Introduction to Parallel Processing

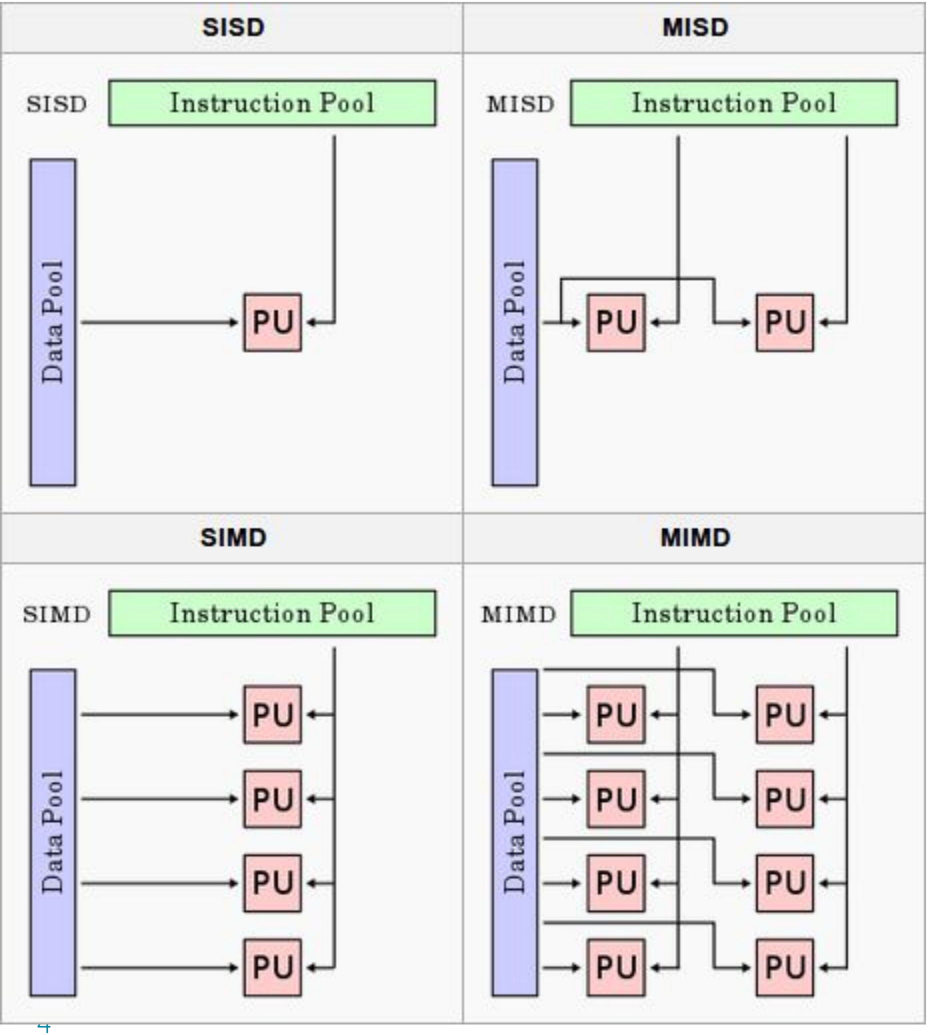
Isaac van Til
Texas State University - CS 3339
Fall 2018

Overview

- Superpipeline / Superscalar
- Multi-core architecture
- Introduction to OpenMP and Multithreading

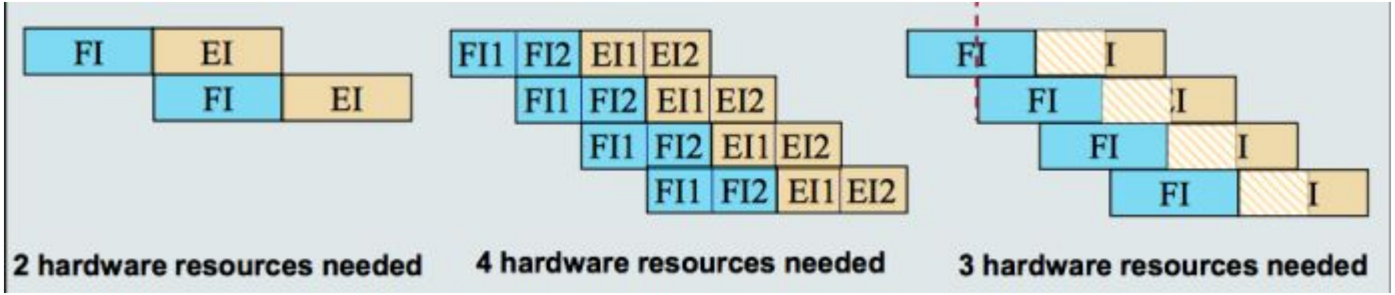
1. Superscalar Processors

Flynn’s Taxonomy



Superpipelining

- Divide the stages of a pipeline into several sub-stages, increases the number of instructions which are handled by the pipeline at the same time


FI
EI
FI
EI
FI1
FI2
EI1
EI2
FI1
FI2
EI1
EI2
FI1
FI2
EI1
EI2
FI1
FI2
EI1
EI2
FI
I
FI
I
FI
I
FI
I
2 hardware resources needed
4 hardware resources needed
3 hardware resources needed

Superpipelining Issues

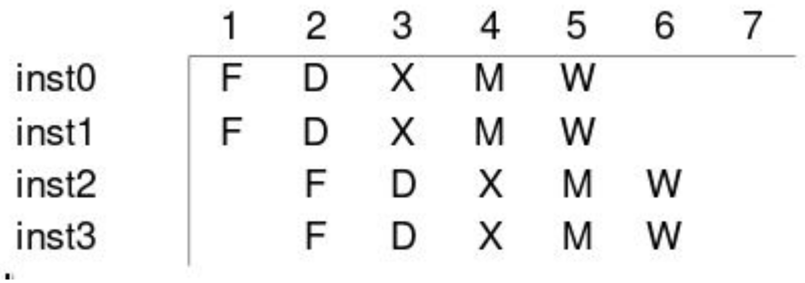
- Increasing the number of stages/substages beyond an optimal limit reduces performance
- Not all stages can be divided into equal-length sub-stages
- Much more complex hardware
- Hazards more difficult to resolve
- Clock skew problem

The “Flynn Bottleneck”

- Single issue performance limit is CPI = 1
- With hazards, CPI >= 1
- Diminishing returns from superpipelining

Solution:

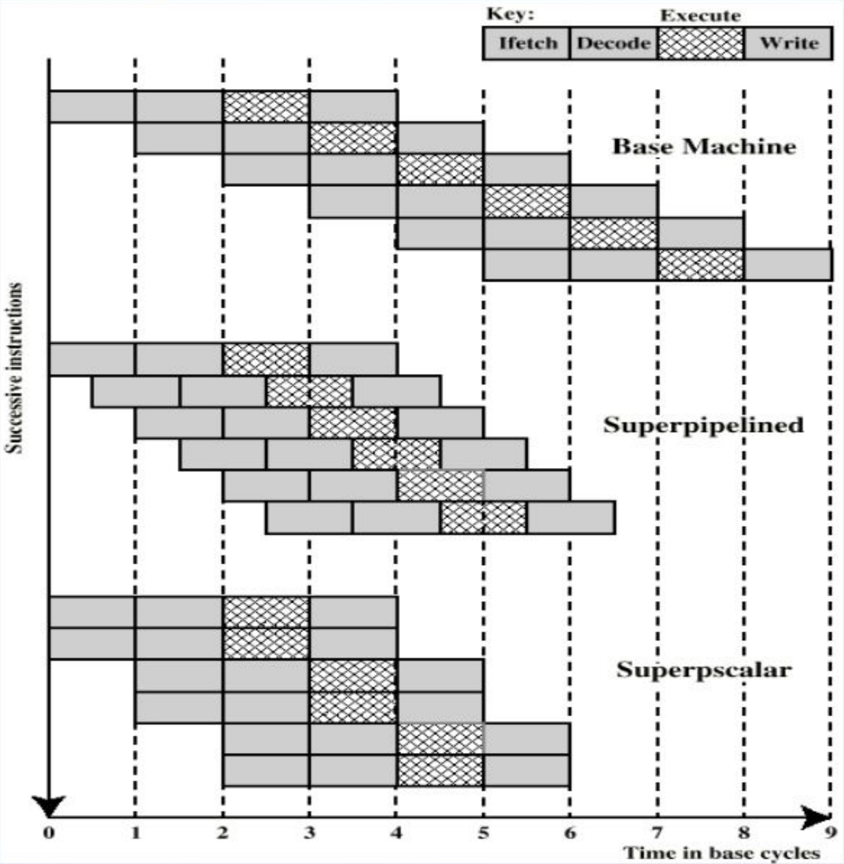
- Issue multiple instructions per cycle
- Instruction-level parallelism (ILP)
- Superscalar architecture



	1	2	3	4	5	6	7
inst0	F	D	X	M	W		
inst1	F	D	X	M	W		
inst2		F	D	X	M	W	
inst3		F	D	X	M	W	

Superscalar vs. Superpipeline

- Base machine: *4*-stage pipeline
  - Instruction fetch
  - Operation decode
  - Operation execution
  - Result write back
- Superpipeline of degree 2
  - A sub-stage often takes half a clock cycle to finish.
- Superscalar of degree 2
  - Two instructions are executed concurrently in each pipeline stage.
  - Duplication of hardware is required by definition.



Instruction Level Parallelism

- The classic von Neumann processors operate through **control flow** (instructions following each other linearly without regard for what data they involve)
- Superscalar processors utilize **data flow**, analyzing several instructions to find data dependencies and executing instructions in parallel that do not depend on each other. This is Instruction Level Parallelism.

Instruction Level Parallelism Mechanisms

- Multiple-issue
- Pipelining
- Branch prediction
- Out-of-order execution
- Prefetching

Introduction to Superscalar Architecture (SSA)

- SSA improves the performance of the execution of scalar instructions
- Several scalar instructions can be initiated simultaneously and executed independently
- Includes all the features in pipelining, but can execute multiple instructions in each pipeline stage

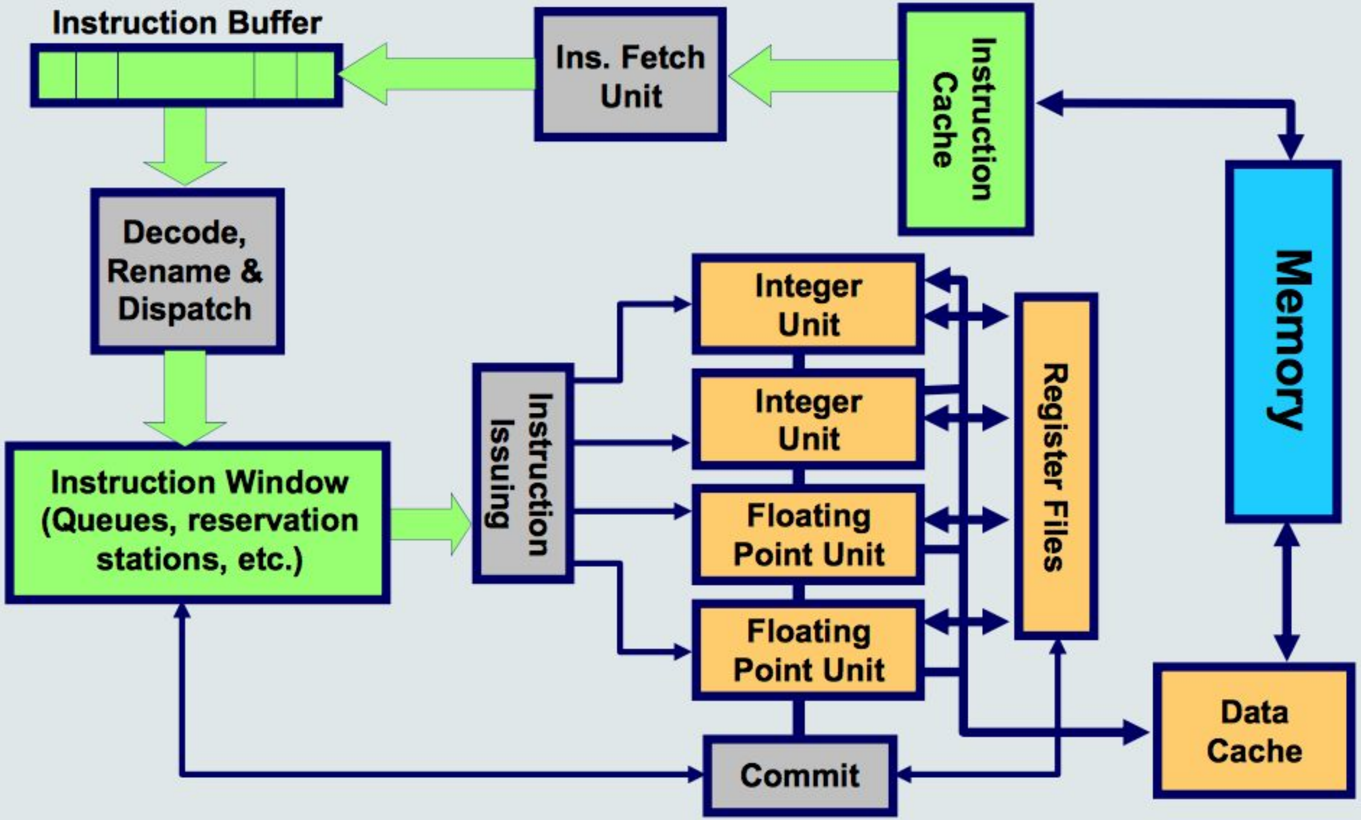
Introduction to Superscalar Architecture (SSA)

- Processors studied so far are fundamentally limited to CPI >= 1
- Superscalar processors enable CPI < 1 (IPC > 1) by executing multiple instructions in parallel
- Can have both in-order and out-of-order superscalar processors

Implementation (in-order)

- Several instructions are issued and completed per clock cycle
- Several pipelines working in parallel
- Depending on the number and kind of parallel units available, a certain number of instructions can be executed in parallel
- Each unit is also pipelined and can execute several operations in different pipeline stages

An SSA Example



Implementation (out-of-order)

- A SSA processor fetches multiple instructions at a time, and attempts to find nearby instructions that are independent of each other
- Based on dependency analysis, the processor may issue and execute instructions in an order that differs from the original machine code
- The processor may eliminate some unnecessary dependencies by the use of additional registers and renaming of register references

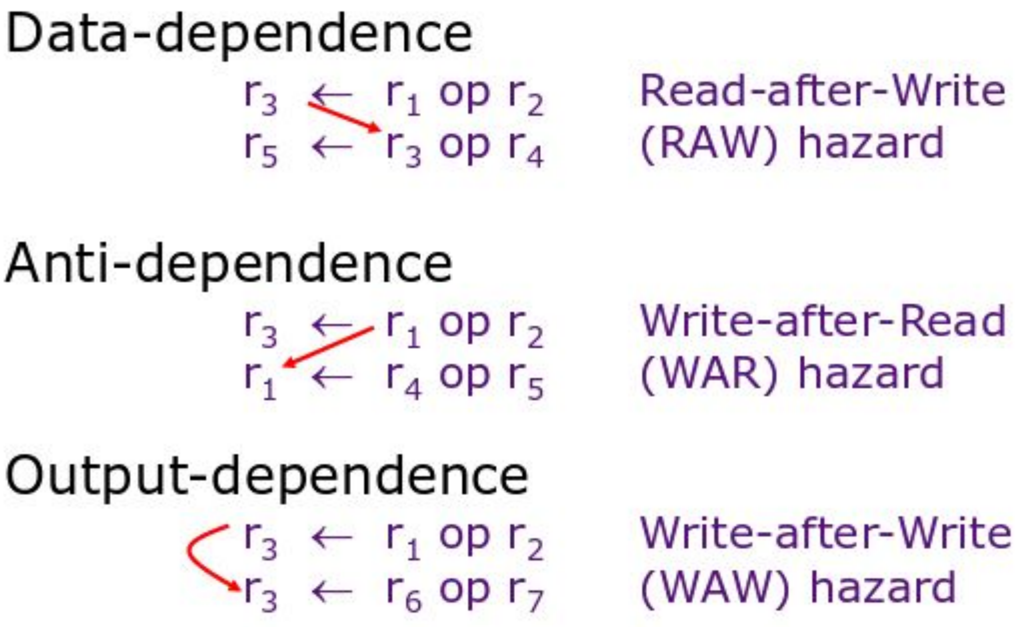
Advantages of Superscalar

- Hardware solves everything (detects possible parallelism, register renaming)
- Binary compatibility
  - Programs don’t need to be changed if functional units are added (without changing the instruction set)
  - Old programs benefit, the hardware simply issues the instructions more efficiently

Problems with Superscalar

- Complex hardware
- Power consumption
- Limited capacity to detect large numbers of parallel instructions
- Limited degree of intrinsic parallelism (instructions requiring the same CPU resources)

Data Hazards - review



Data-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
$r_5 \leftarrow r_3 \text{ op } r_4$

Read-after-Write (RAW) hazard

Anti-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
$r_1 \leftarrow r_4 \text{ op } r_5$

Write-after-Read (WAR) hazard

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
$r_3 \leftarrow r_6 \text{ op } r_7$

Write-after-Write (WAW) hazard

Superscalar Pipeline Diagrams - Ideal

scalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)➔r2`	F	D	X	M	W							
`lw 4(r1)➔r3`		F	D	X	M	W						
`lw 8(r1)➔r4`			F	D	X	M	W					
`add r14,r15➔r6`				F	D	X	M	W				
`add r12,r13➔r7`					F	D	X	M	W			
`add r17,r16➔r8`						F	D	X	M	W		
`lw 0(r18)➔r9`							F	D	X	M	W	

2-way superscalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)➔r2`	F	D	X	M	W							
`lw 4(r1)➔r3`	F	D	X	M	W							
`lw 8(r1)➔r4`		F	D	X	M	W						
`add r14,r15➔r6`		F	D	X	M	W						
`add r12,r13➔r7`			F	D	X	M	W					
`add r17,r16➔r8`			F	D	X	M	W					
`lw 0(r18)➔r9`				F	D	X	M	W				

Superscalar Pipeline Diagrams - Realistic

scalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`		F	D	X	M	W						
`lw 8(r1)→r4`			F	D	X	M	W					
`add r4,r5→r6`				F	d*	D	X	M	W			
`add r2,r3→r7`						F	D	X	M	W		
`add r7,r6→r8`							F	D	X	M	W	
`lw 4(r8)→r9`								F	D	X	M	W

2-way superscalar	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1)→r2`	F	D	X	M	W							
`lw 4(r1)→r3`	F	D	X	M	W							
`lw 8(r1)→r4`		F	D	X	M	W						
`add r4,r5→r6`		F	d*	d*	D	X	M	W				
`add r2,r3→r7`			F	d*	D	X	M	W				
`add r7,r6→r8`					F	D	X	M	W			
`lw 4(r8)→r9`					F	d*	D	X	M	W		

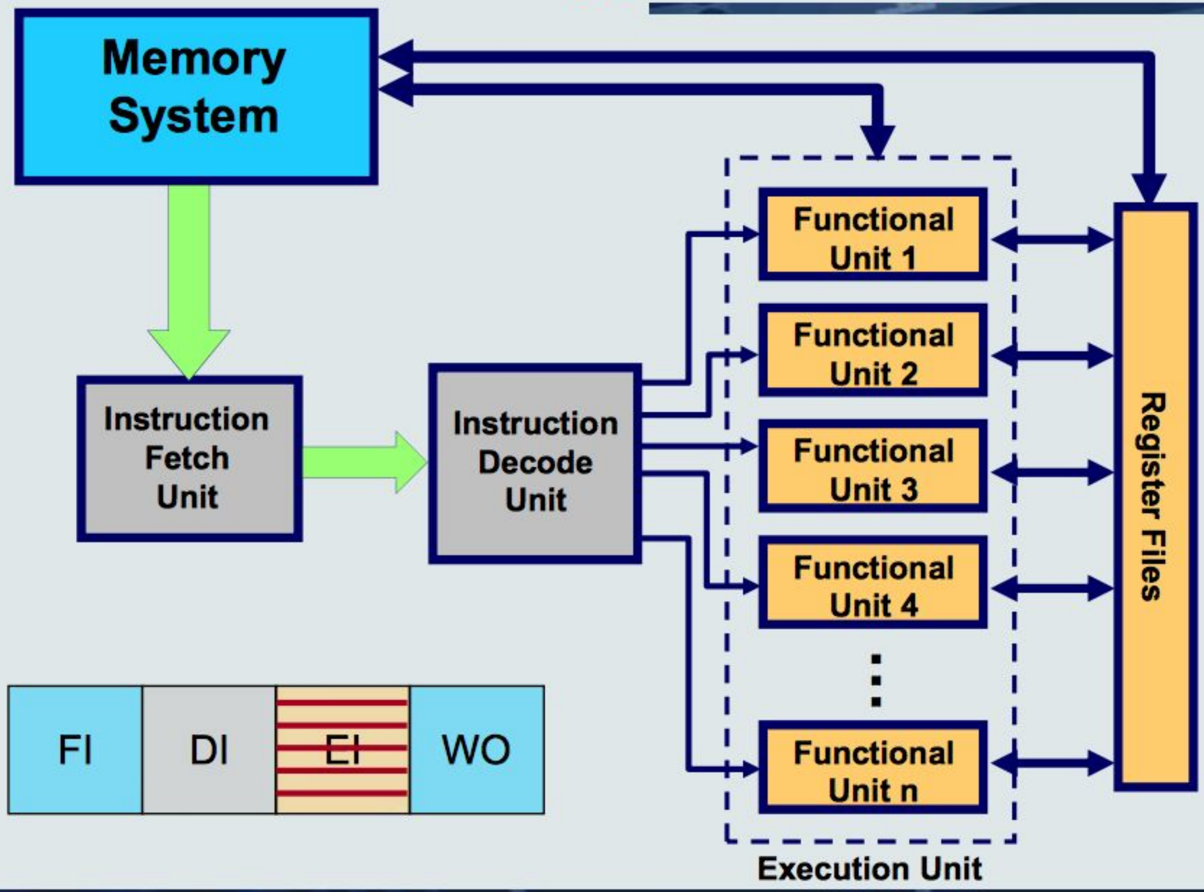
Potential improvements

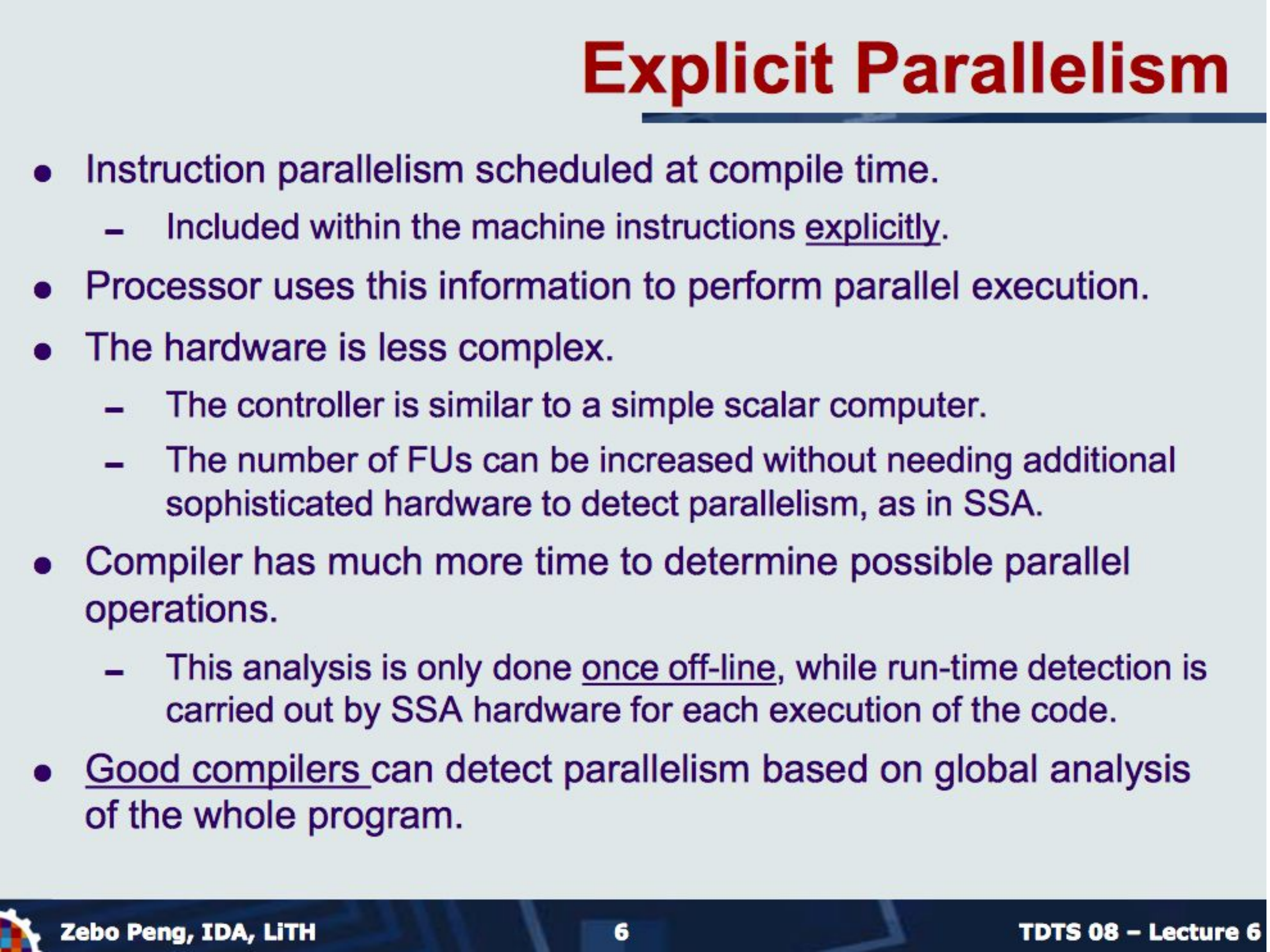
- Very Long Instruction Word Processor (VLIW)
- Explicit Parallelism
- Loop unrolling

Very Long Instruction Word Processors

- Several operations that can be executed in parallel are placed in a single instruction word
- Compile-time parallelism detection
- After one instruction has been fetched all the corresponding operations are issued in parallel (no hardware needed for run-time detection of parallelism)
- Compiler can analyze the entire program to detect parallel operations

Typical Organization





Explicit Parallelism

- Instruction parallelism scheduled at compile time.
  - Included within the machine instructions explicitly.
- Processor uses this information to perform parallel execution.
- The hardware is less complex.
  - The controller is similar to a simple scalar computer.
  - The number of FUs can be increased without needing additional sophisticated hardware to detect parallelism, as in SSA.
- Compiler has much more time to determine possible parallel operations.
  - This analysis is only done once off-line, while run-time detection is carried out by SSA hardware for each execution of the code.
- Good compilers can detect parallelism based on global analysis of the whole program.

An Example

```
for (i=959; i>=0; i--)
   x[i] = x[i] + s;
```

Assumptions:

x is an array of floating point values;

s is a floating point constant.

Memory allocation:

- R1 initially contains the address of the last element in **x**; the other elements are at lower addresses; x[0] is at address 0.
- Floating point register F2 contains the value **s**.
- A floating point value is 8 bytes long.

For an ordinary processor, this C code will be compiled to:

```
Loop: LDD   F0,(R1)     F0:= x[i];(load double)
      ADF   F4,F0,F2    F4:= F0+F2;(add floating point)
      STD   (R1),F4     x[i]:= F4;(store double)
      SBI   R1,R1,#8    R1:= R1-8;
      BGEZ  R1,Loop     branch if R1 ≥ 0.
```

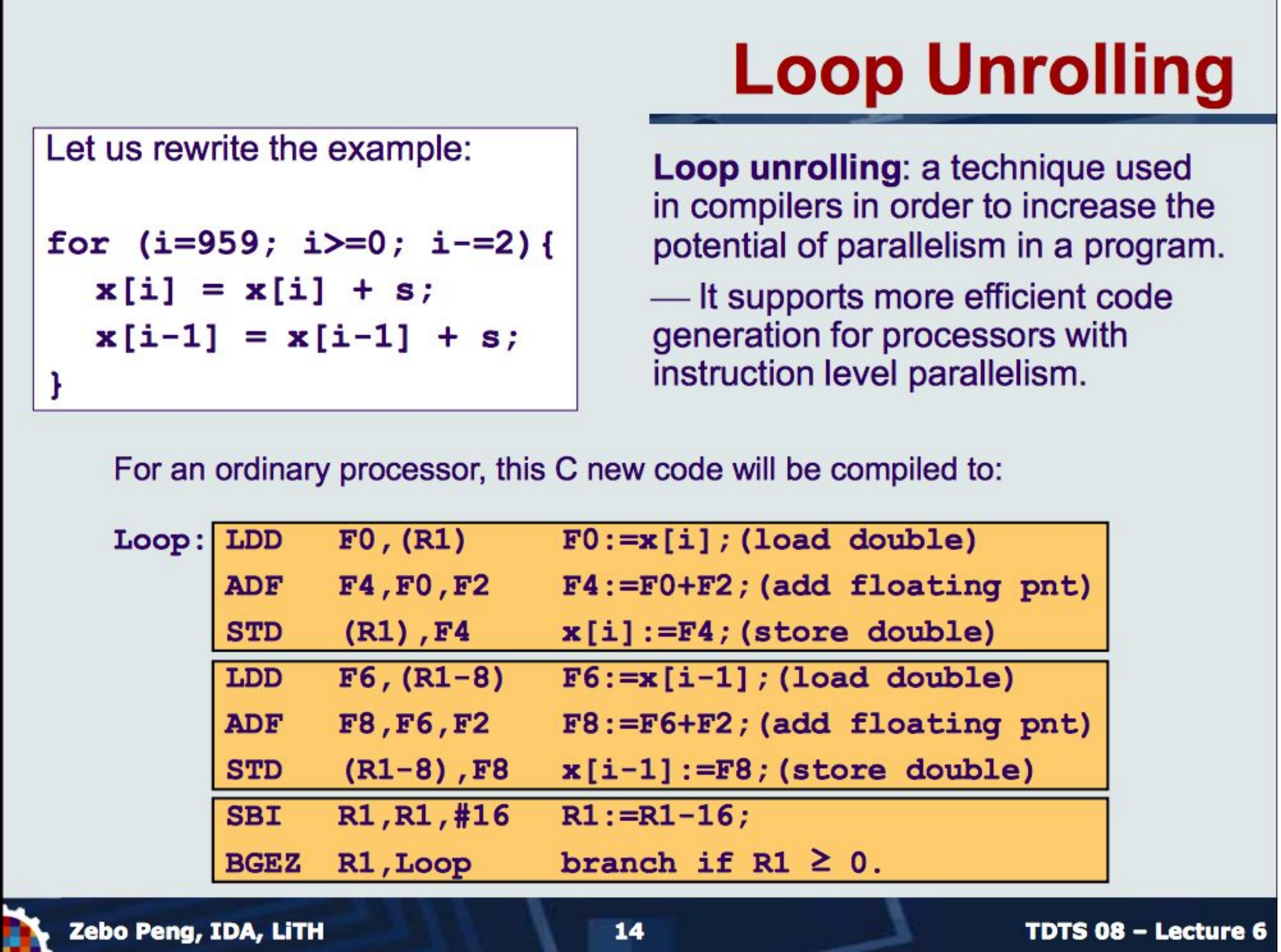
An Example (Cont'd)

```
Loop: LDD    F0,(R1)      load double
      ADF    F4,F0,F2     add FP
      STD    (R1),F4      store double
      SBI    R1,R1,#8
      BGEZ   R1,Loop
```

Note the displacement of 8 for R1 is needed, because we have already subtracted 8 from R1

Cycl	Mem R	Mem R	FP 1	FP 2	I/BRA
1	LDD F0,(R1)				
2					
3			ADF F4,F0,F2		
4					
5					SBI R1,R1,#8
6	STD (R1+8),F4				BGEZ R1,Loop

- One iteration takes 6 cycles; the whole loop takes 960*6 = 5760 cycles.
- Almost no parallelism; most of the fields in the instructions are empty.
- There are two completely empty cycles.



Loop Unrolling

Let us rewrite the example:

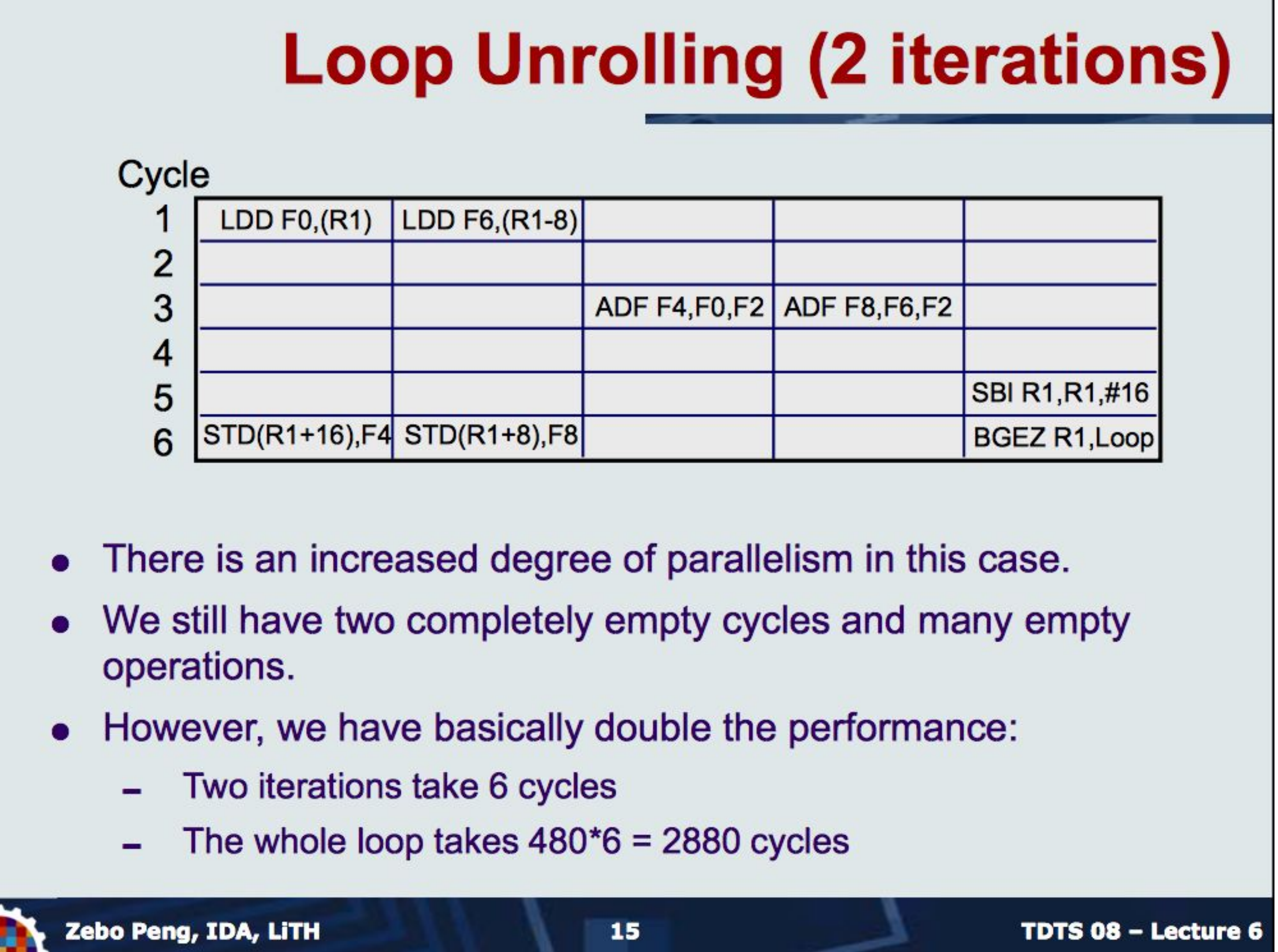
```
for (i=959; i>=0; i-=2){
   x[i] = x[i] + s;
   x[i-1] = x[i-1] + s;
}
```

Loop unrolling: a technique used in compilers in order to increase the potential of parallelism in a program.

— It supports more efficient code generation for processors with instruction level parallelism.

For an ordinary processor, this C new code will be compiled to:

```
Loop: LDD   F0,(R1)      F0:=x[i];(load double)
      ADF   F4,F0,F2     F4:=F0+F2;(add floating pnt)
      STD   (R1),F4      x[i]:=F4;(store double)
      LDD   F6,(R1-8)    F6:=x[i-1];(load double)
      ADF   F8,F6,F2     F8:=F6+F2;(add floating pnt)
      STD   (R1-8),F8    x[i-1]:=F8;(store double)
      SBI   R1,R1,#16    R1:=R1-16;
      BGEZ  R1,Loop      branch if R1 ≥ 0.
```

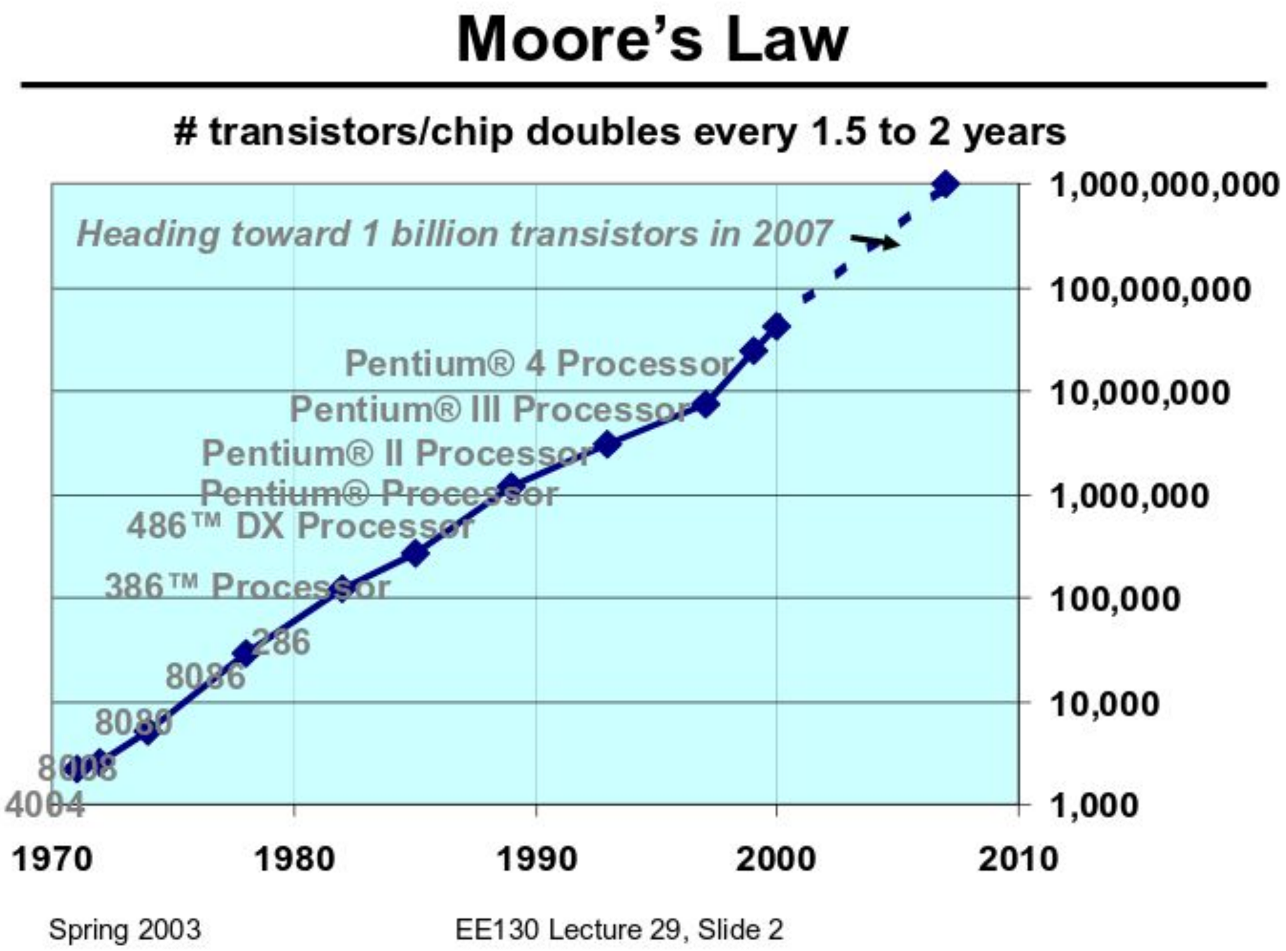


Loop Unrolling (2 iterations)

Cycle					
1	LDD F0,(R1)	LDD F6,(R1-8)			
2					
3			ADF F4,F0,F2	ADF F8,F6,F2	
4					
5					SBI R1,R1,#16
6	STD(R1+16),F4	STD(R1+8),F8			BGEZ R1,Loop

- There is an increased degree of parallelism in this case.
- We still have two completely empty cycles and many empty operations.
- However, we have basically double the performance:
  - Two iterations take 6 cycles
  - The whole loop takes 480*6 = 2880 cycles

2.
Multi-core architecture OpenMP & Multithreading



Moore’s Law

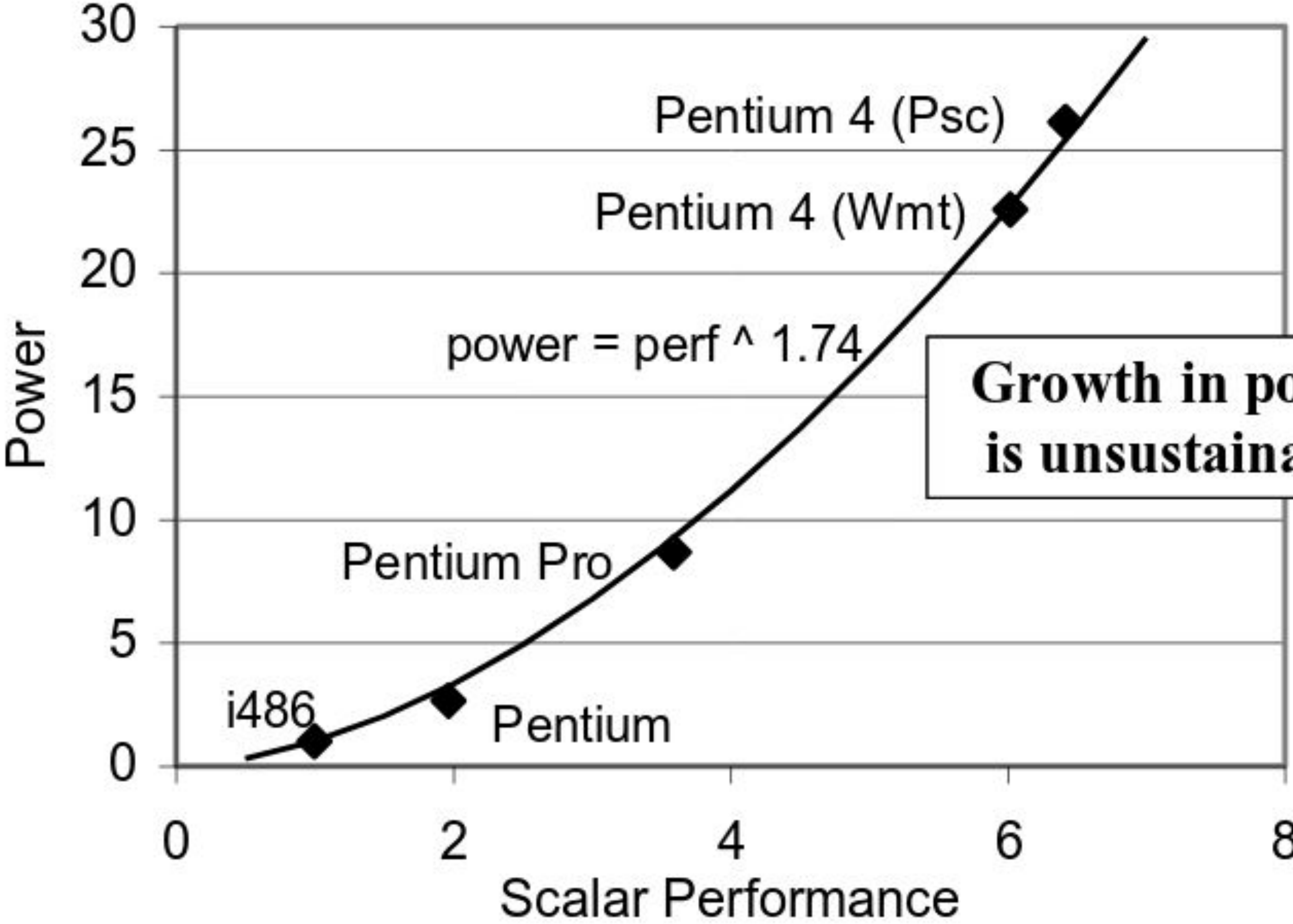
transistors/chip doubles every 1.5 to 2 years

Spring 2003

EE130 Lecture 29, Slide 2

Consequences of Moore's Law

- Performance increased in tandem with transistor density
- People expect that performance and optimization comes from hardware
- Less emphasis is put on software optimization



Growth in power is unsustainable

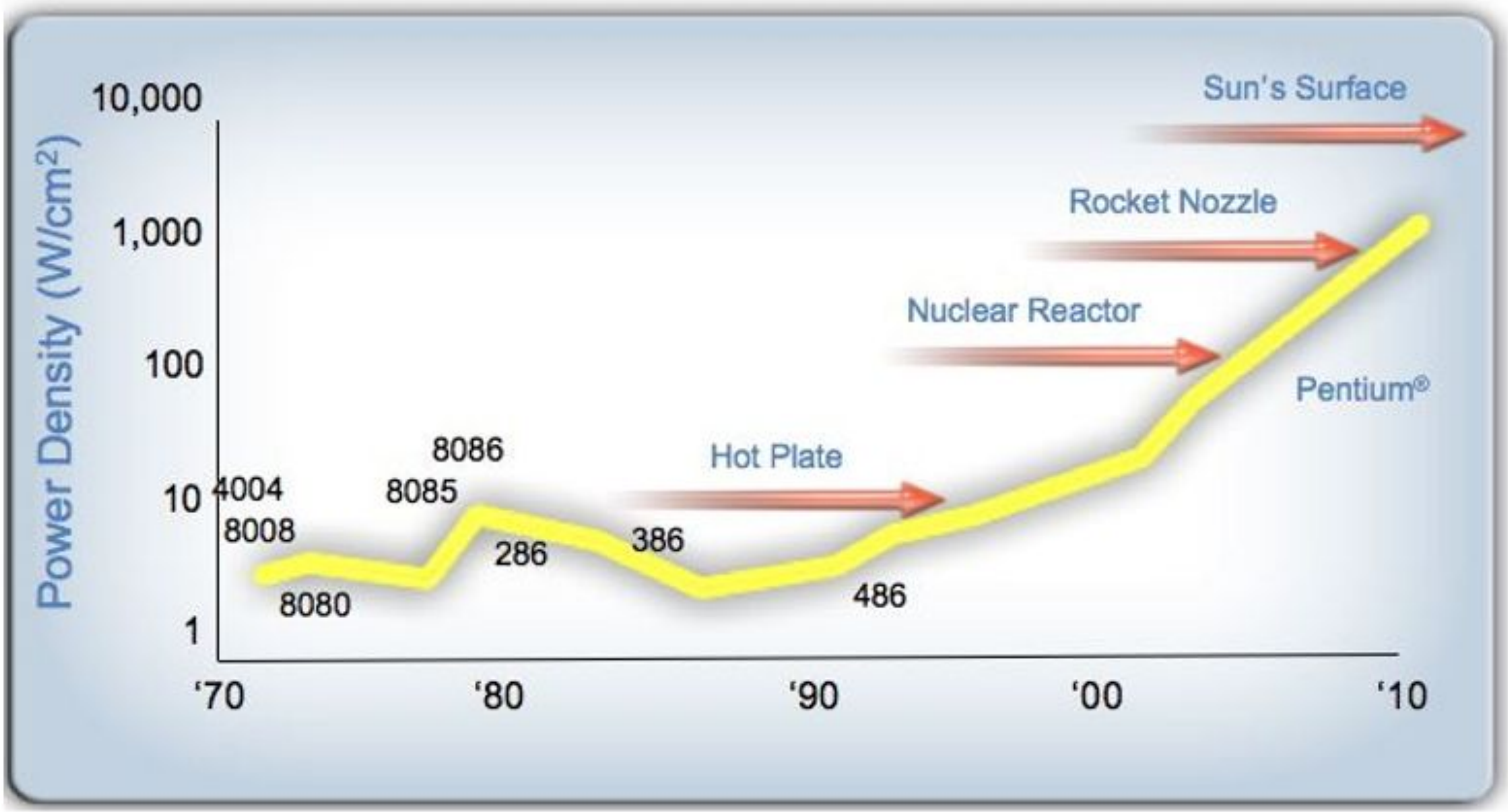
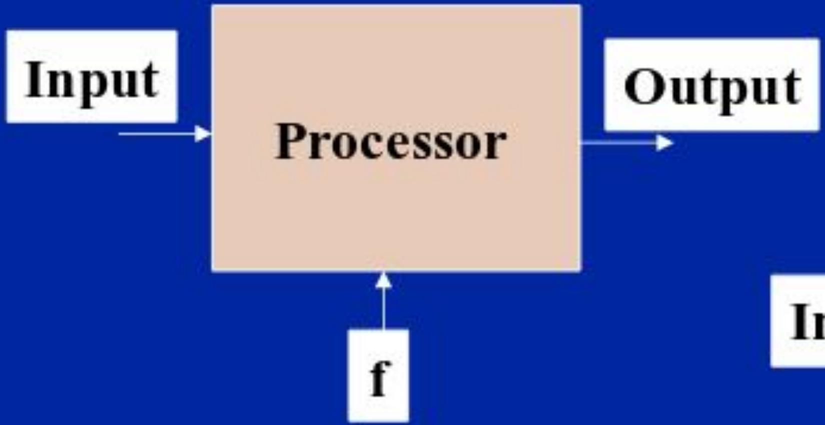


Figure 1.28: Projected heat dissipation of a CPU if trends had continued – this graph courtesy Pat Helsinger

Solution:

- Create simpler cores designed around power optimization instead of performance optimization
- Use more than one of them!



Input

Processor

Output

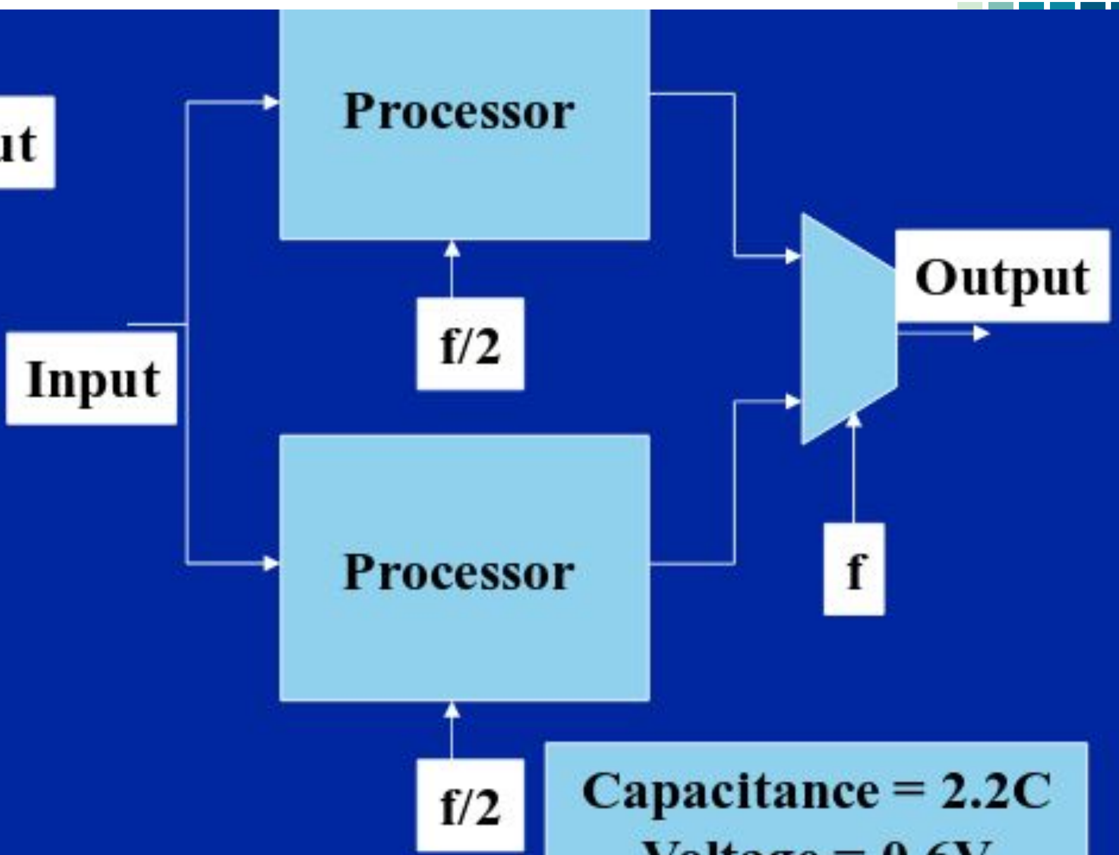
f

Capacitance = C

Voltage = V

Frequency = f

Power = CV^2f



Input

Processor

f/2

Processor

f/2

Output

f

Capacitance = 2.2C

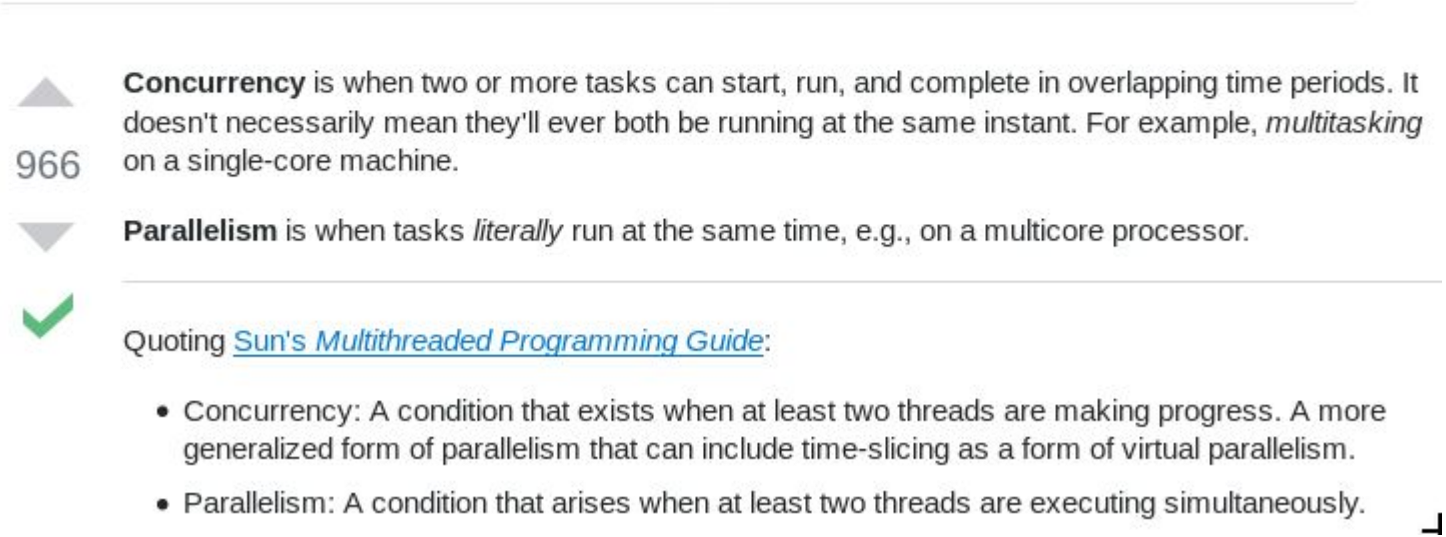
Voltage = 0.6V

Frequency = 0.5f

Power = $0.396CV^2f$

Results:

- Performance now comes from software
- Parallel computing

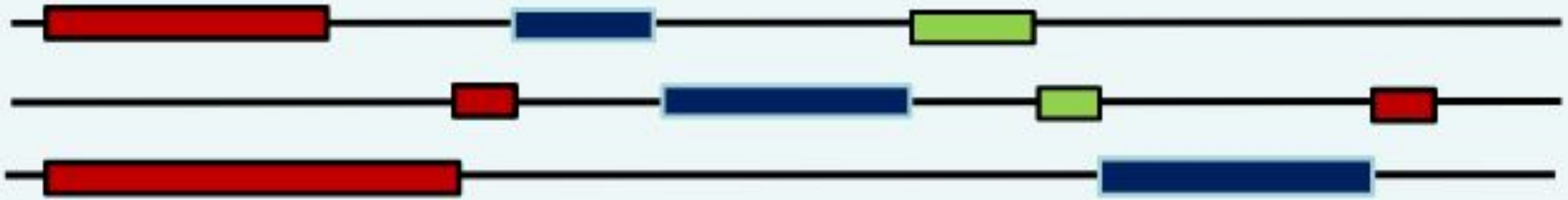


Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, *multitasking* on a single-core machine.

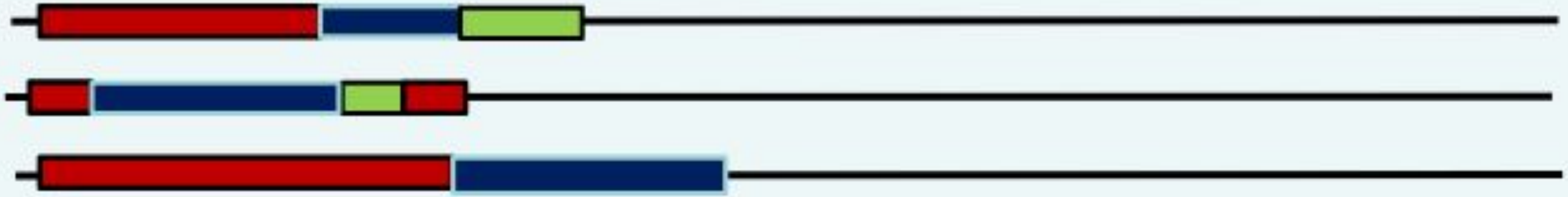
Parallelism is when tasks *literally* run at the same time, e.g., on a multicore processor.

Quoting Sun's *Multithreaded Programming Guide*:

- Concurrency: A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.
- Parallelism: A condition that arises when at least two threads are executing simultaneously.



Concurrent, non-parallel Execution



Concurrent, parallel Execution

An important distinction

- Concurrent problems are inherently concurrent (think web server) and you cannot even define the problem without concurrency
- Parallel problems have things happening at the same time, but you could do it without concurrency
- We will discuss **parallel** applications

Writing Parallel Software

- Step 1: find concurrency
- Step 2: organize into an algorithm that exploits parallelism
- Step 3: implement in a parallel language

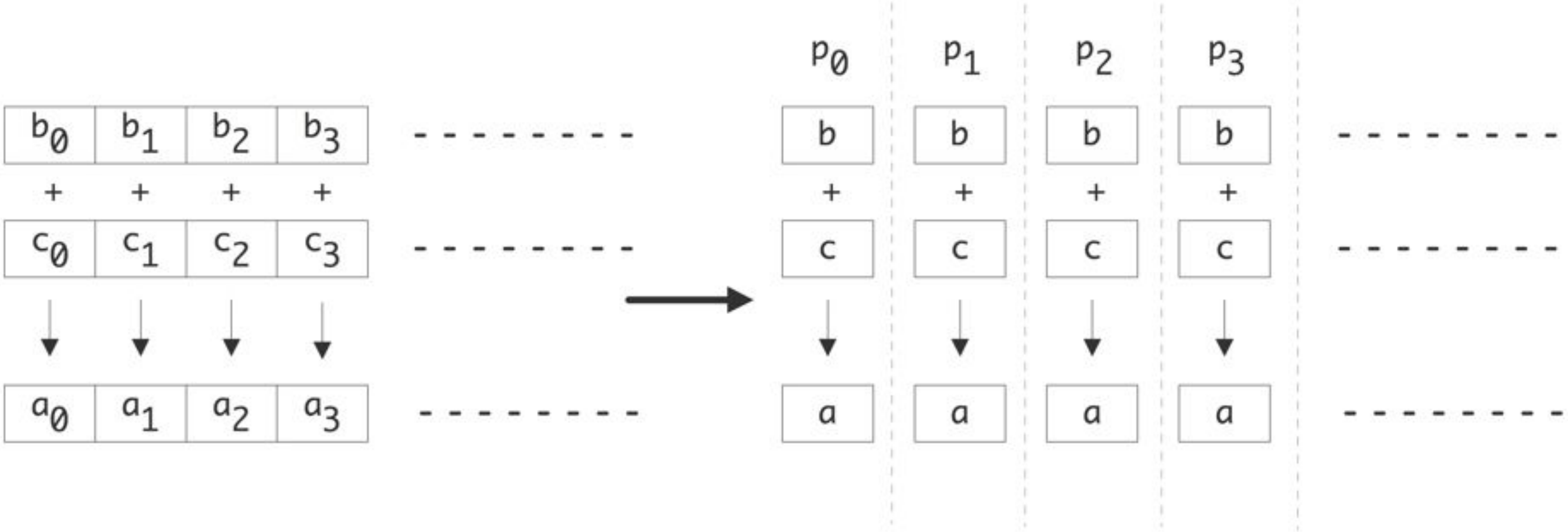
Example: Vector Addition

```
for (i=0; i<n; i++)
   a[i] = b[i] + c[i];
```

Where is the concurrency?

How can we exploit parallelism?

Ideal case (*n* processors):



```
for (i=my_low; i<my_high; i++)
   a[i] = b[i] + c[i];
```

The original algorithm takes *n* time.

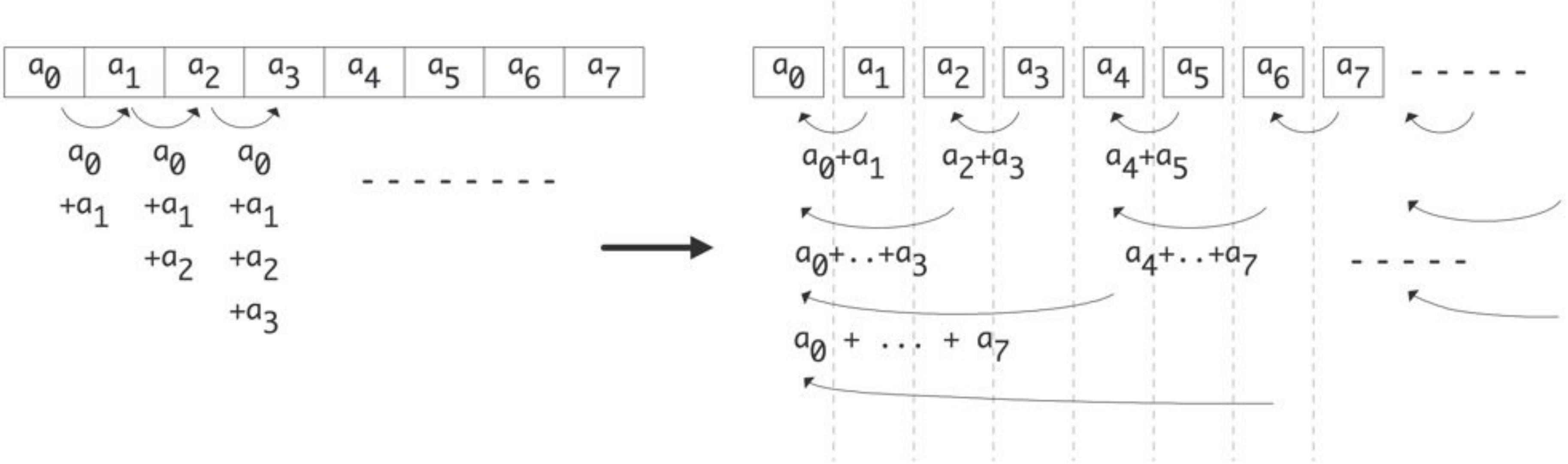
Parallel execution on *p* processors results in *n/p*

The parallel algorithm is faster by a factor of *p.*

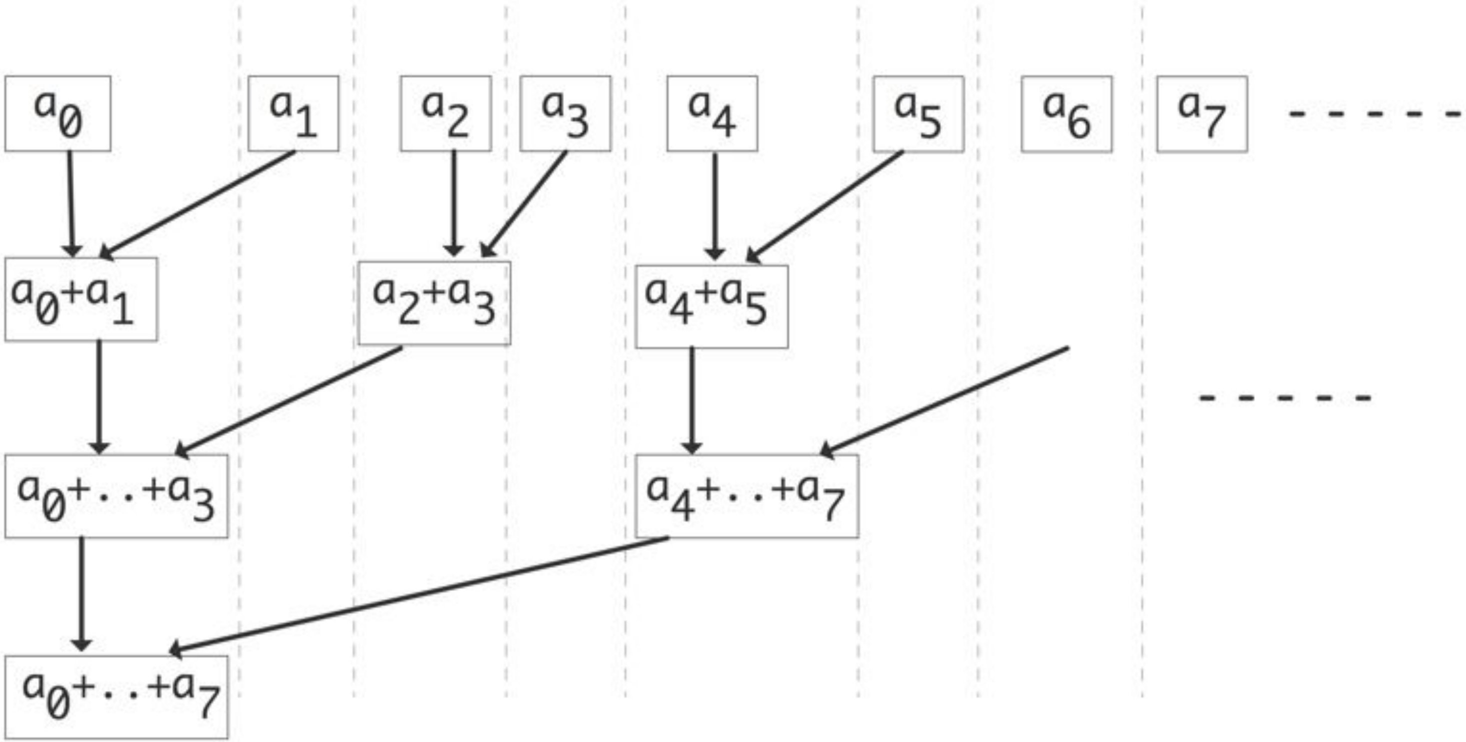
This one is slightly less obvious

```
s = 0;
for (i=0; i<n; i++)
   s += x[i];
```

```
for (s=2; s<2*n; s*=2) {
    for (i=0; i<n-s/2; i+=s)
        x[i] += x[i+s/2];
}
```



a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
a_0 $+a_1$
a_0 $+a_1$ $+a_2$
a_0 $+a_1$ $+a_2$ $+a_3$
- - - - - - - -
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 - - - - -
a_0+a_1
a_2+a_3
a_4+a_5
$a_0+..+a_3$
$a_4+..+a_7$
- - - - -
a_0 + ... + a_7



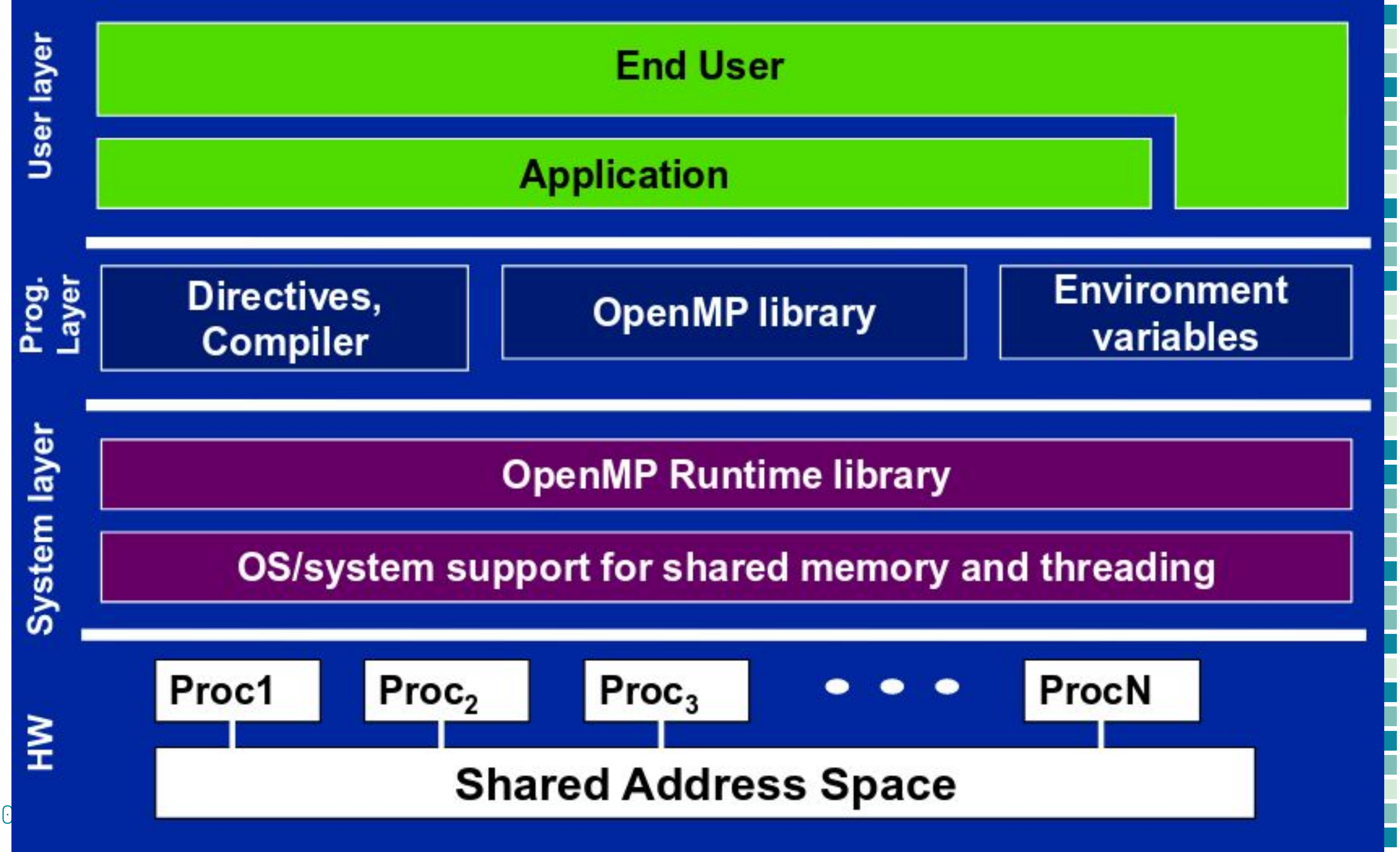
a_0
a_1
a_2
a_3
a_4
a_5
a_6
a_7
- - - - -
a_0+a_1
a_2+a_3
a_4+a_5
- - - - -
$a_0+..+a_3$
$a_4+..+a_7$
$a_0+..+a_7$

Initial observations

- Some algorithms need to be rewritten to make them parallel
- A parallel algorithm may not show perfect speedup
- Communicating data (example 2) between processors takes time

Open Multi-Processing (OpenMP)

- After you have a parallel algorithm, you can use a parallel language like OpenMP to implement it on a parallel computer
- OpenMP is a set of compiler directives and library routines for writing C/C++/FORTRAN in parallel
- Uses low level pthread stuff so you don’t have to!



User layer

End User

Application

Prog. Layer

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

Proc1

Proc$_2$

Proc$_3$

• • •

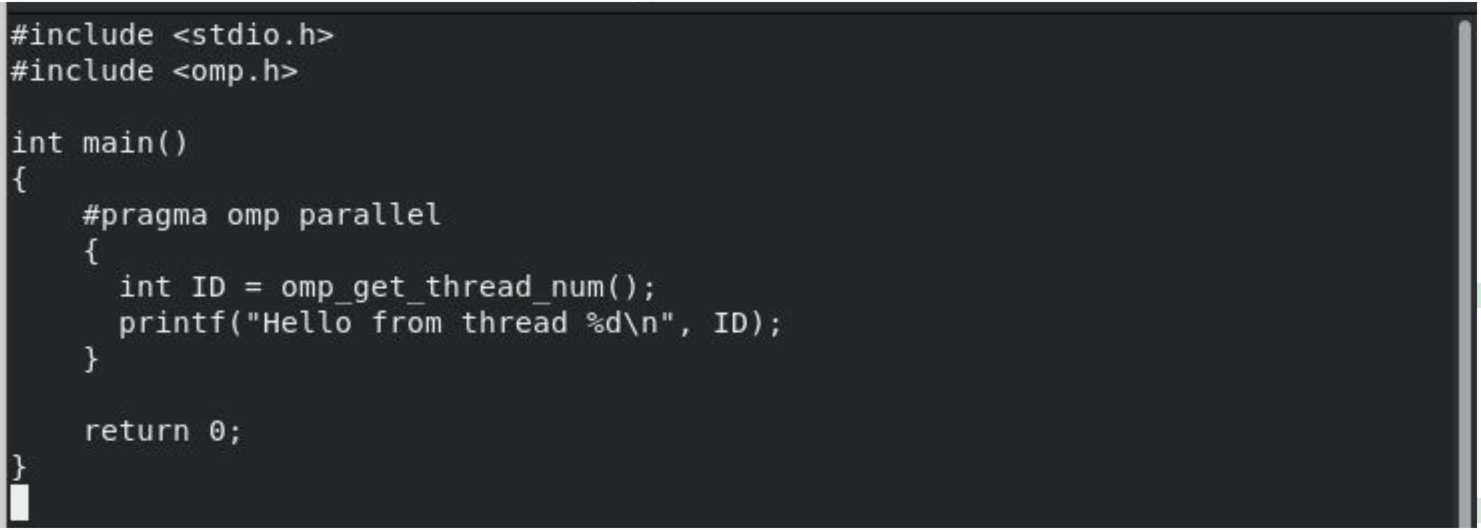
ProcN

Shared Address Space

Basic Syntax

- `#include <omp.h>`
- Most constructs are compiler directives
  - `#pragma omp` *`construct [clause[clause]...]`*
- Most constructs apply to a structured block

```
#pragma omp parallel
{
   do_stuff();
}
```

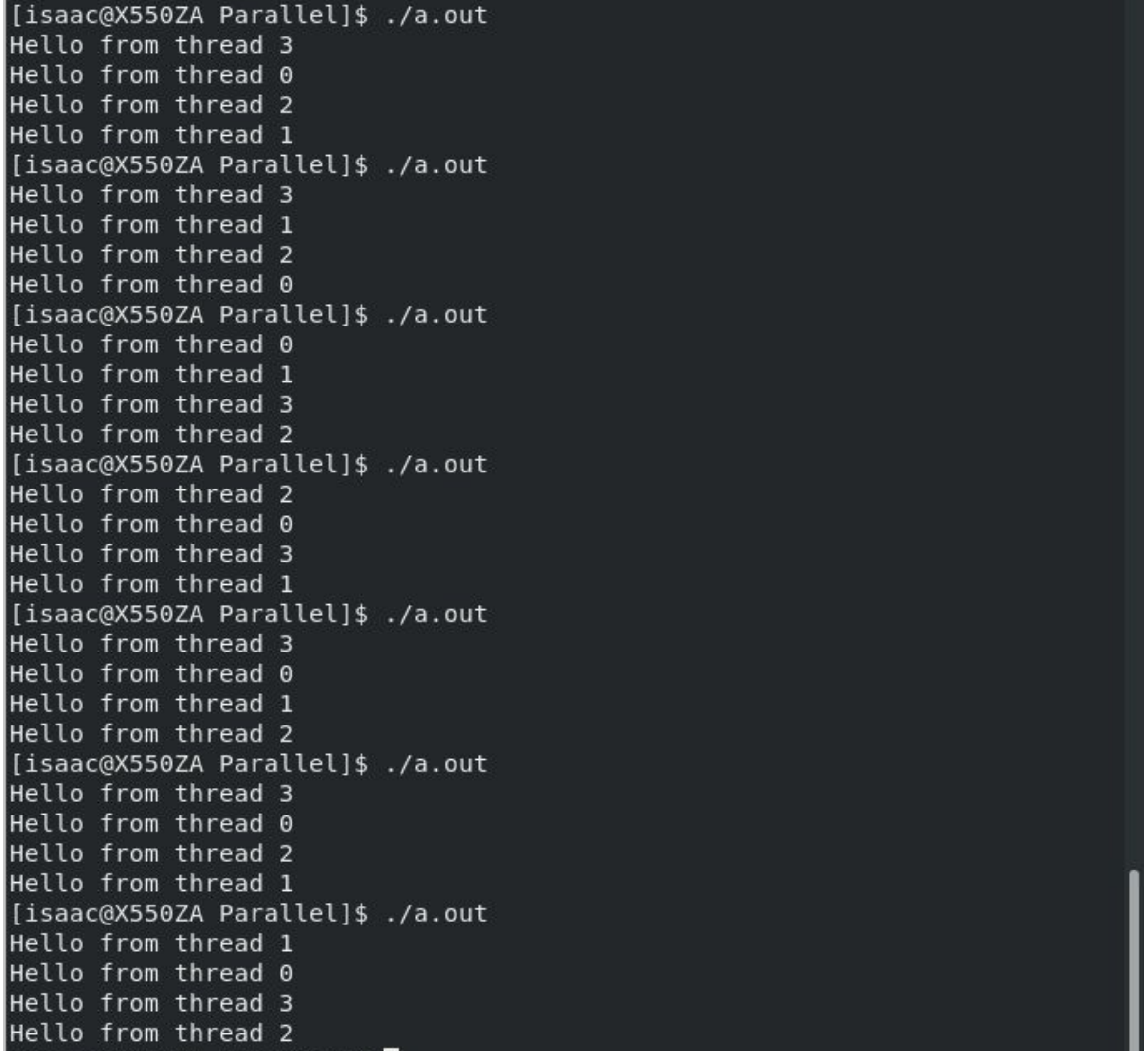


```
#include <stdio.h>
#include <omp.h>

int main()
{
    #pragma omp parallel
    {
      int ID = omp_get_thread_num();
      printf("Hello from thread %d\n", ID);
    }

    return 0;
}
```

Compiling

```
$gcc -fopenmp hello.c
```

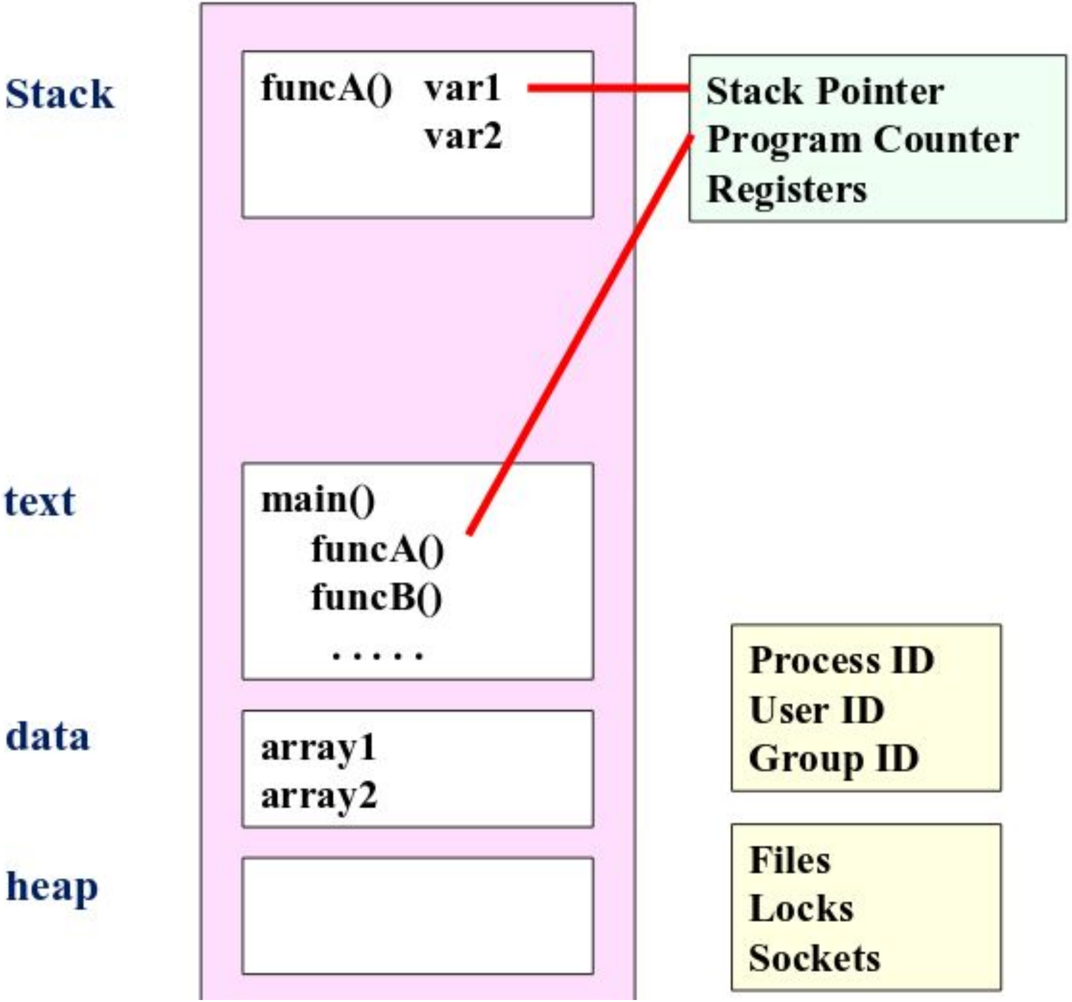
Other compilers- just Google it.



```
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 3
Hello from thread 0
Hello from thread 2
Hello from thread 1
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 3
Hello from thread 1
Hello from thread 2
Hello from thread 0
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 0
Hello from thread 1
Hello from thread 3
Hello from thread 2
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 2
Hello from thread 0
Hello from thread 3
Hello from thread 1
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 3
Hello from thread 0
Hello from thread 1
Hello from thread 2
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 3
Hello from thread 0
Hello from thread 2
Hello from thread 1
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 1
Hello from thread 0
Hello from thread 3
Hello from thread 2
```

Shared Memory Environment

- Symmetric Multiprocessing (SMP)
  - Shared address space with equal time access for each processor
  - Uniform Memory Access (UMA)
  - OS treats every processor the same way
- Non Uniform Memory Architecture (NUMA)
  - “Equal Time”
  - Cache architecture



Stack

funcA() var1
var2

Stack Pointer
Program Counter
Registers

text

main()
funcA()
funcB()
.....

data

array1
array2

heap

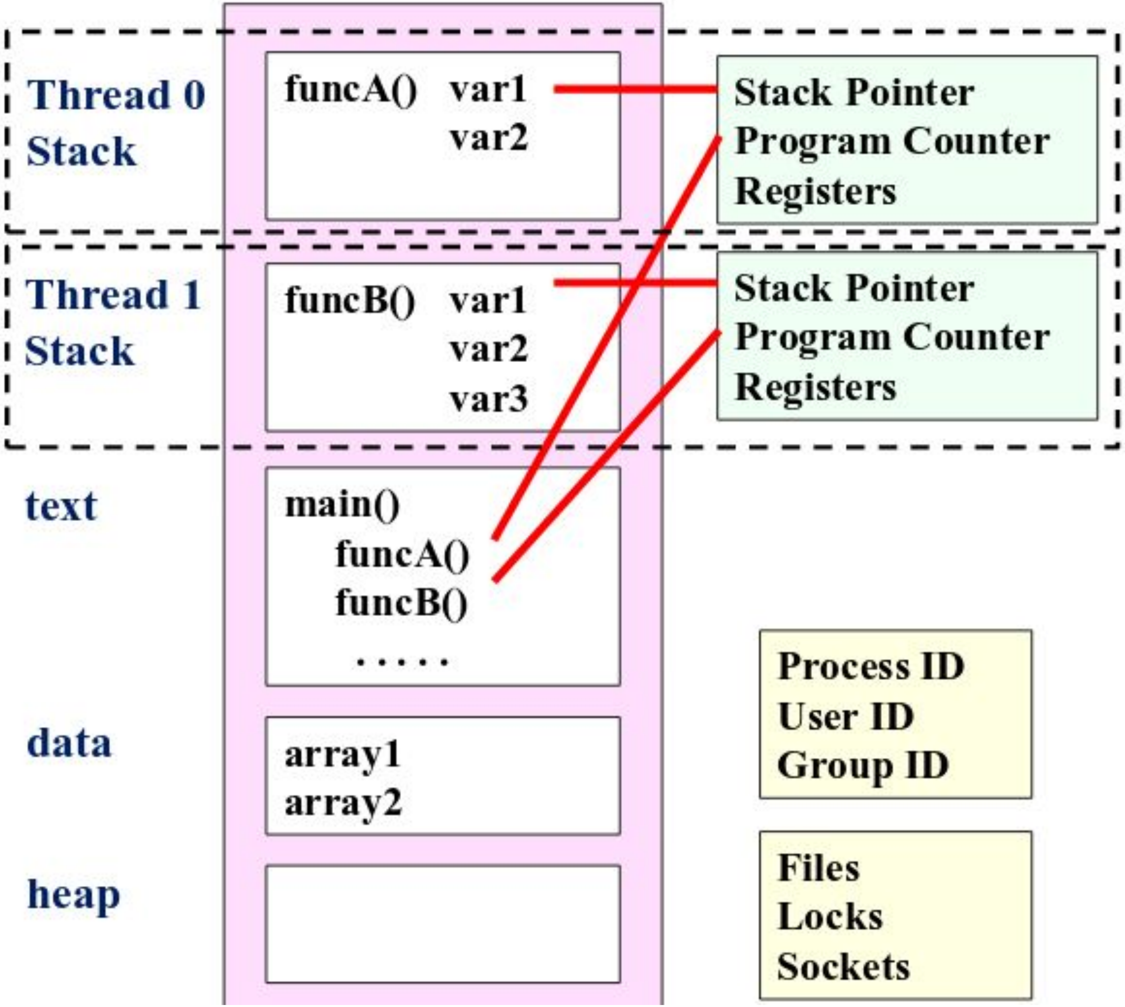
Process ID
User ID
Group ID

Files
Locks
Sockets

Process

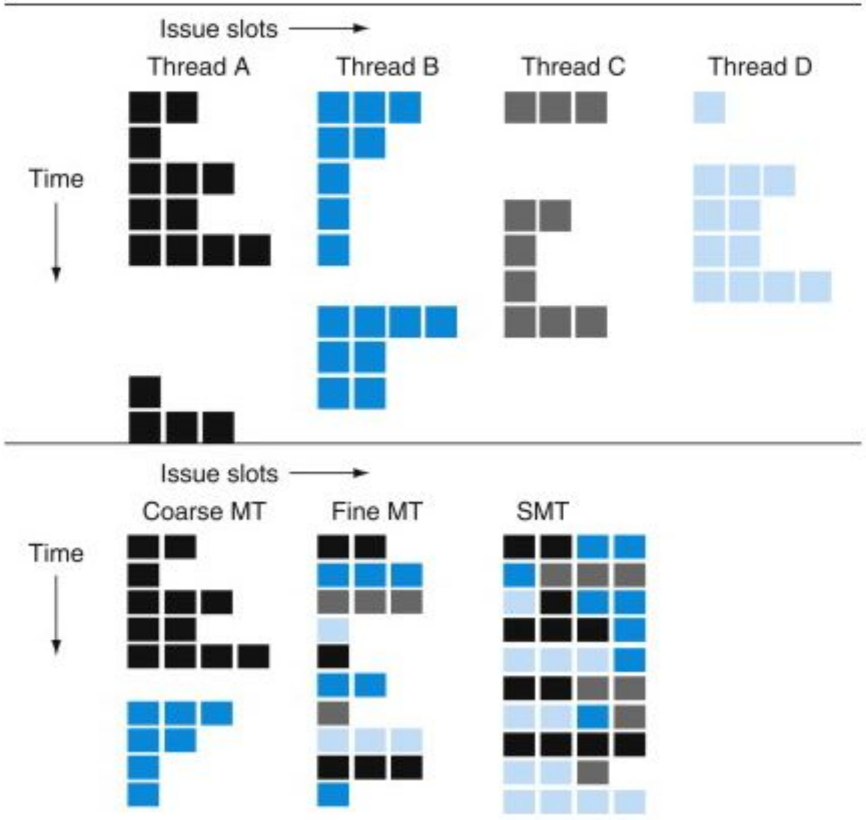
- An instance of a program execution.
- The execution context of a running program … i.e. the resources associated with a program's execution.

- Operating System further decomposes processes into threads by fragmenting the stack
- Each thread has its own piece of the stack, but they share text, data, and heap
- Threads that share this in a process allow for very cheap context switching



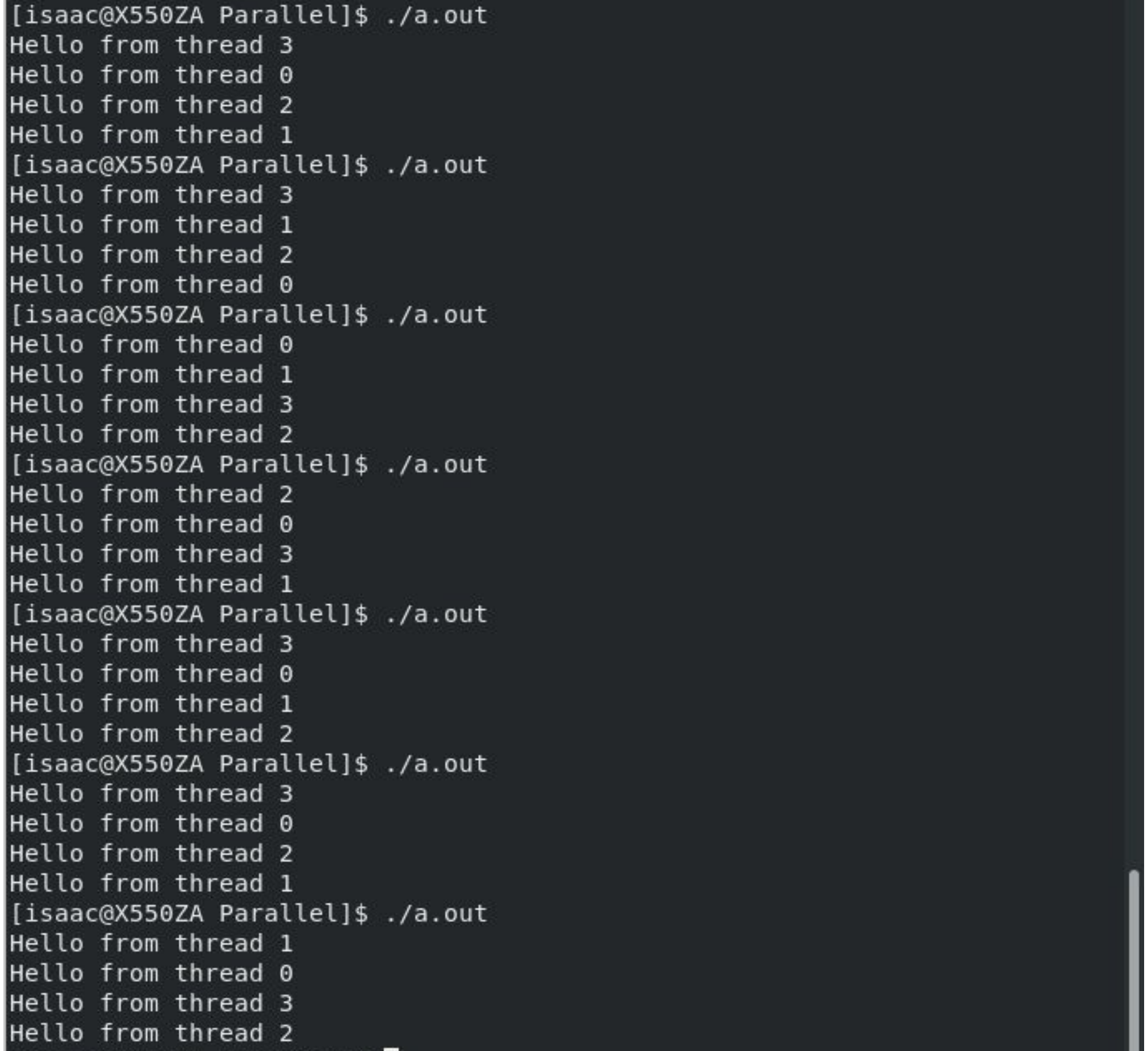
Threads:

- Threads are "light weight processes”
- Threads share Process state among multiple threads … this greatly reduces the cost of switching context.


Issue slots
Thread A
Thread B
Thread C
Thread D
Time
Issue slots
Coarse MT
Fine MT
SMT
Time

Multithreading in OpenMP

- Single process can spawn multiple threads
- Can have more threads than processor cores
- Threads swap in and out, interleaved in time in all possible ways because of dynamic scheduling capability of the OS
- Have to check program for all possible thread interleaves!



```
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 3
Hello from thread 0
Hello from thread 2
Hello from thread 1
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 3
Hello from thread 1
Hello from thread 2
Hello from thread 0
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 0
Hello from thread 1
Hello from thread 3
Hello from thread 2
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 2
Hello from thread 0
Hello from thread 3
Hello from thread 1
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 3
Hello from thread 0
Hello from thread 1
Hello from thread 2
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 3
Hello from thread 0
Hello from thread 2
Hello from thread 1
[isaac@X550ZA Parallel]$ ./a.out
Hello from thread 1
Hello from thread 0
Hello from thread 3
Hello from thread 2
```

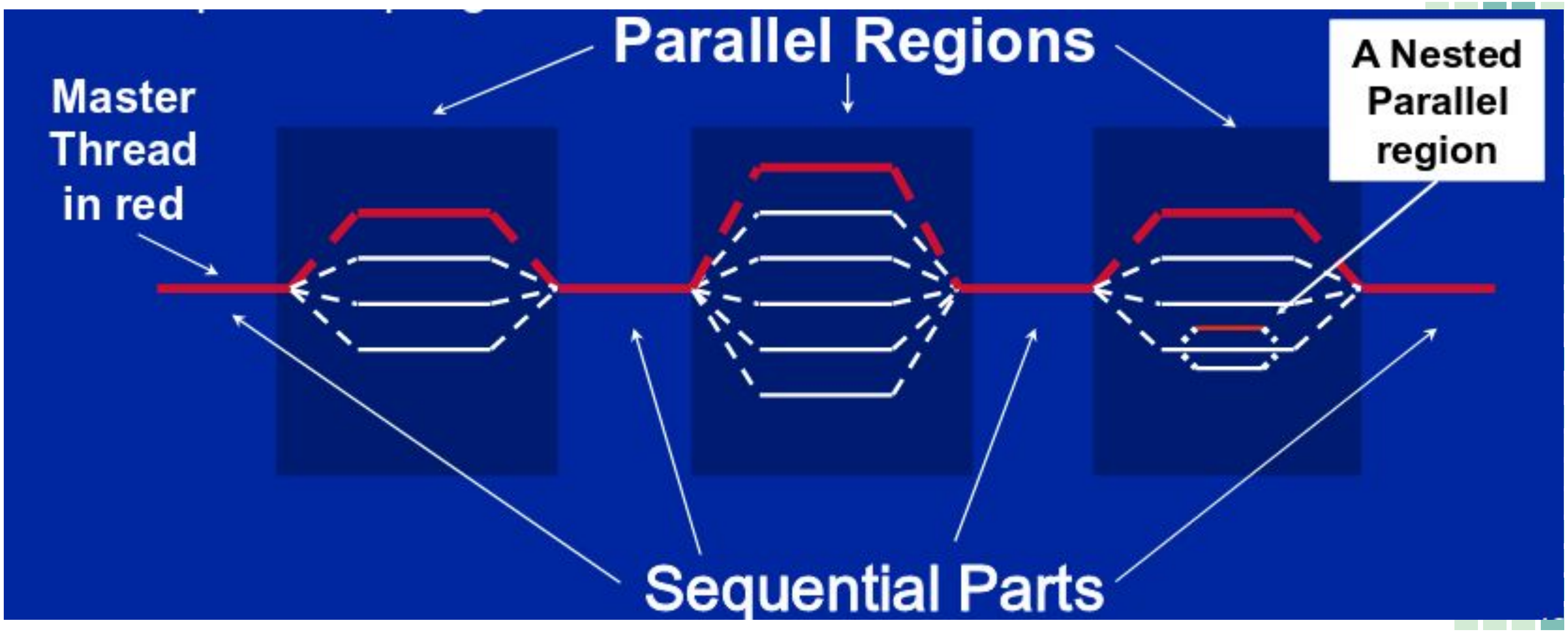
Race Conditions

- Running a program and getting a different answer each time
- Caused by threads sharing data in the heap in a bad way
- Control access to shared variables by **synchronization**

Synchronization

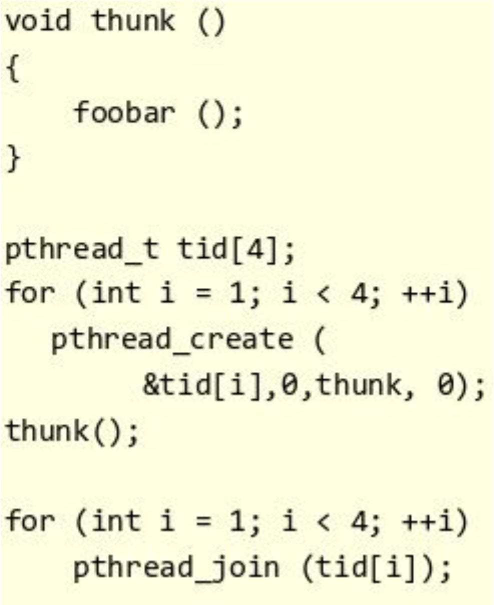
- Protect and order access to shared variables from different threads
- Expensive!
- Synchronise as little as possible by correctly managing the data environment

Fork-Join Parallelism with `#pragma omp parallel`

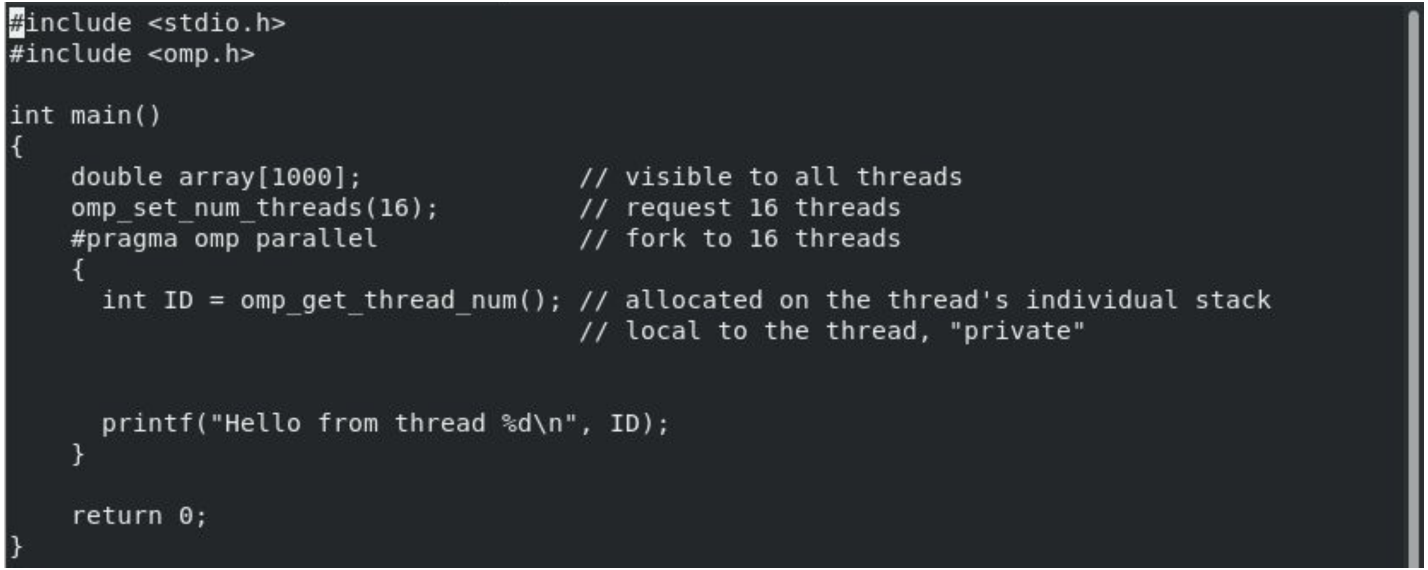


What the compiler does

```
#pragma omp parallel num_threads(4)
{
    foobar();
}
```



```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
         &tid[i],0,thunk, 0);
thunk();

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```



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

int main()
{
    double array[1000];              // visible to all threads
    omp_set_num_threads(16);         // request 16 threads
    #pragma omp parallel             // fork to 16 threads
    {
      int ID = omp_get_thread_num(); // allocated on the thread's individual stack
                                     // local to the thread, "private"


      printf("Hello from thread %d\n", ID);
    }

    return 0;
}
```

Requesting threads / data management

Sequential code

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

OpenMP parallel region

```
#pragma omp parallel
{
            int id, i, Nthrds, istart, iend;
            id = omp_get_thread_num();
            Nthrds = omp_get_num_threads();
            istart = id * N / Nthrds;
            iend = (id+1) * N / Nthrds;
            if (id == Nthrds-1)iend = N;
            for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel
#pragma omp for
            for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

Synchronization

Bringing one or more threads to a well defined and known point in their execution (protect data conflicts, reduce race conditions)

- Barrier: each thread waits at the barrier until all threads arrive
- Mutual exclusion: define a block of code that only one thread at a time can execute

Levels of Synchronization

- High level
  - Critical
  - Atomic
  - Barrier
  - Ordered
- Low level
  - Flush
  - Locks

- Barrier - each thread waits until all the others arrive
  - `#pragma omp barrier`
- Critical - specifies a region that only one thread at a time can enter
  - `#pragma omp critical`
- Atomic - provides mutual exclusion when updating a memory location
  - `#pragma omp atomic`

Additional Resources

- **Excellent book and course on High Performance Computing available entirely for free:** https://bitbucket.org/VictorEijkhout/hpc-book-and-course/
- **OpenMP documentation:** https://www.openmp.org/
- **OpenMP tutorial:** https://computing.llnl.gov/tutorials/openMP/
- **StackOverflow discussion on concurrency vs parallelism:** https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelismhttps://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism