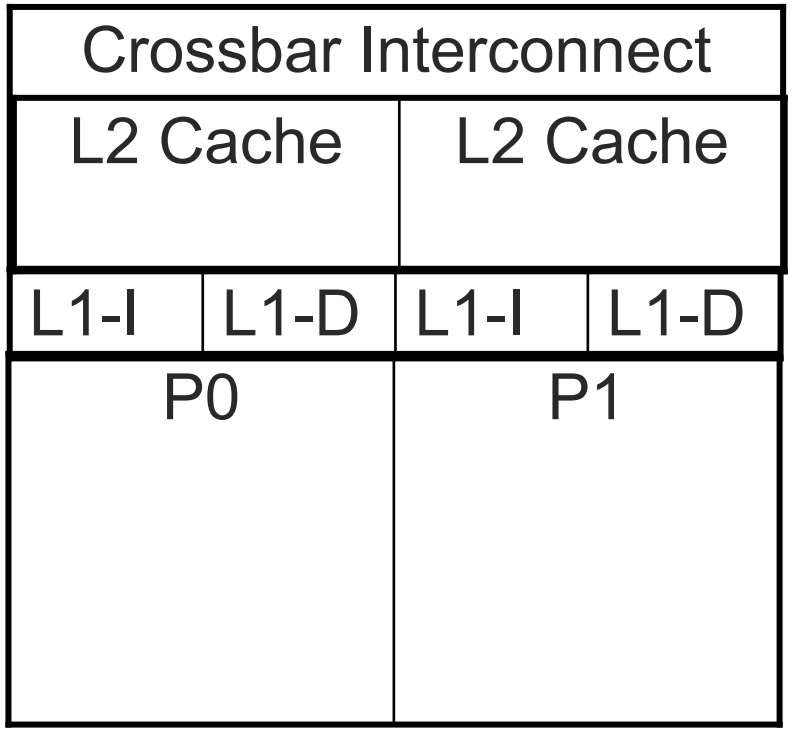
Processor 1: Intel Core Duo

» Dual-core



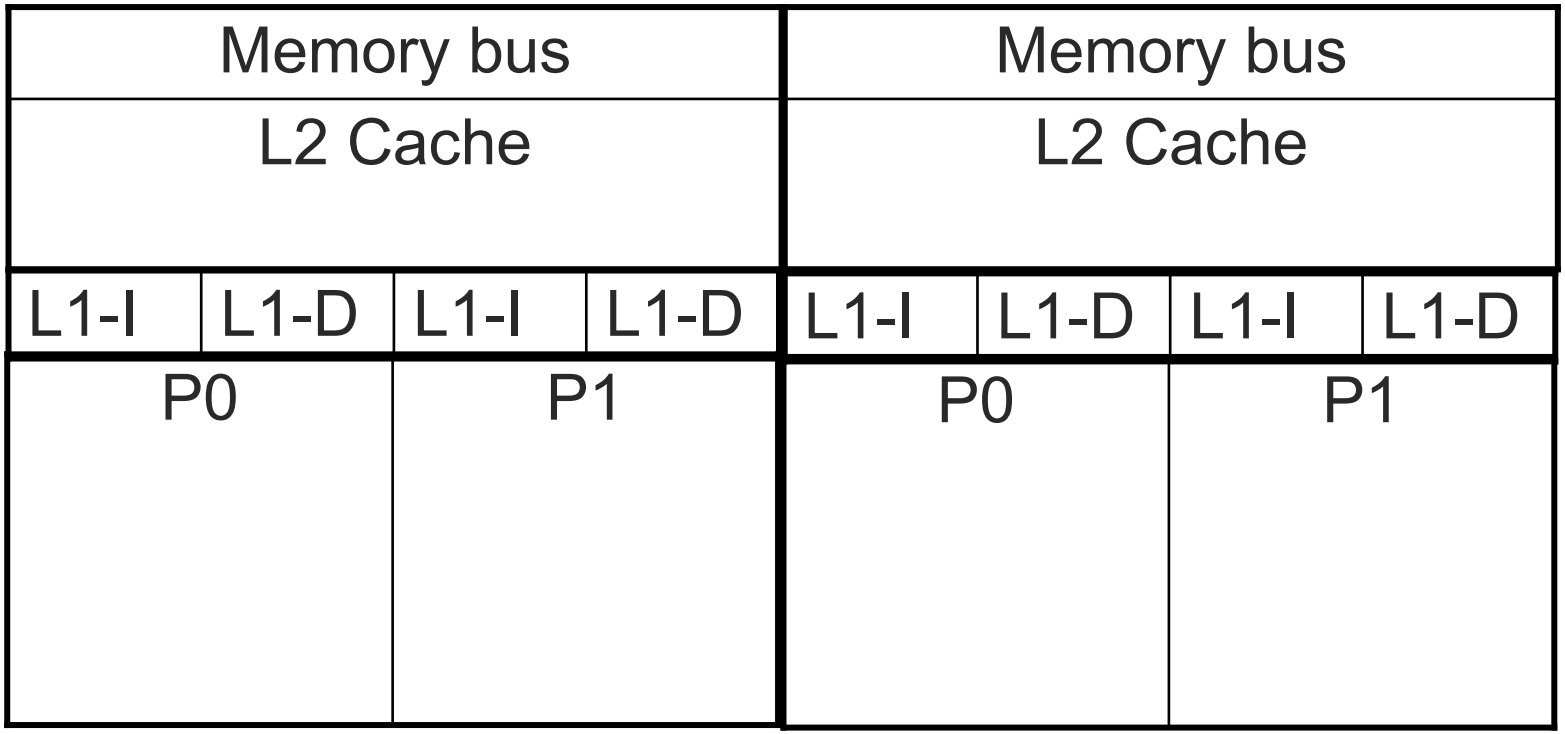
Processor 2: AMD Dual Core Opteron

» Dual-core



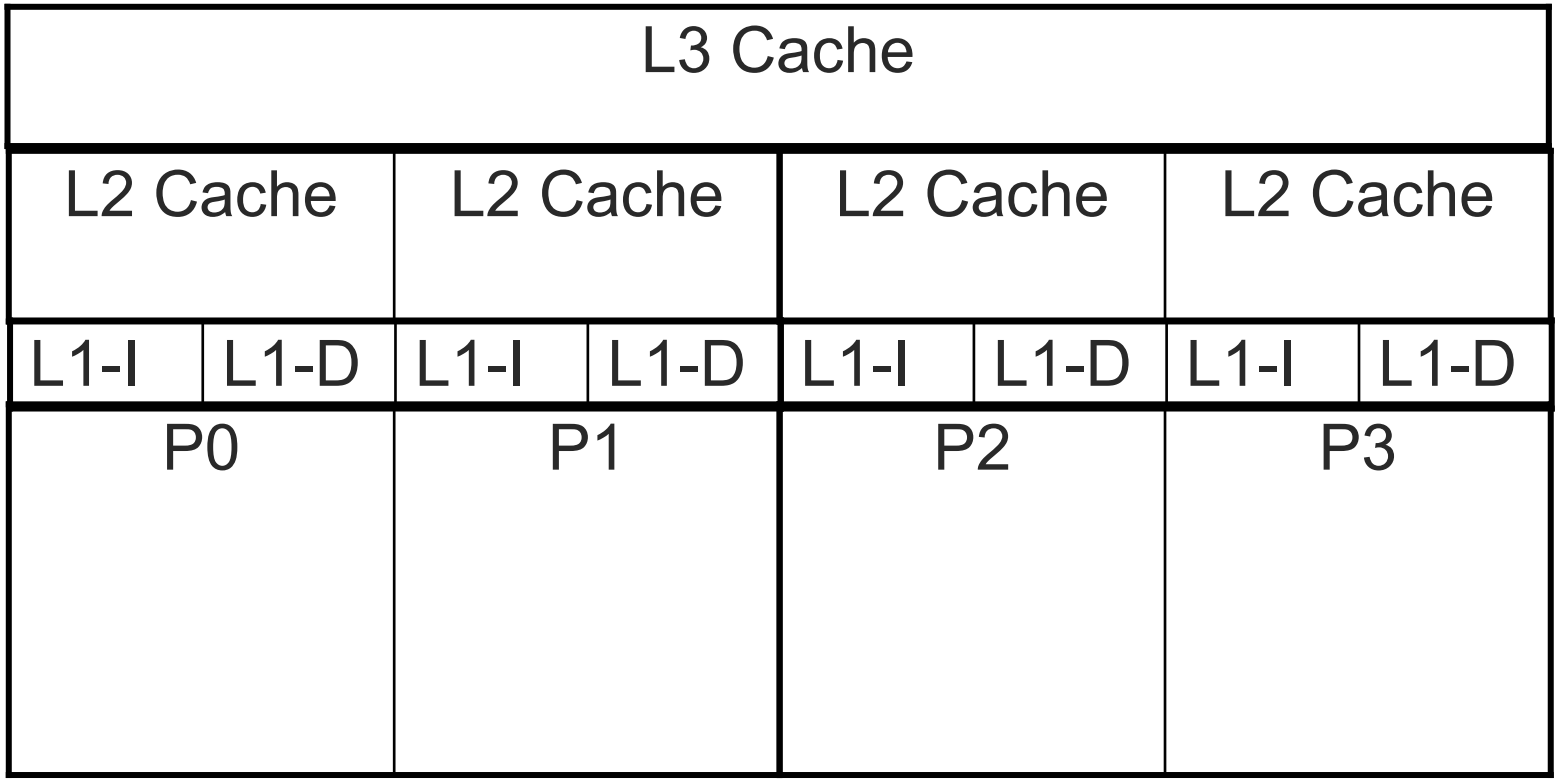
Processor 3: Intel Quad Core

» Quad-core: two dual core glued together



Processor 4: AMD Quad Core Opteron

» Phenom-II Quad-core

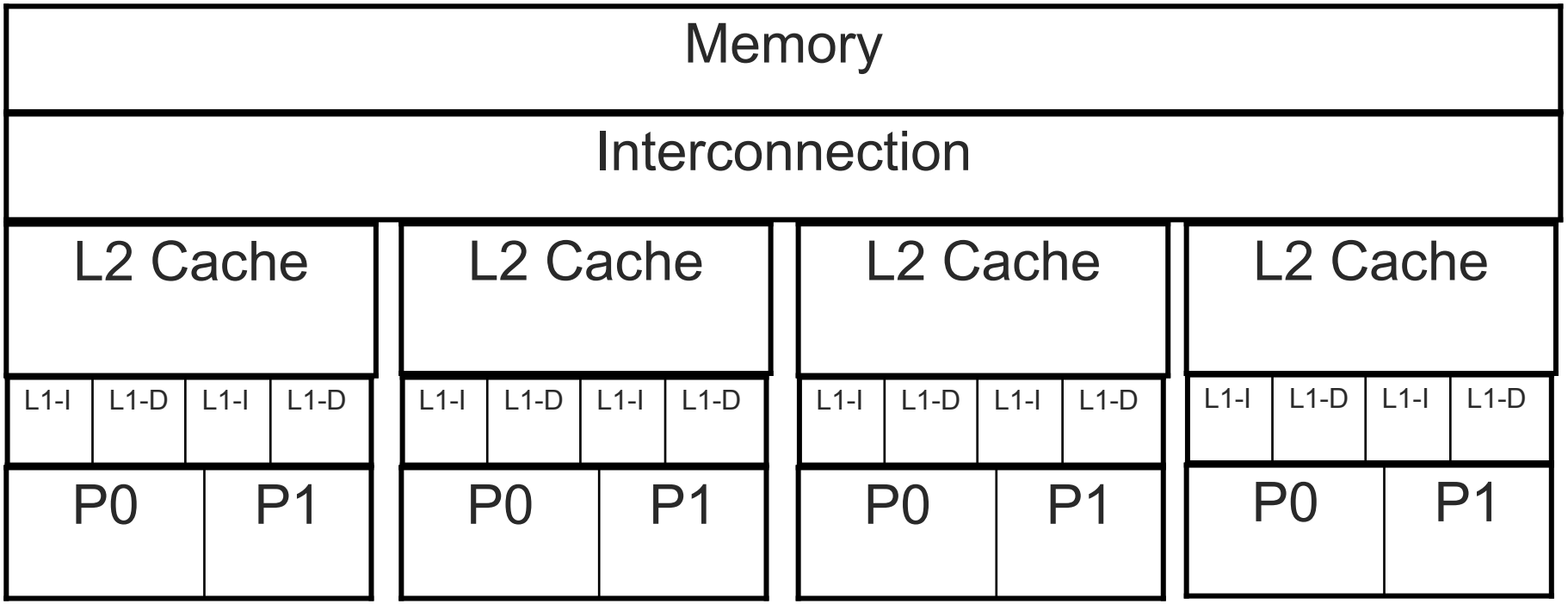


Processor 5: Intel i7

- Intel i7 has a similar architecture as AMD Quad Opteron, each processor core has its dedicated L2 and all share an L3 cache
- Intel i7 core is hyper-threading, one physical core supports two logic cores simultaneously
- Intel i7 QPI (Intel's new interconnection) has higher bandwidth
- Intel i7 has better performance than AMD Quad Opteron
- AMD's new 6-core Opteron (Istanbul) has similar architecture as Quad-core (Shanghai) but with 2 more cores

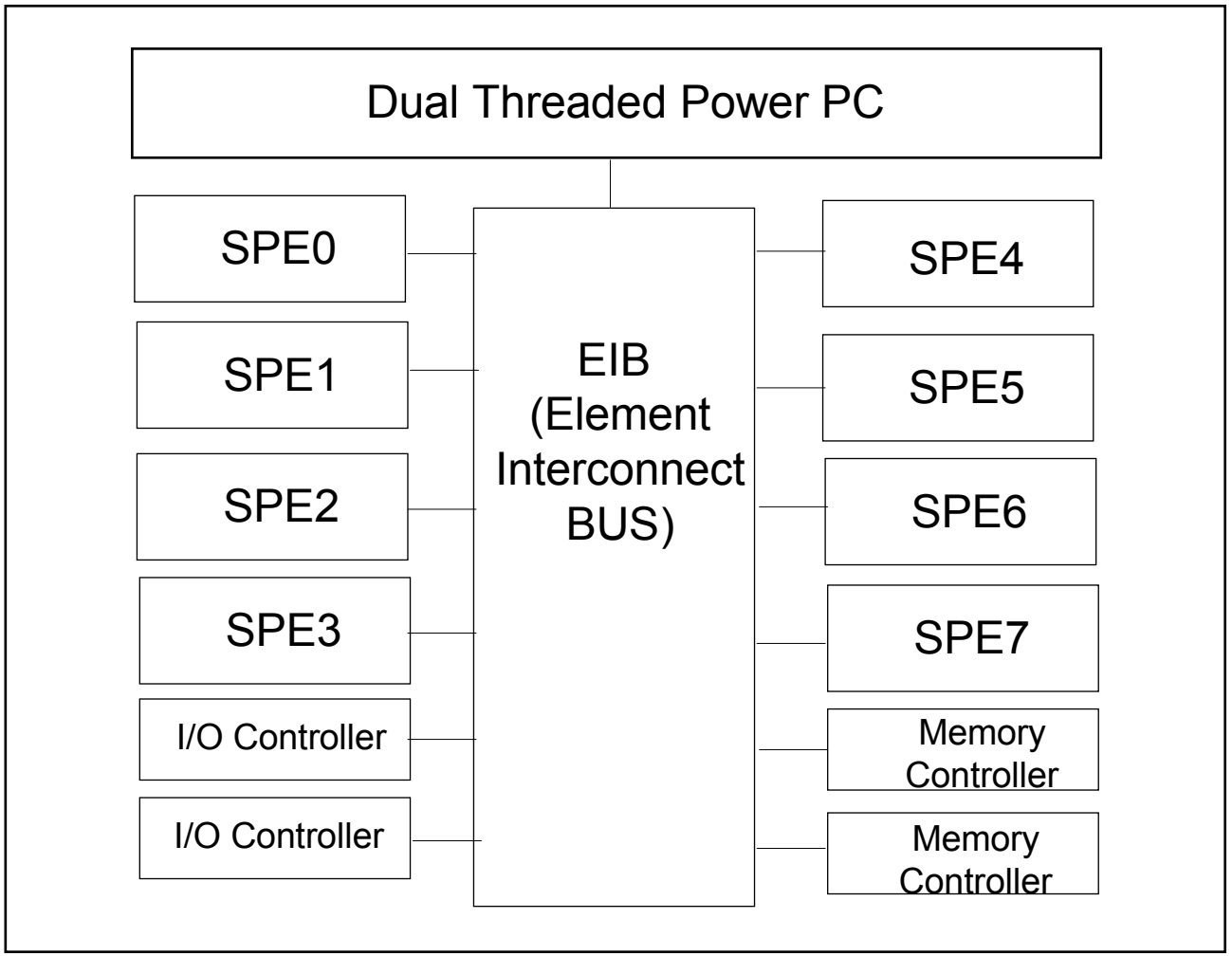
Processor 6: Larger SMP (Scale Symmetric Multiprocessors)

- Multiprocessor: more than one processor boundary
- The interconnection could be a simple memory bus (not scalable) or crossbar (such as Sun Fire E25K scale to 72 processors)



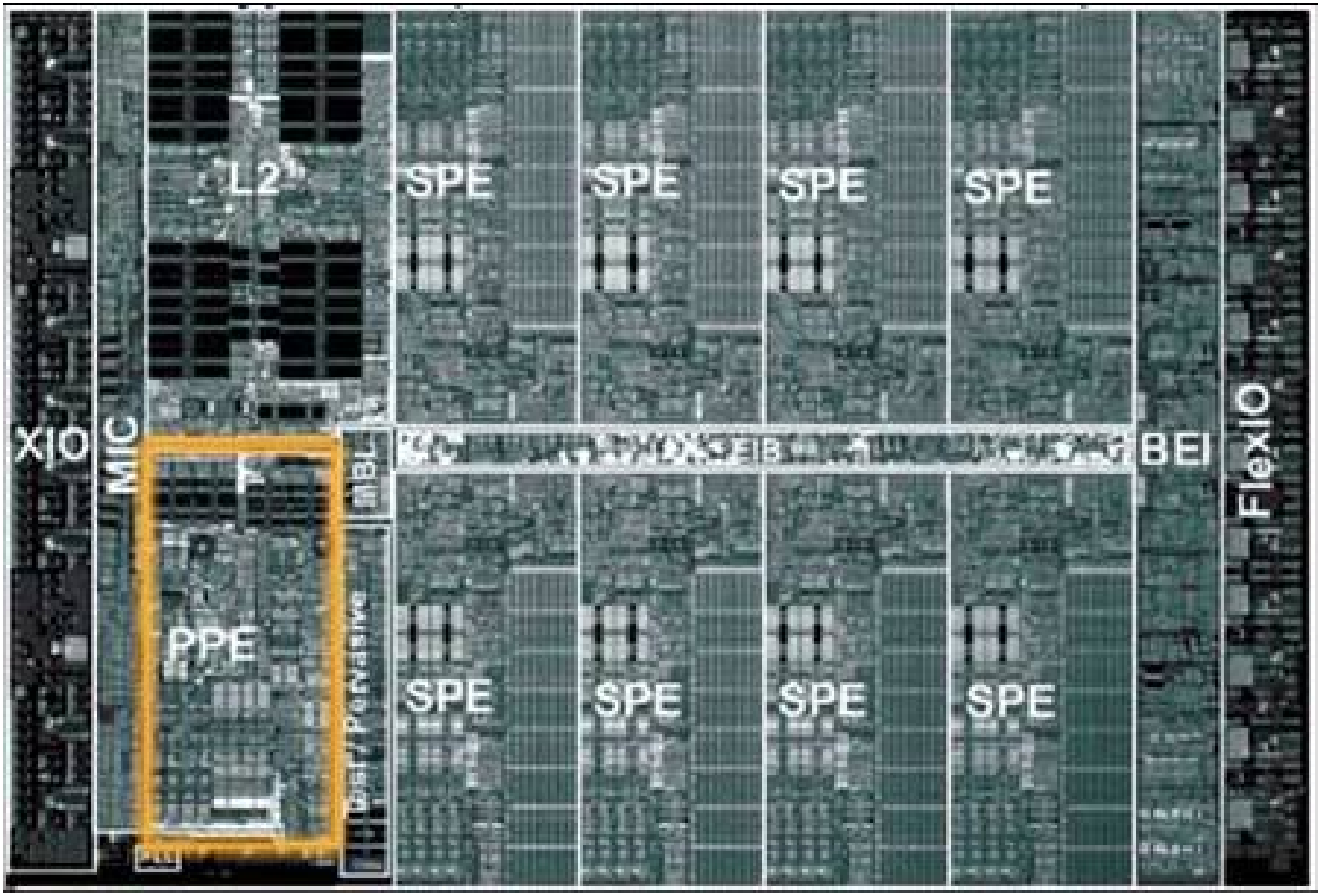
Processor 7: Cell

» Heterogeneous parallelism



Processor 7: Cell (Continued)

» Cell Xray'ed



Processor 7: Cell (Continued)

- A Joint Development by IBM, Sony and Toshiba
- CPU for PS3 and also powers the Roadrunner, the fastest supercomputer so far
- A Hyper-threading PowerPC Core with 8 Special Purpose Processor SPE (Synergistic Processing Elements) that support 32-bit vector operations
- Each SPE has 256KB on chip RAM, a SIMD capable of 4 single precision floating-point operations per cycle
- High Memory Bandwidth (2*12.8GB/s)
- Capacity of EIB 204.8GB/s
- Very hard to program
  - *On chip RAM sync with main RAM, timing the data movement, data partitioning*
  - *PowerPC core is of different programming model with SPEs*

Hyper Threading

- Also known as “SMT (Simultaneous Multi-Threading). Large scale use started at Intel Norwood (Pentium 4) architecture. Now even Intel Atom (netbook CPU) processor supports hyper-threading
- One physical core supports multiple logical cores
  - *Making one physical core with multiple execution units to appear as two or more logical cores at application level*
  - *Intel i7 each core has 6 execution units capable of executing 3 memory operations and 3 calculations in the same time*
- SMT or Hyper-Threading try to use as much as possible of all executing units
  - *looks for instruction parallelism in two threads instead of just one, with the goal of leaving as few units unused as possible. Extremely effective when the two threads are executing tasks that are highly separate. The performance gain can be as high as 40%+.*
  - *On the other hand, two threads involving intensive calculation will only increase the pressure on the same calculating units, putting them in competition with each other for access to the cache. Turning off hyper-threading will gain performance slightly*

Hyper Threading (continued)

» Hyper-threading performance gain on Intel Atom
  - *Run a parallelized super-pi program with OpenMP, gained 47% performance*

Number of Threads	Seconds	Speedup
1	2.817	1
2	1.911	1.47

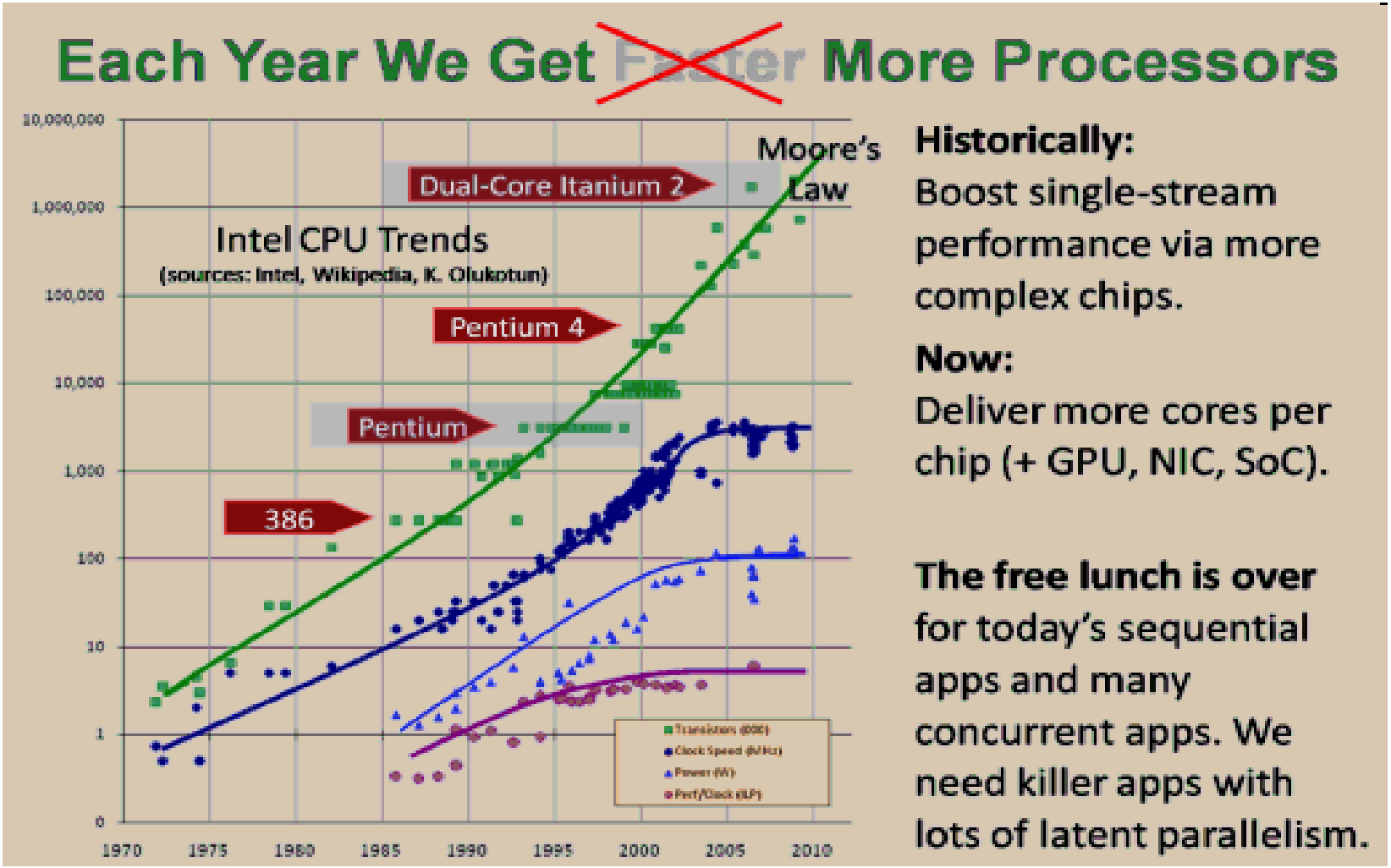
Multi-core and Many-core

- Current status (circa 2009) Muti-core: 8-core
  - *6-core processors, general purpose processor, Intel and AMD*
- Future: Many-core, 10 or more cores, 64 SMT
- Some GPUs already have more than 100 cores

	Intel Core 2 Duo	Hypothetical Larrabee
# of CPU Cores	2 out of order	10 in-order in experiment, 16-32 in production
Hyper-threading	1	4 SMT
Instructions per Issue4	4 per clock	2 per clock
VPU Lanes	4-wide SSE16	16-wide
L2 Cache	4MB	4MB
Single-Stream Throughput	4 per clock	2 per clock
Vector Throughput	8 per clock	160 per clock

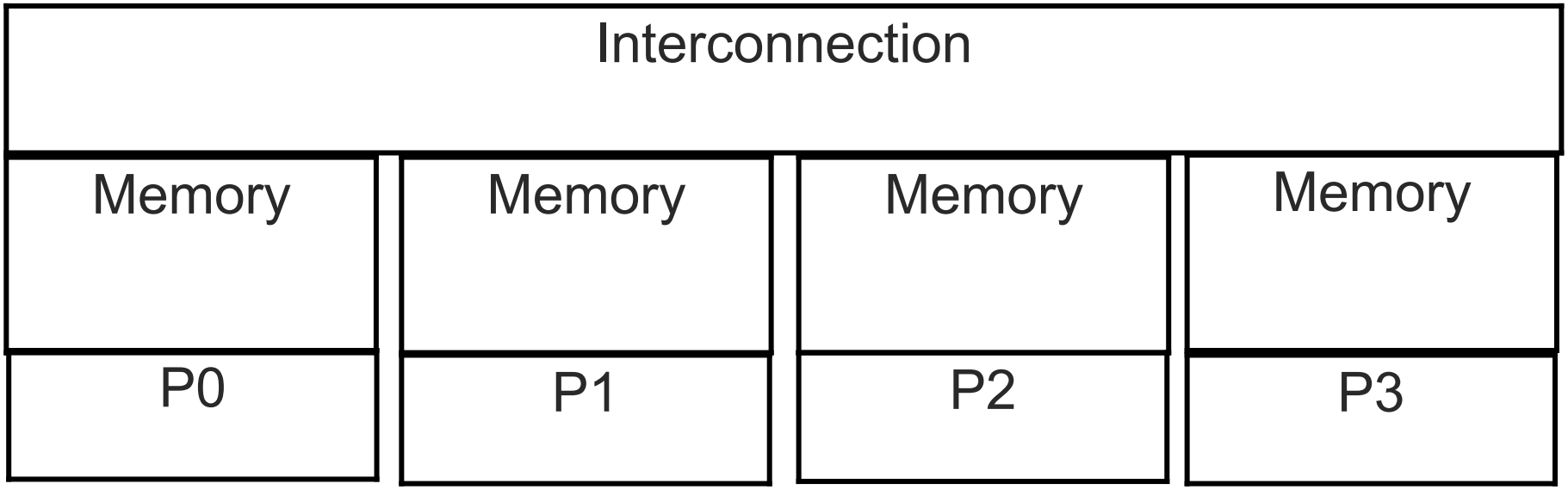
The Future is Many-core

» "more" is in and "faster" is out



Processor 8: Distributed Memory Machine

- Consists of multiple computers, each has its own memory
- The computers are connected with an interconnection
- Clusters



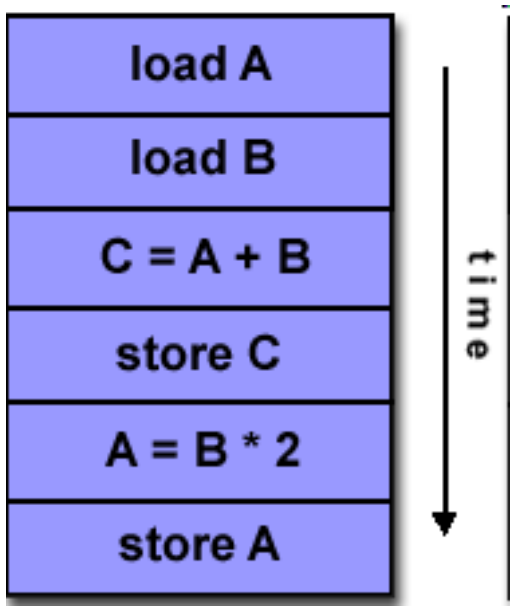
Flynn's Classical Taxonomy

- Flynn 1966, 2-dimention of streams, instruction and data
- Still valid to most extent today

SISD Single Instruction, Single Data	SIMD Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

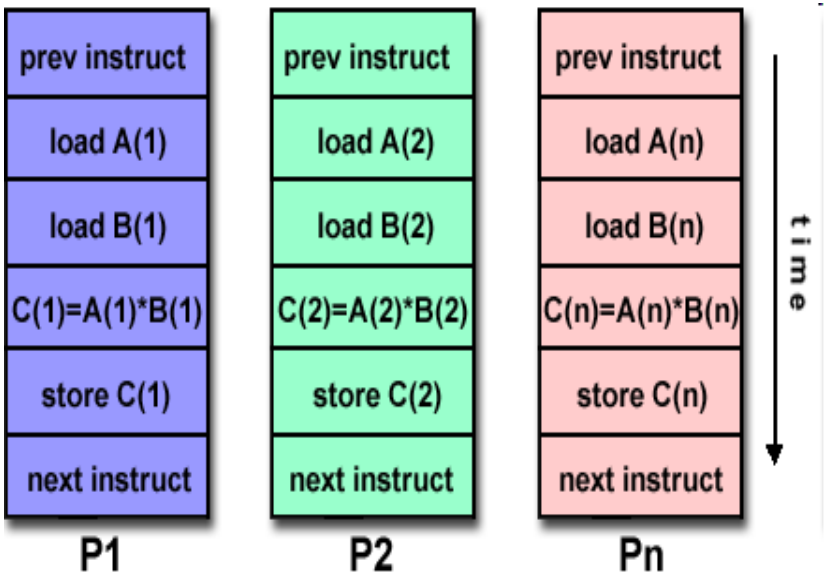
SISD

- Single Instruction, Single Data
  - *Classical von-Neumann computer, serial*
  - *One instruction stream acted on one data during one clock cycle*
  - *No parallelism*
  - *Deterministic execution*
  - *Conceptually, single-core processor*



SIMD

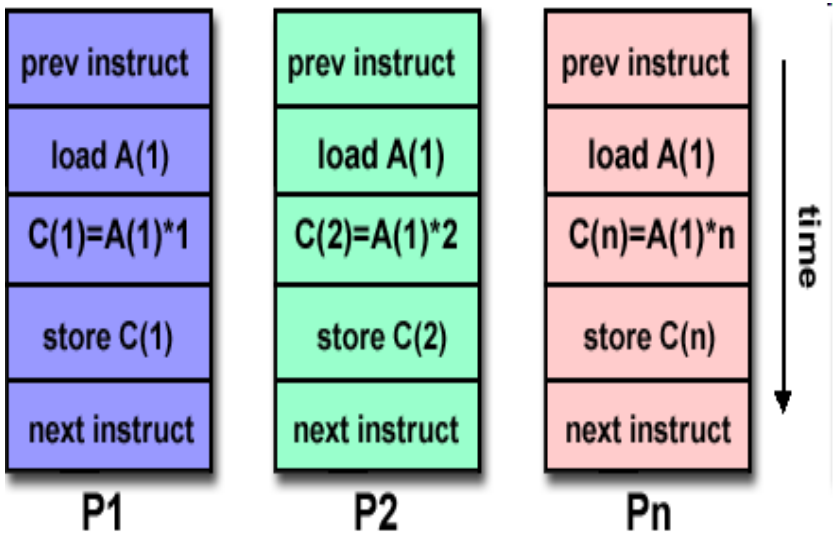
- Single Instruction, Multiple Data
  - *Single Instruction: multiple execution units, all processing units execute the same instruction at any given clock cycle*
  - *Multiple Data: Each execution unit can operate on a different data elements*
  - *Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.*
  - *Synchronous (lockstep) and deterministic execution*
  - *Two varieties: Processor Arrays and Vector Pipelines*
  - *Examples:*
    - *Processor Arrays: Connection Machine CM-2, ILLIAC IV*
    - *Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10*
    - *Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.*



MISD

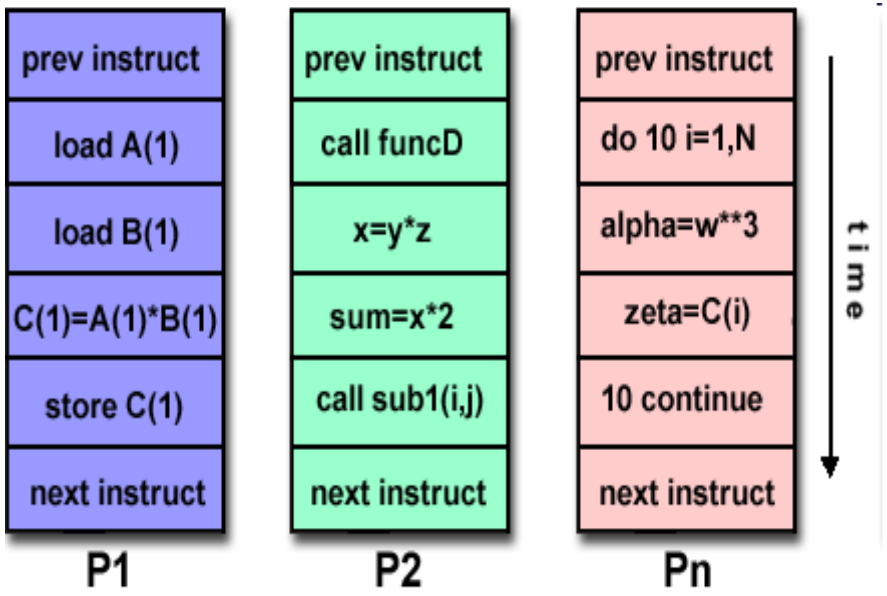
» Multiple Instruction, Single Data

- *A single data stream is fed into multiple processing units.*
- *Each processing unit operates on the data independently via independent instruction streams.*
- *Few actual examples of this class of parallel computer have ever existed.*
- *Some conceivable uses might be:*
  - *multiple cryptography algorithms attempting to crack a single coded message.*
  - *High-reliability or High-redundancy computing, as in current Linux and Windows*



MIMD

» Multiple Instruction, Multiple Data
  - *Currently, the most common type of parallel computer. Most modern computers fall into this category.*
  - *Multiple Instruction: every processor may be executing a different instruction stream*
  - *Multiple Data: every processor may be working with a different data stream*
  - *Execution can be synchronous or asynchronous, deterministic or non-deterministic*
  - *Examples:*
    - *most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.*
    - *Even some single-core processor can be classified as MIMD*
  - *Note: many MIMD architectures also include SIMD execution sub-components, e.g., Cell, Nvidia CUDA*



SPMD

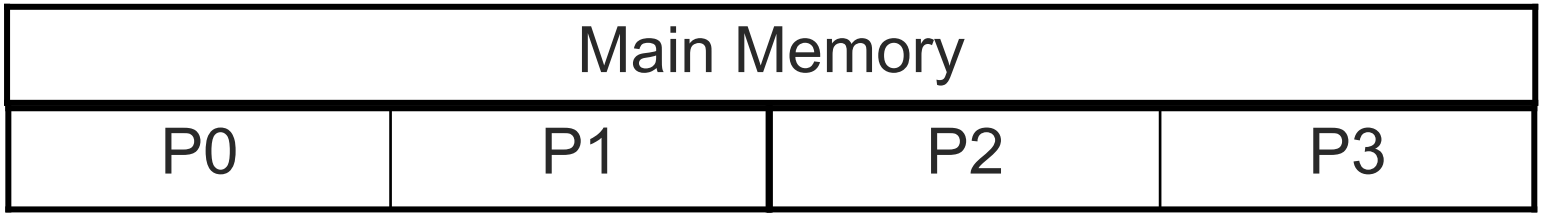
- SPMD: Single Program, Multiple Data
  - *A problem with Flynn's approach: Is today's single-core processor a SIMD or MIMD?*
  - *Conceptually, a single program from programmers' point of view*
  - *Even though multiple instructions are issued but still from one single program stream*
  - *More of software classification as SIMD, MIMD for hardware*

Overview

- Parallelism
- Parallel Architectures
- ***Parallel Programming***
  - *General Purpose Mechanism*
  - *Why Java*
- Example: Parallel Sum

Physical Model for SMP: PRAM

» PRAM: Parallel RAM



» Advantages

- *Unit cost of memory access, ignoring difference between L1,L2,L3 and main memory*

» Disadvantage

- *Unit cost of memory access,*
- *Programmers need to be cautious on memory access*
- *algorithm behavior inconsistent*

Programming Models for SMP: Threading

- Multi-threading mechanism in programming languages
  - *C POSIX Pthread, standard C threading library*
  - *Java Thread*
    - *Supported in the language level, behavior is consistent across different platforms*
    - *Thread is a class*
    - *Standard, simple, easy to understand*
    - *Three most important primitives*
      - *start(), run(): to spawn a new thread*
      - *join(): waiting the new to be finished*
      - *synchronized: allow only one thread access the segment (code or data) in the same time*

Programming Models: Shared Memory vs Message Passing

- Shared Memory
  - *Natural extension to flat main memory*
    - *PRAM*
    - *Threading, one thread corresponds to a processor, conceptually*
- Messaging Passing
  - *A more universal mechanism for both uniform memory access and non-uniform access, local access and remote access*
  - *Good for a large scale supercomputing with complicated memory access models*
    - *Large scale computer with clusters of smaller components, each component has local memory and also need to access memory in other components*
    - *LINPACK for Supercomputing*
  - *More difficult to program*

Why Java

- Java has built-in threading and synchronization mechanisms
- Java has consistent behavior across different platforms
- Java has good enough performance after JIT was released
  - *Java code does not even include some matrix loop interchange optimizations*

Linkpack benchmark (circa 1998)

Language	C (gcc)	Java (JDK1.1.6 No JIT)	Java (JDK1.2 with JIT)	Limbo (No JIT)	Limbo (With JIT)
No Optimization	0.16	0.86	0.16	4.46	0.30
With Optimization	0.07	0.86	0.15	N/A	N/A

Overview

- Parallelism
- Parallel Architectures
- Parallel Programming
  - *General Purpose Mechanism*
  - *Why Java*
- ***Example: Parallel Sum***

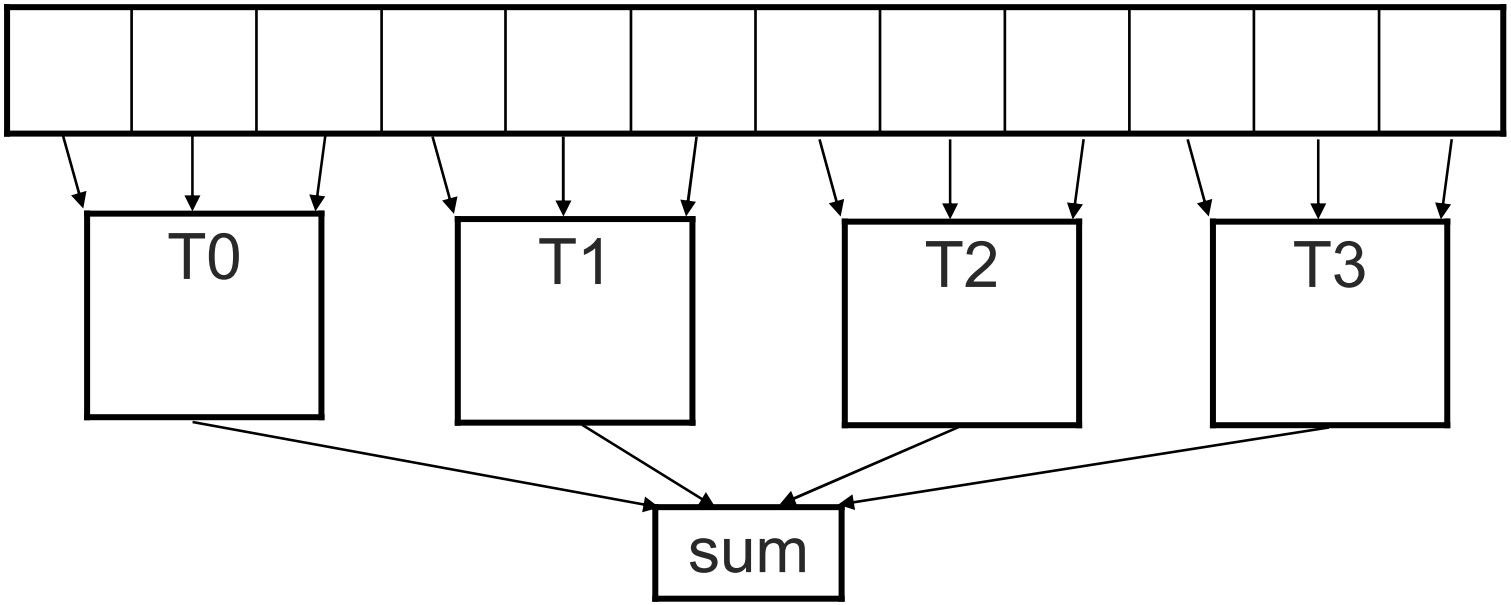
Example: Sum of Array Elements

- Given an array, find the sum of all elements in the array
  - *Use parallel computer to solve the problem*
  - *Assume there is no overflow of sum result*
- Simple algorithm:
  - *Partition the array into m segments, with m threads, each segment corresponding to a thread.*
  - *Each thread calculates the sum of the segment.*

Example: Sum of Array Elements

» Some primitive ideas of algorithm:

- *Partition the array into m segments, with m threads, each segment corresponding to a thread.*
- *Each thread is mapped to a core.*



Example: Sum of Array Elements

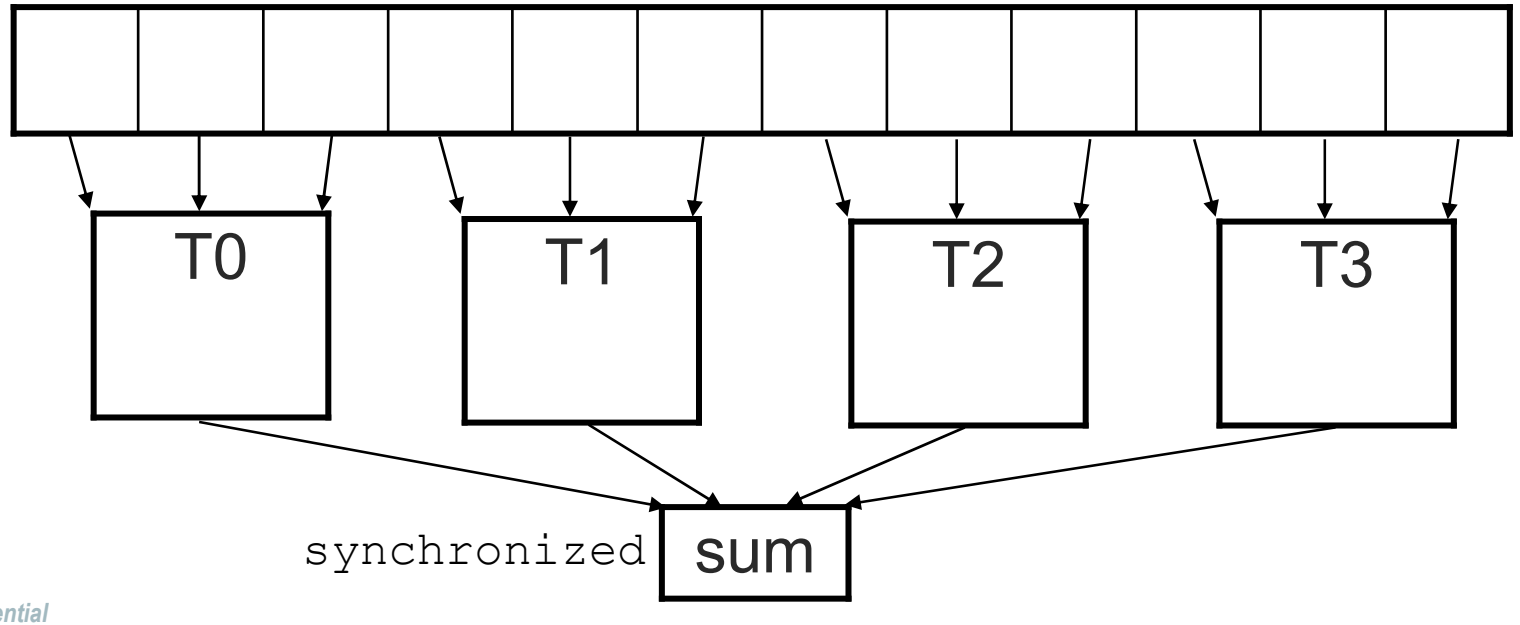
```
» public class ArraySum extends Thread {
»         public static int sizeOfArray = 96000000;
»         public static long[] longArray = new long[sizeOfArray];
»         public static long sum = 0;

»         private static Object lock = new Object();
»         int startPos;
»         int endPos;
»
» public ArraySum(int startPos, int endPos) {
»         this.startPos = startPos;
»         this.endPos = endPos;
» }

» public void run() {
»         for (int i = startPos; i < endPos; i++) {
»                 sum += longArray[i];
»         }
» }
```

Example: Sum of Array Elements, main() program

```
» public static void main(String args[]) {
»         int numOfThreads = Integer.parseInt(args[0]);
»
»         Thread[] threadPool = new Thread[numOfThreads];
»         int len = ArraySum.sizeOfArray/numOfThreads;
»         for (int i = 0; i < numOfThreads; i++) {
»                 threadPool[i] = new ArraySum(i*len, (i+1)*len);
»                 threadPool[i].start();
»         }
»
»         for (int i = 0; i < numOfThreads; i++) {
»                 try {
»                         threadPool[i].join();
»                 } catch (InterruptedException e) {
»                         // do nothing
»                 }
»         }
» }
```

Example: Sum of Array Elements, Correctness

- Did you find the problem with the program?
- The "`sum`" variable needs to be locked. Otherwise, the result might be incorrect because some threads may fetch the "sum" in the same time
  - *Need to apply synchronization primitive to "sum" variable*
  - synchronized (lock), where "lock" is lock for "sum"

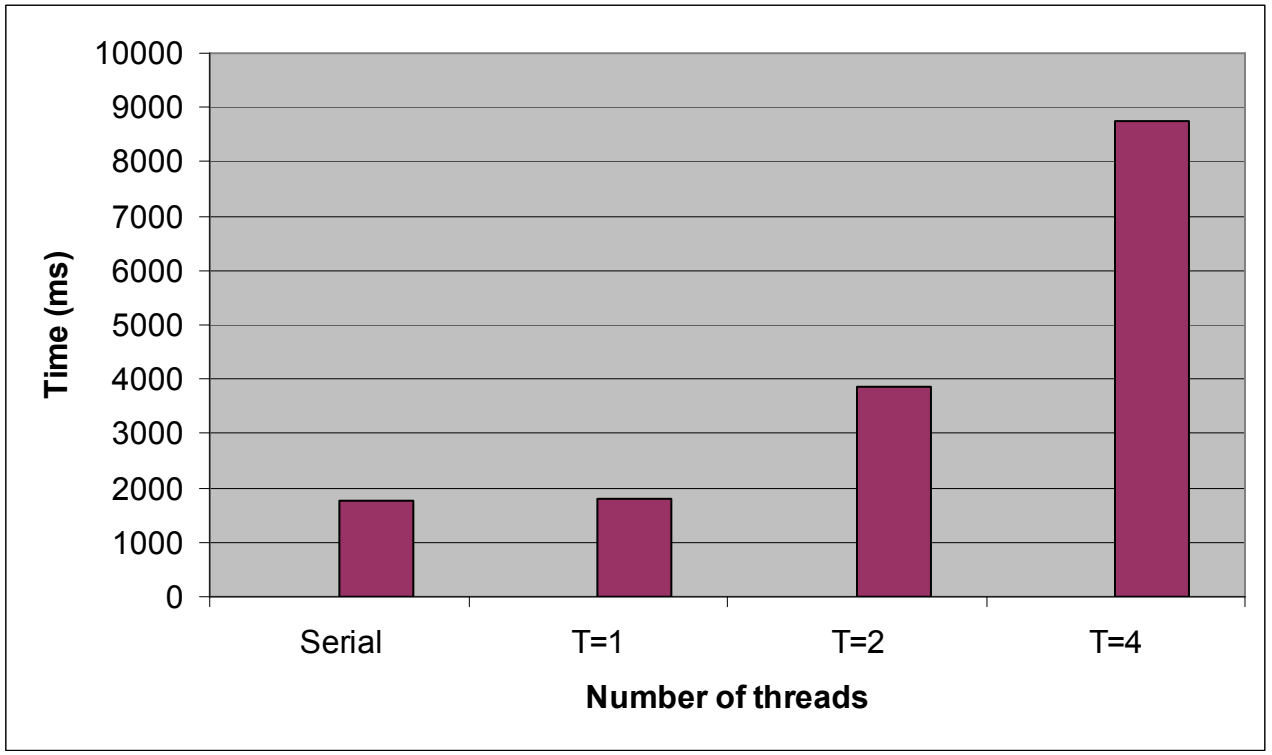


Example: Sum of Array Elements

```
public class ArraySum extends Thread {
        public static int sizeOfArray = 96000000;
        public static long[] longArray = new long[sizeOfArray];
        public static long sum = 0;
        private static Object lock = new Object();
        int startPos;
        int endPos;

public ArraySum(int startPos, int endPos) {
        this.startPos = startPos;
        this.endPos = endPos;
}

public void run() {
        for (int i = startPos; i < endPos; i++) {
            - synchronized (lock) {
                    sum += longArray[i];
            - }
        }
}
```

Example: Sum of Array Elements, Performance

- Configuration: 4-core, 2Ghz, 2G RAM, array is of 96M elements with “long” type
- Global lock on the “`sum`” variable causes significant performance degradation

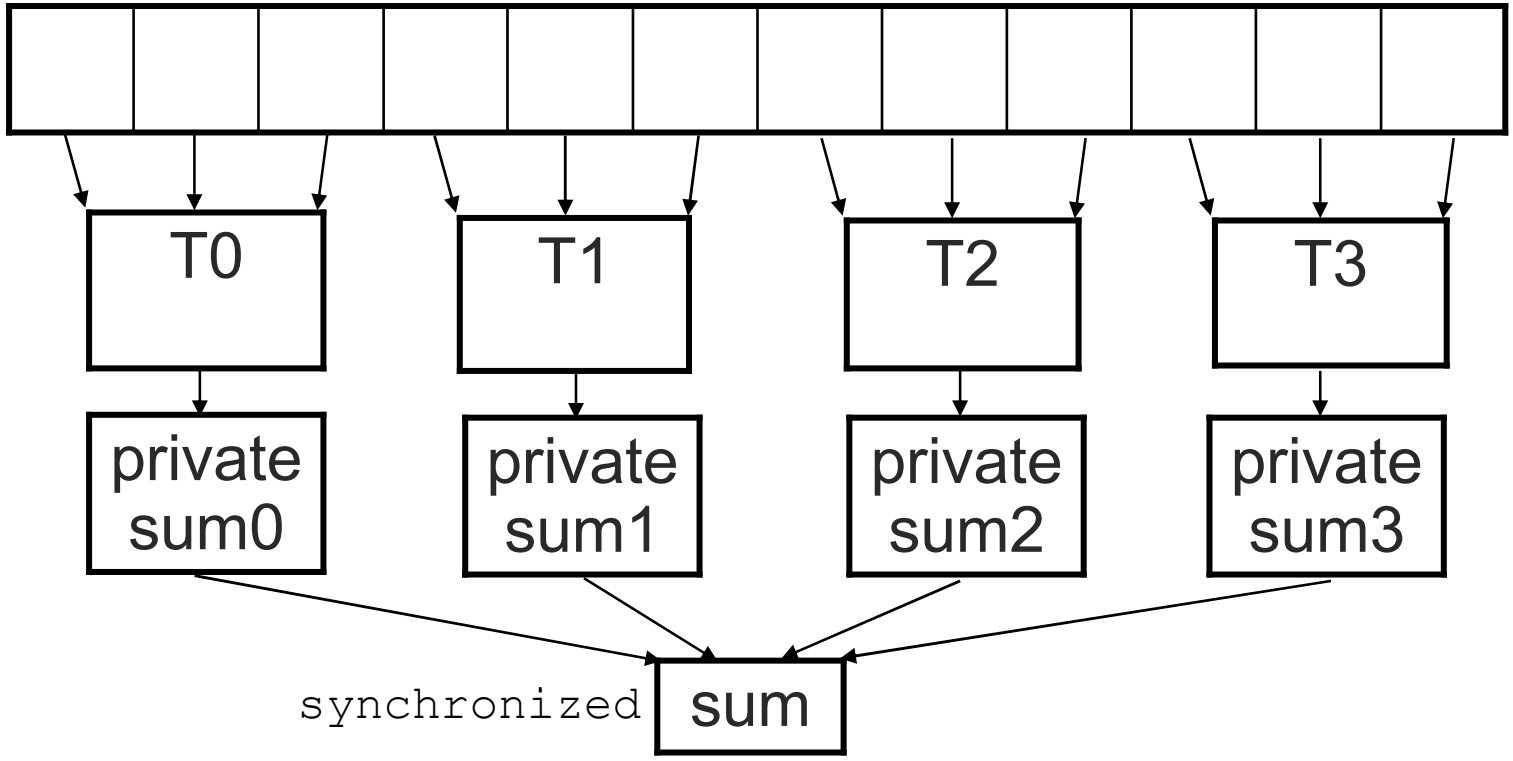


Example: Sum of Array Elements, Performance

- The program behaves correctly but the performance is *very* bad.
- Each “addition” needs a *very* expensive “synchronized” statement on the variable “`sum`”!
- The performance is worse than the serial version of the program because of the overhead cost of threading and synchronization. Each context switching of threads causes tens to hundreds of instructions
- Solution: create a private sum for each thread. No need to lock the private sum.
- Then do locked sum on all private sums

Example: Sum of Array Elements

» Reduction: sum of a few `private_sum` variables and put into a global variable

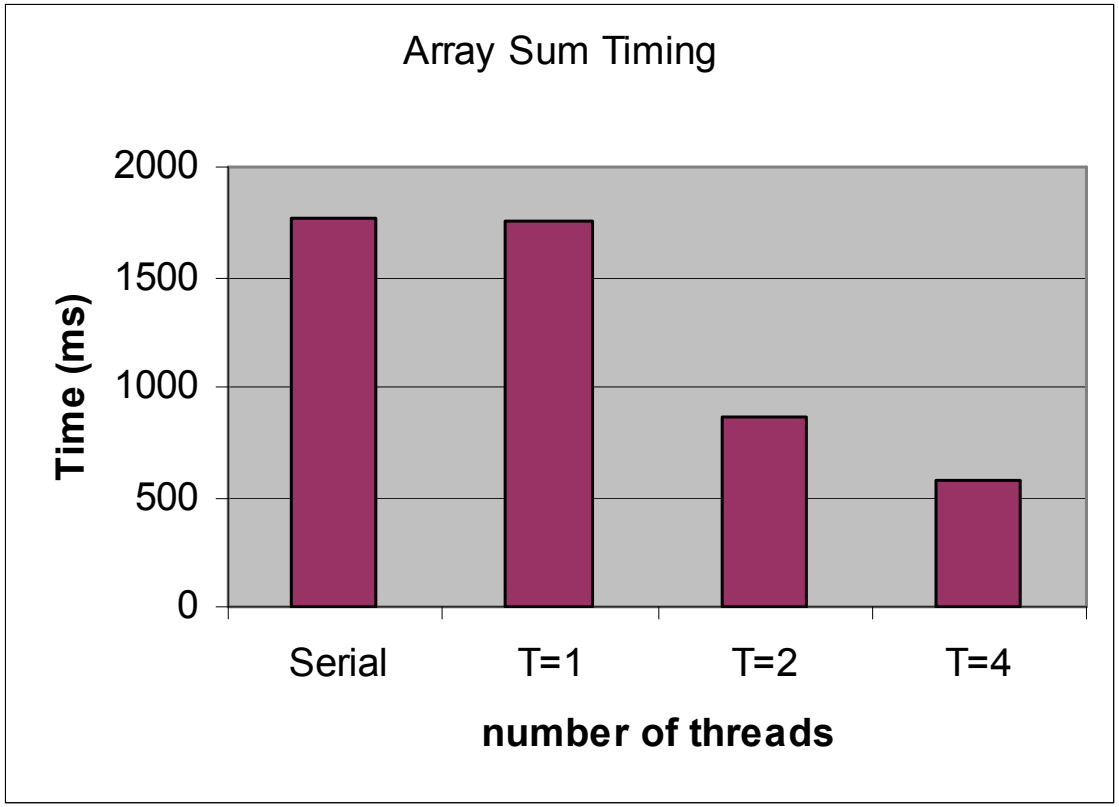


Example: Sum of Array Elements

```
» public class ArraySum extends Thread {
»         public static int sizeOfArray = 96000000;
»         public static long[] longArray = new long[sizeOfArray];
»         public static long sum = 0;
»         private static Object lock = new Object();
»         public long privateSum = 0;
»         int startPos;
»         int endPos;
»
» public ArraySum1(int startPos, int endPos) {
»         this.startPos = startPos;
»         this.endPos = endPos;
» }
» public void run() {
»         for (int i = startPos; i < endPos; i++) {
»                 privateSum += longArray[i];
»         }
»         synchronized (lock) {
»                 sum += privateSum;
»         }
» }
```

Example: Sum of Array Elements

» Timing configuration: 4-core 2Ghz, 2G RAM, array is of 96M elements with “long” type



Example: Sum of Array Elements Further Improvement

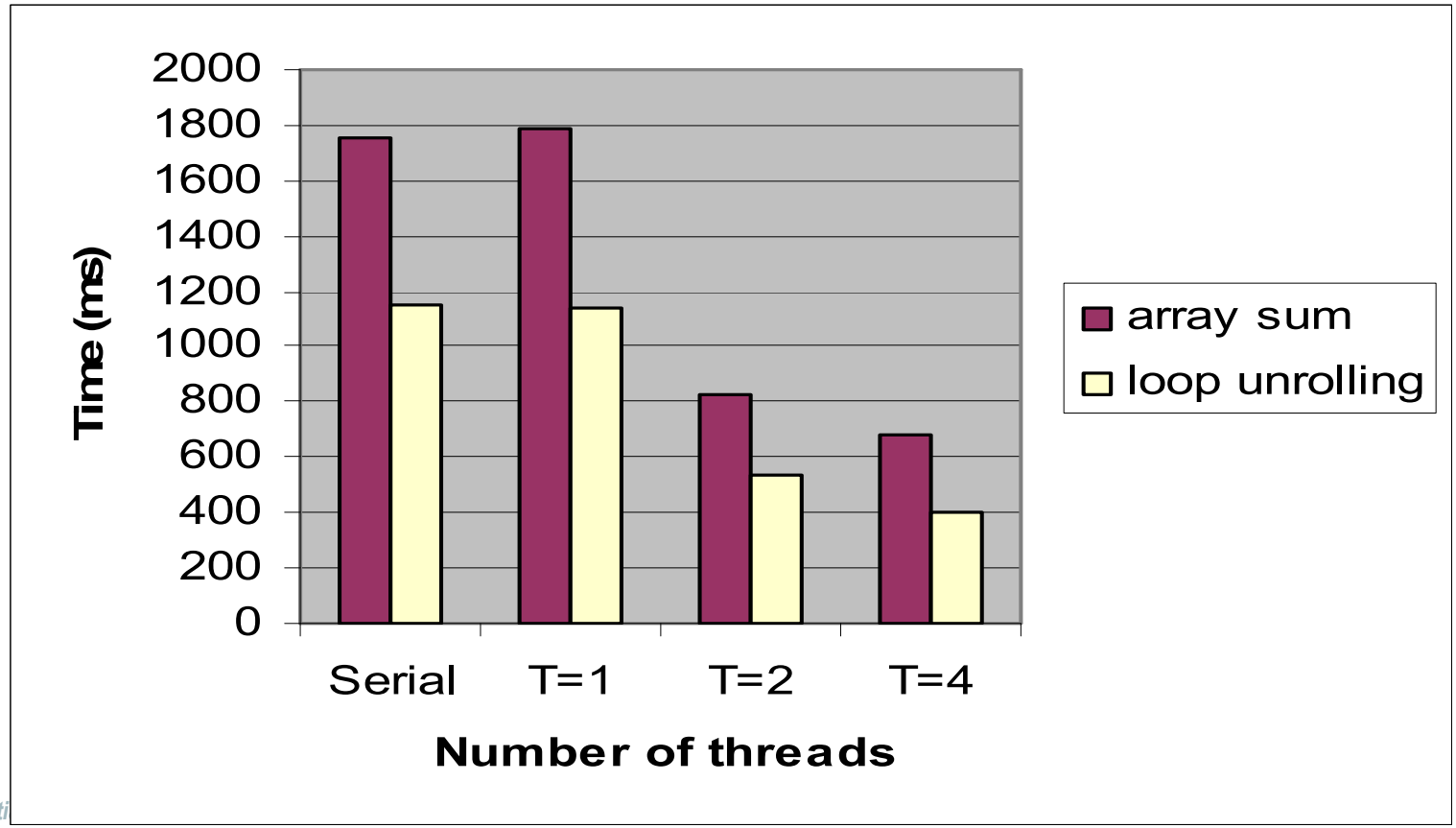
- Loop Unrolling, nothing significantly related to parallel computing but a general programming practice that applies well to parallel computing when a loop is simply applied to an array
  - *Unrolled statements have not data dependency*
  - *Less time spent on array index manipulations*
  - *Manual unrolling better than compiler*

Example: Sum of Array Elements Loop Unrolling

```
public class ArraySum extends Thread {
        public static int sizeOfArray = 96000000;
        public static long[] longArray = new long[sizeOfArray];
        public static long sum = 0;
        private static Object lock = new Object();
        public long privateSum = 0;
        int startPos;
        int endPos;
public ArraySum1(int startPos, int endPos) {
        this.startPos = startPos;
        this.endPos = endPos;
}
public void run() {
        for (int i = startPos; i < endPos; i+=8) {
                long x0=longArray[i]; long x1 = longArray[i+1]; long x2=longArray[i+2]; long x3=longArray[i+3];
                long x4=longArray[i+4]; long x5=longArray[i+5];long x6=longArray[i+6];long x7=longArray[i+7];
                privateSum += x0+x1+x2+x3+x4+x5+x6+x7;
        }
        synchronized (lock) {
                sum += privateSum;
        }
}
```

Example: Sum of Array Elements Further Improvement

- Config: 4-core, 2Ghz, 2G RAM, 96M elements array with "`long`" type
- Loop unrolling

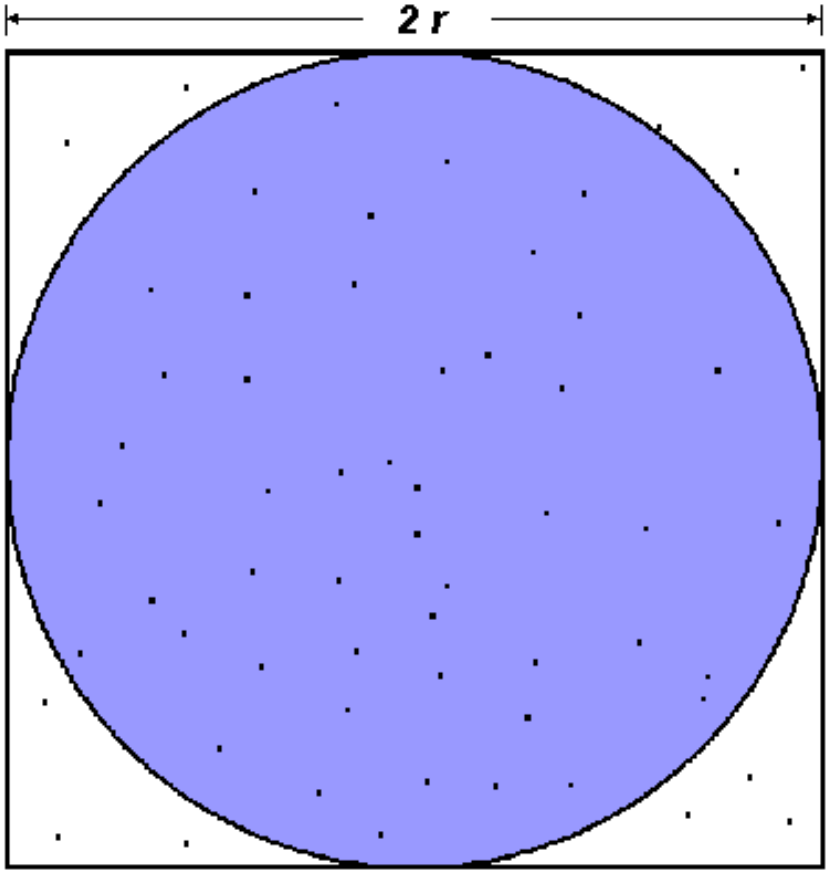


Example: Sum of Array Elements Loop Unrolling

- Performance gains 40%
  - *In older machines, loop unrolling would get ~20% performance gain at best. But in modern processors, performance can get even better*
    - *Avoid pipeline stalls*
    - *Reduce branches*
    - *Increase instruction level parallelism*

Example: Monte-Carlo Calculation of PI

- The value of PI can be calculated in a number of ways. The following method of approximating PI is called Monte-Carlo methods: you don't calculate, you simulate
  - *Inscribe a circle in a square*
  - *Randomly generate points in the square*
  - *Determine the number of points in the square that are also in the circle*
  - *Let r be the number of points in the circle divided by the number of points in the square*
  - *PI ~ 4 r*



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Example: Monte-Carlo Calculation of PI

- Monte-Carlo Calculation is not algorithmic, rather a simulation
  - *Normally when you can't think of a good algorithm*
  - *The more points generated, the better the approximation*
  - *The random number generator can distribute evenly across entire studied data space*
  - *Performance is bad. Compared with a PI algorithm (such as super-pi), Monte-Carlo is a few order of magnitude of slower*

Example: Monte-Carlo Calculation of PI

```
» int npoints = 1000000;
» int circleCcount = 0;
» for (int =0; i< npoints; i++) {
»     x = random(); y = random();
»     if (((x – 1)*(x-1)+(y-1)*(y-1)) <1) {
»         // if the point is inside the circle
»         circleCount++;
» }
» PI = 4.0*circleCount/npoints;
```

Example: Monte-Carlo Calculation of PI: Parallelism

- Monte-Carlo Calculation is extremely easy to be parallelized
  - *Every loop can be a separate thread*
  - *No data dependency*
  - *Just partition the for loop, no need to partition the data set*
  - *An extreme example of task parallelism*
  - *Almost no synchronization nor barrier are needed*
  - *Almost linear speedup*

Summary

- Parallelism is useful
  - *Faster*
- Not all things can be parallelized
  - *Fibonacci series:* `F(n+1) = F(n)+F(n-1)`
    - *Can't get a straight-forward parallel algorithm unless you have some deep algorithm knowledge*
  - *There're problems inherently sequential. In complexity theory, it is still unknown whether **NC** = **P**, but most researchers suspect this to be false, meaning that there are probably some tractable problems which are "inherently sequential"*
- Data Parallel vs Task Parallel
  - *Task Parallelism is relatively easy to be realized, the main focus is on throughput*
  - *Data Parallelism is harder, need to partition the data, design the right algorithms*