Advanced Architectures

General Outlook

Fundamental performance parameters

$$T = CPI \times IC \times \tau$$

$$CPI = CPI^{ideal} + CPI^{stall}$$

Technological limitations

IC grows with software complexity

τ reaching physical limit $\approx 10^{-10}$ seconds $= \left(10 \text{ GHz}\right)^{-1}$

$CPI^{ideal} = 1$ for integer pipeline

$$CPI^{stall} = CPI^{stall}_{data\,dependency} + CPI^{stall}_{cache\,miss} + CPI^{stall}_{branch\,penalty}$$

Areas for possible improvement

Instruction and thread level parallelism to achieve $CPI^{ideal} < 1$

Reducing instruction dependency stalls to lower $CPI^{stall}_{data\,dependency}$

Reducing cache latency to lower $CPI^{stall}_{cache\,miss}$

Reducing branch stalls to lower $CPI^{stall}_{branch\,penalty}$

Practical Successes

Instruction Level Parallelism (ILP)

Provide multiple copies of hardware units in each processor

Begin executing multiple instructions on same clock cycle

Multiple instructions finish on every clock cycle $\Rightarrow$ CPI < 1

Reducing instruction dependency stalls

Compiler rescheduling or dynamic rescheduling (out-of-order execution)

Improving floating point performance

Parallel process FP instructions

Reducing branch stalls

Advanced branch prediction

Reducing cache latency

Processor pre-fetches cache blocks based on address prediction

Optimization of data structures

Thread Level Parallelism

Provide multiple complete processor cores

Divide code into independently executing threads

Types of Parallelism

Pipelining

Instruction I_{n+1} begins before In completes

1 instruction completes on every clock cycle τ

R = 1 / τ instructions complete every second

Superscalar

M > 1 copies of pipeline in parallel

Execute M instructions start on same clock cycle

M instructions complete on every clock cycle

Superpipelining

Divide pipeline into smaller stages — less work per stage

Less work $\Rightarrow$ shorter clock cycle $\tau' < \tau \Rightarrow$ higher clock rate R' > R

R' > R instructions complete every second

Multiprocessor

N > 1 program sections running on N processors

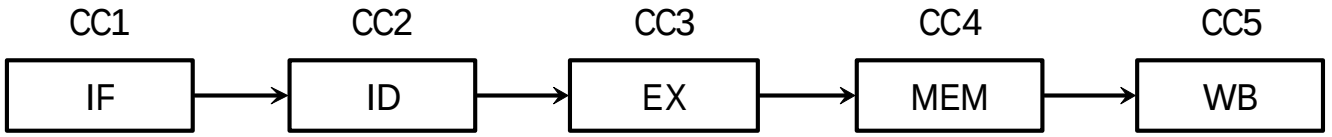
Overall program runs in less time

DLX Pipeline

Five pipeline stages

New instruction begins on each clock cycle

One instruction completes on each clock cycle



	1	2	3	4	5	6	7	8
I_1	IF	ID	EX	MEM	WB			
I_2		IF	ID	EX	MEM	WB		
I_3			IF	ID	EX	MEM	WB	

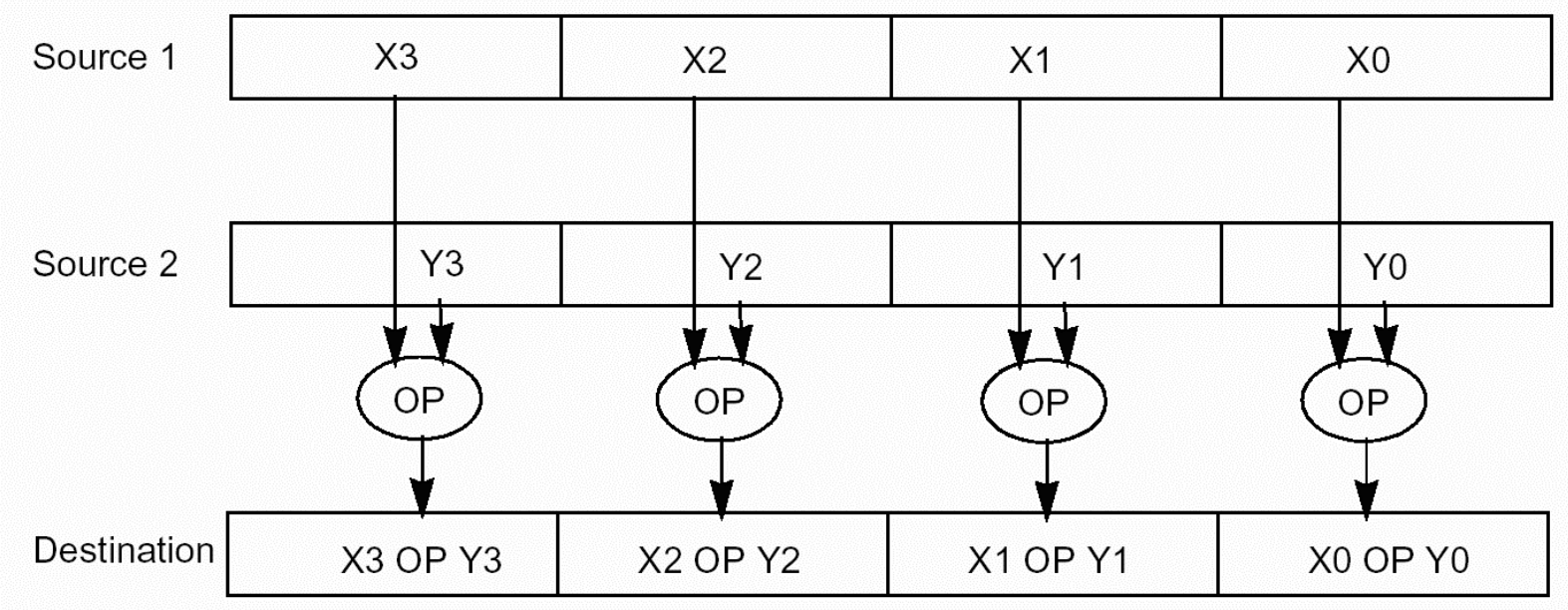
$$CPI^{ideal} = \frac{\text{cycles}}{\text{instruction}} = \frac{IC+4}{IC} \xrightarrow[IC \to \text{large}]{} 1$$

$$\text{Run-Time}^{\text{ideal}}_{\text{pipeline}} = CPI^{ideal} \times IC \times \tau \xrightarrow[IC \to \text{large}]{} IC \times \tau$$

Vector Processors

SIMD execution model

Single Instruction performed in parallel on Multiple Data



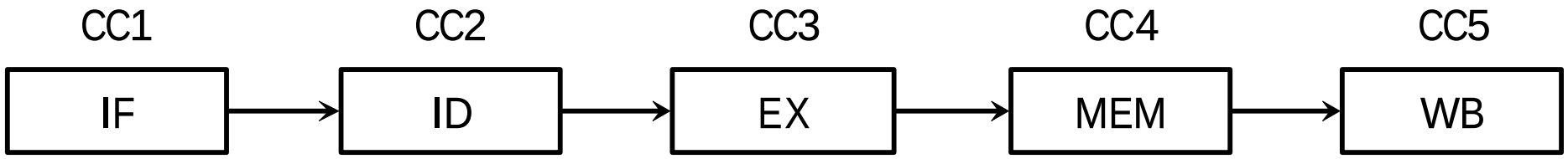
Typical applications for vector operations

Data compression/decompression

Audio processing

Graphics processing

DLX Vector Pipeline Example



p_LW P1, 400(R1)	Load 4 memory words (16 bytes) to register P1
p_LW P2, 800(R1)	Load 4 memory words (16 bytes) to register P2
p_ADD P3, P1, P2	Perform 4 word additions in parallel

	1	2	3	4	5	6	7	8
p_LW P1, 400(R1)	**IF**	**ID**	**EX**	**MEM**	**WB**			
p_LW P2, 800(R1)		**IF**	**ID**	**EX**	**MEM**	**WB**		
p_ADD P3, P1, P2			**IF**	**ID**	**ID**	**EX**	**MEM**	**WB**

Examples of Vector Processing

Intel MMX Technology

Integer operations on 64-bit registers

8 bytes, 4 words, or 2 dwords

Intel SSE (SSE2/SSE3/AVX) Technology

Similar to MMX for Floating Point operations

	SSE	SSE2/SSE3/AVX
Register width	128 bits = 16 bytes	256 bits = 32 bytes
single-precision FP operations	4	8
double-precision FP operations	2	4

PowerPC AltiVec vector processor

Similar to SSE

128-bit registers

Compiler Support

Vector instructions part of processor instruction set

Reasonable compiler supports vectorization

Intel Vectorization Example

```
int i;
float a[100], red;
…
red = 0;
for (i = 0; i < 100; i++) {
     red += a[i];
}
```

Loop runs **100** times

Operates on one array element in each loop iteration

```
int i;
float a[100], red;
…
red = 0;
ASM p_XOR xmm0,xmm0              /* zero 128-bit accumulator */
for (i = 0; i < 25; i++) {
     ASM p_ADD xmm0,a[4*i]       /* four 32-bit additions */
}
ASM h_ADD xmm0                   /* add four 32-bit FP registers to one */
ASM p_MOV red, xmm0              /* move FP sum to memory location */
```

Loop runs **25** times

Operates on 4 array elements in each loop iteration

Cache Refresh Latency

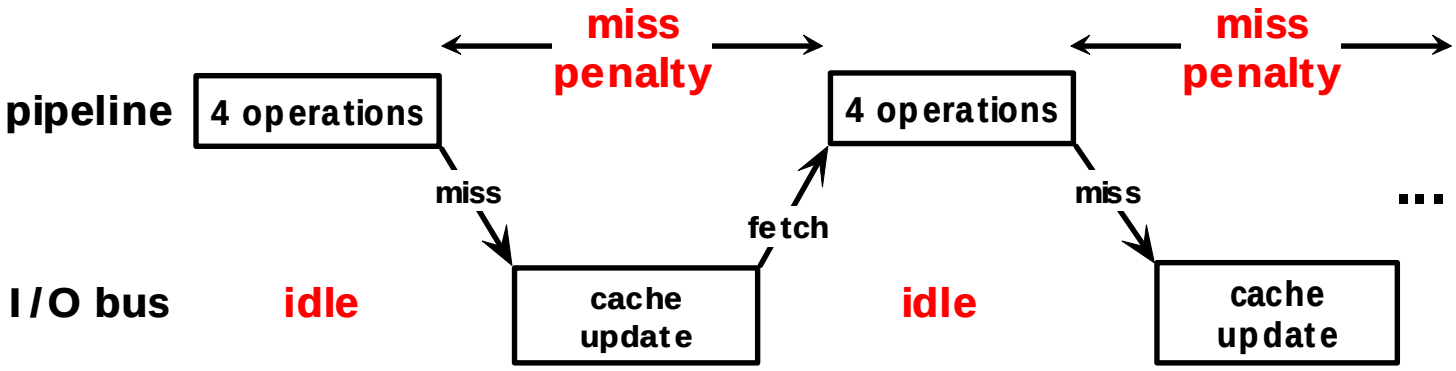
```
for (i = 0; i < 1024; i++) {
     for (j = 0; j < 1024; j++) {
          SSE_operation a[i][j];
     }
}
```

Reads 1024 × 1024 SSE operands (16-byte operands) = 16 MB

Reads sequentially, without repeated access to same data

Pentium 4 has 64 byte block size = 4 x 16-byte operand

Will miss in L1 on every 4 accesses



Cache Prefetch

```
for (i = 0; i < 1024; i++) {
     for (j = 0; j < 1024; j++) {
          prefetch a[i][j+8];
          SSE_operation a[i][j];
     }
}
```

Reads 1024 × 1024 SSE operands (16-byte values) = 16 MB

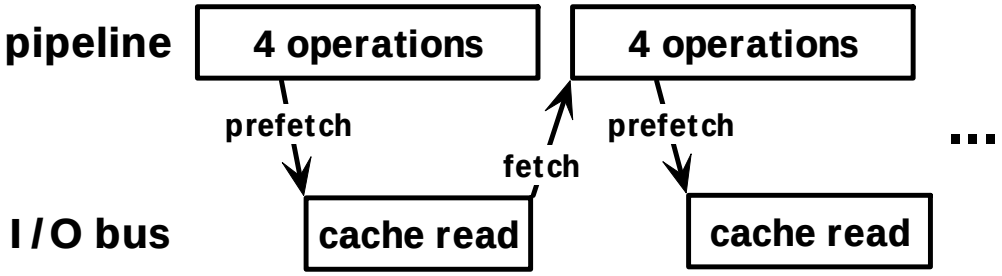
On Pentium 4 prefetching,

Software prefetch loads 128 bytes (2 cache blocks)

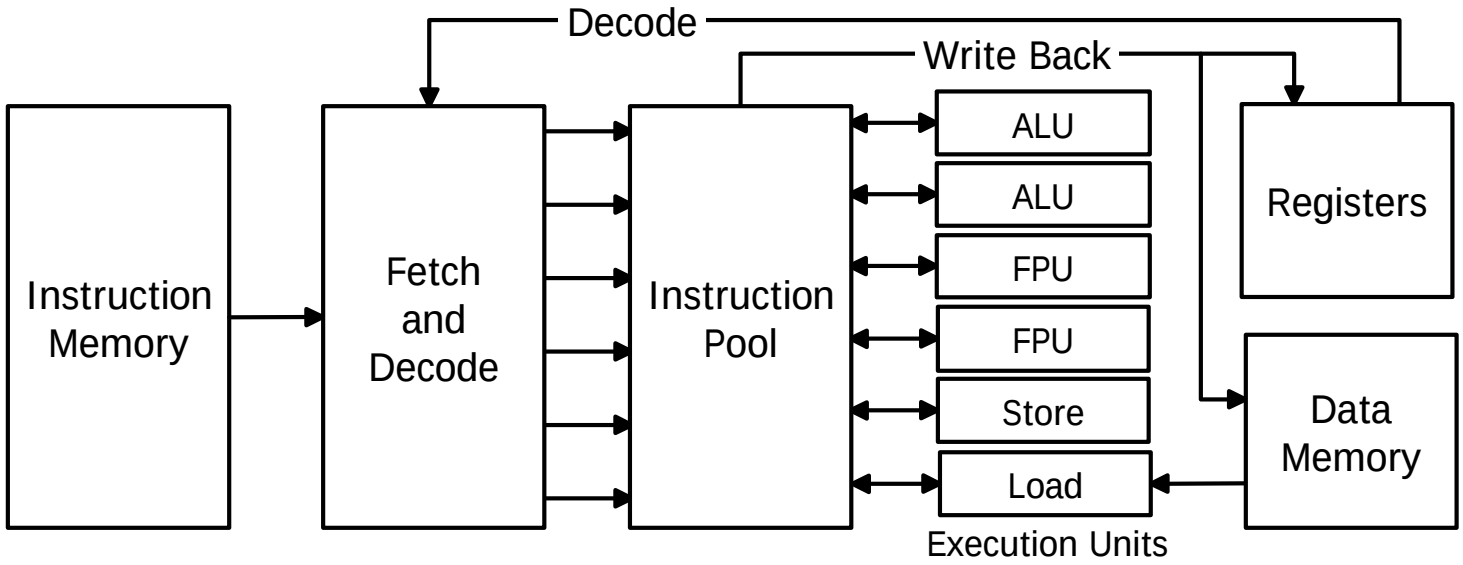
128 bytes = (128/16) = 8 SSE operands

Prefetch 8 operands forward

NOP on prefetch of cache hit



P6 Superscalar with Dynamic Rescheduling



Intel P6 architecture for Intel x86 since Pentium II

Fetch / Decode

Convert IA-32 instructions to 1 – 6 RISC-type **micro-ops** per CC

Instruction pool — out-of-order dynamic rescheduling

Holds micro-ops until ready for execution

Scheduler issues micro-ops to parallel execution in ALU, FPU, Load, Store

Finished micro-ops return to instruction pool with execution results

Retirement (in-order write back)

Finished micro-ops write in original program order

Instruction Scoreboard

Status field assigned to instructions in program listing

NR	Not Ready	At least one source operand not available
R	Ready	All source operands available
X	Executed	Instruction executed — destination operand not available
F	Finished	Instruction executed — all destination operand(s) available

Instructions executed according to

- Status fields
  - Only instructions marked Ready can be executed
- Scheduling policy
  - Depends on hardware organization

Update scoreboard after each clock cycle

- Completed instructions marked Finished
- Instructions marked Ready as operands become available

Scoreboard Example for DLX

Instruction status after EX clock cycle (ignoring IF, ID, MEM, WB)

Instruction	1	2	3	4	5	6	7	8	9	10	11	12
LW R2,[X]	X	F	F	F	F	F	F	F	F	F	F	F
ADD R2,R2,#123	NR	R	F	F	F	F	F	F	F	F	F	F
SW [X],R2	NR	NR	R	F	F	F	F	F	F	F	F	F
LW R3,[Y]	R	R	R	R	X	F	F	F	F	F	F	F
SUB R3,R3,#456	NR	NR	NR	NR	NR	R	F	F	F	F	F	F
SW [Y],R3	NR	NR	NR	NR	NR	NR	R	F	F	F	F	F
LW R4,[Z]	R	R	R	R	R	R	R	R	X	F	F	F
SUB R4,R4,#789	NR	NR	NR	NR	NR	NR	NR	NR	NR	R	F	F
SW [Z],R4	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	R	F

Scheduling rules

Issue only ready instructions

Choose instructions in **ORIGINAL PROGRAM ORDER**

Scoreboard generates inefficient code

NR	Not Ready
R	Ready
X	Executed
F	Executed and destination operand available

Scoreboard Example for Dynamic DLX

Instruction status after EX clock cycle (ignoring IF, ID, MEM, WB)

Instruction	1	2	3	4	5	6	7	8	9
LW R2,[X]	X	F	F	F	F	F	F	F	F
ADD R2,R2,#123	NR	R	R	F	F	F	F	F	F
SW [X],R2	NR	NR	NR	R	R	R	F	F	F
LW R3,[Y]	R	X	F	F	F	F	F	F	F
SUB R3,R3,#456	NR	NR	R	R	F	F	F	F	F
SW [Y],R3	NR	NR	NR	NR	R	R	R	F	F
LW R4,[Z]	R	R	X	F	F	F	F	F	F
SUB R4,R4,#789	NR	NR	R	R	R	F	F	F	F
SW [Z],R4	NR	NR	NR	NR	NR	R	R	R	F

Scheduling rules

Issue only ready instructions

Choose instructions in **ORDER THEY BECOME READY**

Scoreboard generates

compiler-rescheduled code

NR	Not Ready
R	Ready
X	Executed
F	Executed and destination operand available

Scoreboard Example for P6

Execution condition after each clock cycle (ignoring fetch, decode, write-back)

Instruction	Execution Unit	CC1	CC2	CC3	CC4	CC5	CC6
`LW R2,[X]`	Load	F	F	F	F	F	
`ADD R2,R2,#123`	ALU	R	F	F	F	F	
`SW [X],R2`	Store	NR	R	F	F	F	
`LW R3,[Y]`	Load	R	F	F	F	F	
`SUB R3,R3,#456`	ALU	NR	R	F	F	F	
`SW [Y],R3`	Store	NR	NR	R	F	F	
`LW R4,[Z]`	Load	R	R	F	F	F	
`SUB R4,R4,#789`	ALU	NR	NR	R	F	F	
`SW [Z],R4`	Store	NR	NR	NR	R	F	

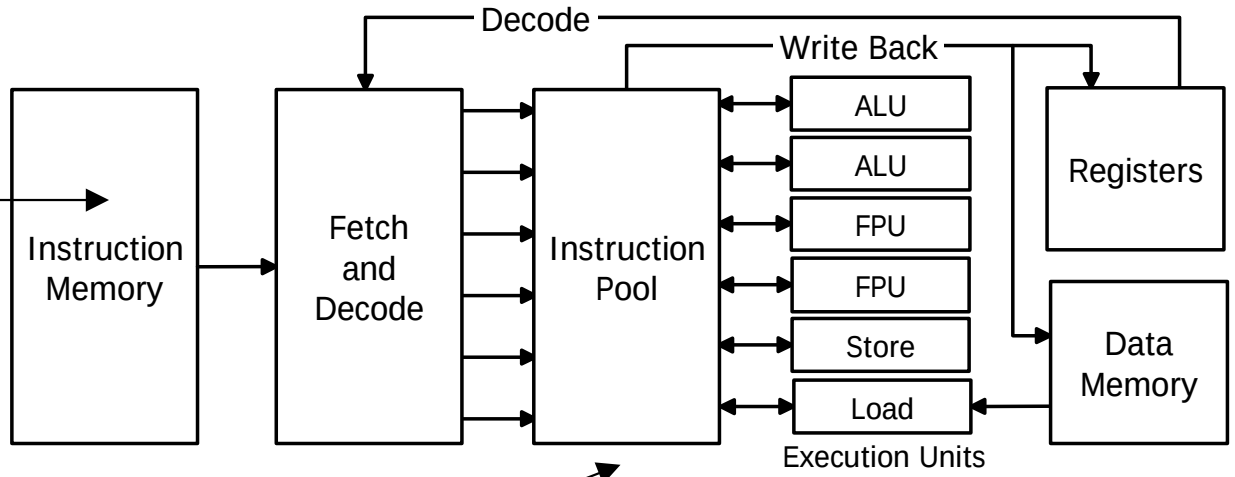
Scheduling rules

Issued only ready instructions
Among ready instructions, maintain program list order
Only 1 load and 1 store per CC
Up to 2 ALU and 2 FPU instructions per CC

NR	Not Ready
R	Ready
F	Finished

Program Execution in P6

```
ADD [X],123
SUB [Y],567
SUB [Z],789
```



IA-32 instructions decoded in 2 CC to RISC micro-ops with register renaming

Dynamic scheduling

```
LW   R2,[X]
ADD  R2,R2,#123
SW   [X],R2
LW   R3,[Y]
SUB  R3,R3,#567
SW   [Y],R3
LW   R4,[Z]
SUB  R4,R4,#789
SW   [Z],R4
```

`LW R2,[X]`	Load	CC1
`ADD R2,R2,#123` `LW R3,[Y]`	ALU Load	CC2

`SW [X],R2`
`SUB R3,R3,#567`
`LW R4,[Z]`
	Store
	ALU
	Load
		CC3

`SW [Y],R3` `SUB R4,R4,#789`	Store ALU	CC4
`SW [Z],R4`	Store	CC5

Hardware Utilization

Unit	CC1	CC2	CC3	CC4	CC5
ALU	IDLE	ADD R2,R2,#123	SUB R3,R3,#567	SUB R4,R4,#789	IDLE
ALU	IDLE	IDLE	IDLE	IDLE	IDLE
FPU	IDLE	IDLE	IDLE	IDLE	IDLE
FPU	IDLE	IDLE	IDLE	IDLE	IDLE
Load	LW R2,[X]	LW R3,[Y]	LW R4,[Z]	IDLE	IDLE
Store	IDLE	IDLE	SW [X],R2	SW [Y],R3	SW [Y],R4

Good program efficiency

Program executes in minimum number of sequential cycles

Low hardware utilization

Most execution units idle in most clock cycles

Higher ILP ⇒ higher utilization of execution units

Higher utilization ⇒ larger pool of independent instructions

Speculation — deep branch prediction

Many instructions executed before program flow determined

Hardware multithreading

Instructions from different threads are independent

Deep Superpipeline for DLX

Divide each pipeline stage into 2 smaller stages

Each new stage does half the work in half the time

New stage finishes in half the time $\Rightarrow$ Double clock speed

	1	2	3	4	5	6	7	8	9	10	11	12
I_1	IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2		
I_2		IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2	
I_3			IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2

Double clock speed $\Rightarrow$ $\tau_{\text{superpipeline}} = \dfrac{\tau_{\text{pipeline}}}{2}$

$$CPI_{\substack{\text{ideal}\\\text{superpipeline}}} = \frac{IC+9}{IC} \xrightarrow[IC \to \text{large}]{} 1$$

$$T_{\substack{\text{ideal}\\\text{superpipeline}}} = CPI_{\substack{\text{ideal}\\\text{superpipeline}}} \times IC \times \frac{\tau_{\text{pipeline}}}{2} \xrightarrow[IC \to \text{large}]{} IC \times \frac{\tau_{\text{pipeline}}}{2} = \frac{1}{2} T_{\substack{\text{ideal}\\\text{pipeline}}}$$

Problems with deep superpipeline

Some instructions cannot be effectively split

Some operations do not scale in time — faster clock $\Rightarrow$ more stall cycles

Cache update, branch penalty, page fault, etc

Pentium 4 Superpipeline

Pentium III

10 stage pipeline at clock speed up to about 1.5 GHz

Pentium 4

20 stage pipeline at clock speed up to about 4.0 GHz

Expect

1.5 GHz processor ~ **50%** faster than same processor at 1.0 GHz

Measurement on SPEC CINT2000

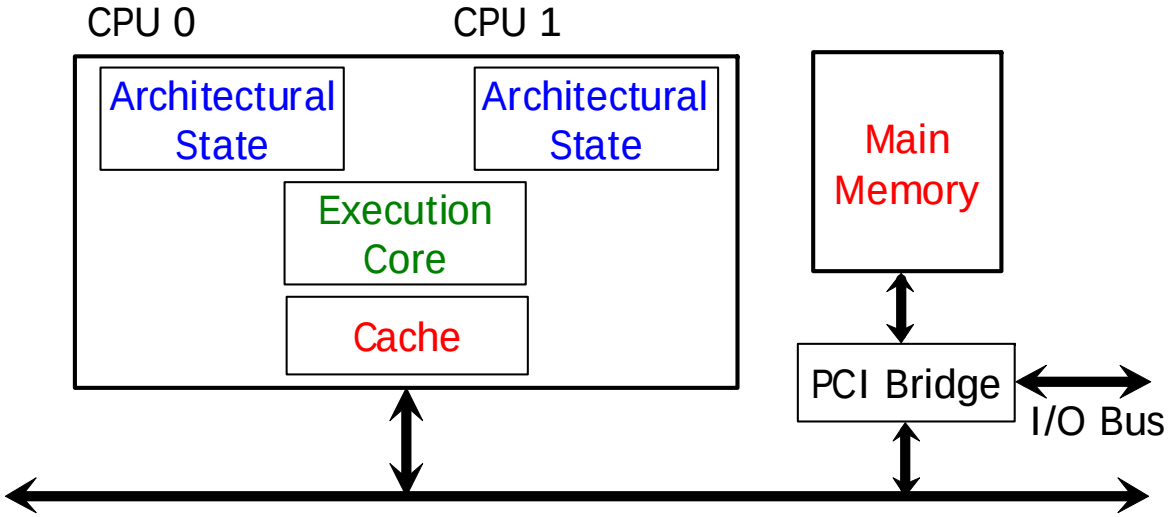
1.5 GHz Pentium-4 ~ **20%** faster than 1.0 GHz Pentium-III

$$S = \frac{CPI^{PIII} \times IC^{PIII} \times \frac{1}{1.0\ \text{GHz}}}{CPI^{P4} \times IC^{P4} \times \frac{1}{1.5\ \text{GHz}}} = \frac{CPI^{PIII}}{CPI^{P4}} \times \frac{1.5}{1.0} = 1.2 \Rightarrow \boxed{CPI^{P4} = 1.25 \times CPI^{PIII}}$$

$$CPI^{P4}_{ideal} + CPI^{P4}_{stall} = 1.25 \times \left(CPI^{PIII}_{ideal} + CPI^{PIII}_{stall} \right)$$

$$CPI^{P4}_{ideal} = CPI^{PIII}_{ideal} = 1 \Rightarrow \boxed{CPI^{P4}_{stall} = 1.25 \times CPI^{PIII}_{stall} + 0.25 \approx 0.5}$$

Hyper-Threading



Architectural State

Registers, stack pointers and program counter

Execution Core

ALU, FPU, vector processors, memory unit

Two copies of architectural state — one execution core

OS sees two sets of registers — looks like two CPUs

OS assigns threads to CPU 0 and CPU 1

CPU 0 and CPU 1 issue instructions to shared execution core

No stall in either thread

CPU 0 and CPU 1 issue instructions on alternate clock cycles

Stall in one thread

Other CPU issues instructions on each clock cycle until stall ends

Both CPUs keep working on most clock cycles

Expected Improvement from Hyper-Threading

Without hyper-threading

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stall}} \times P(\text{stall}) = CPS \times P_S$$

$CPS =$ cycles per stall

With hyper-threading

$$CPI^{stall}{}' = \frac{\text{stall cycles}}{\text{stall}} \times P(\text{2 simultaneous stalls}) \approx CPS \times \left[P_S\right]^2$$

Speedup

$$S = \frac{CPI \times IC \times \tau}{CPI^{HT} \times IC^{HT} \times \tau^{HT}} = \frac{CPI}{CPI^{HT}} = \frac{1 + CPS \times P_S}{1 + CPS \times \left[P_S\right]^2}$$

Take for Pentium-4

$$CPI^{stall} \approx 0.5$$

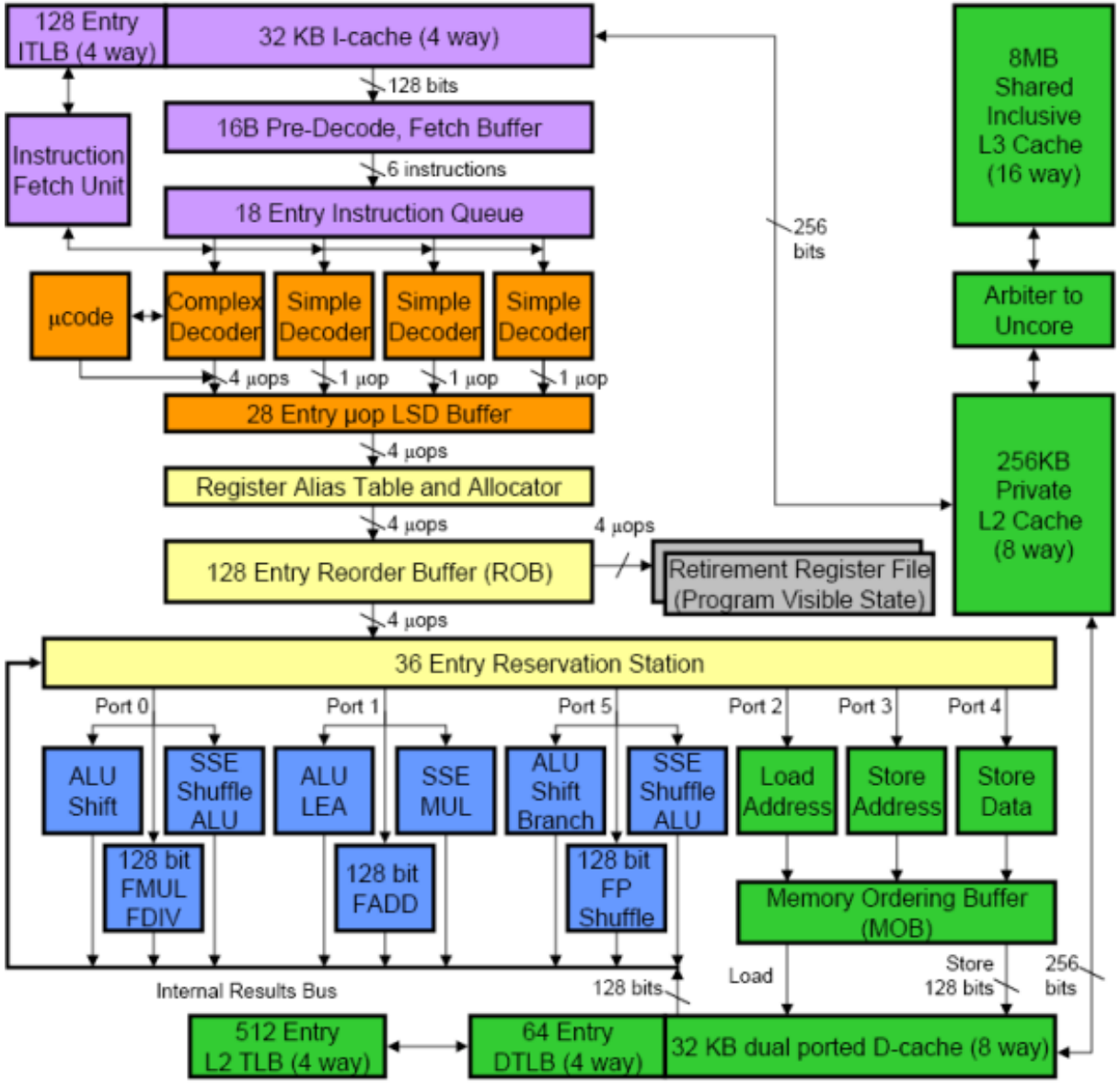
$$CPS = 2 \text{ cycles per stall} \Rightarrow P_S = 0.5 / 2 = 0.25$$

$$S = \frac{1 + 2 \times 0.25}{1 + 2 \times \left[0.25\right]^2} = 1.2$$

Measured improvement $= 1.2$

Intel, "Hyper-Threading Technology Architecture and Microarchitecture"

Intel Nehalem Micro-Architecture



David Kanter, "Inside Nehalem: Intel's Future Processor and System", http://realworldtech.com/includes/templates/articles.cfm?ArticleID=RWT040208182719&mode=print

Amdahl's Equation in Parallel Processing

$$CPI^{N-processors} = \frac{CPI^{N=1}}{N} \times P(\text{work can be parallelized})$$

$$+CPI^{N=1} \times P(\text{work cannot be parallelized})$$

$$= \frac{CPI^{N=1}}{N} \times F_P + CPI^{N=1} \times (1 - F_P)$$

$F_P =$ Fraction of processing that can be performed independently

$N =$ Number of processing units

$$S = \frac{CPI^{N=1} \times IC \times \tau}{CPI^{N-processors} \times IC \times \tau} = \frac{CPI^{N=1}}{CPI^{N-processors}} = \frac{1}{(1 - F_P) + \frac{F_P}{N}}$$

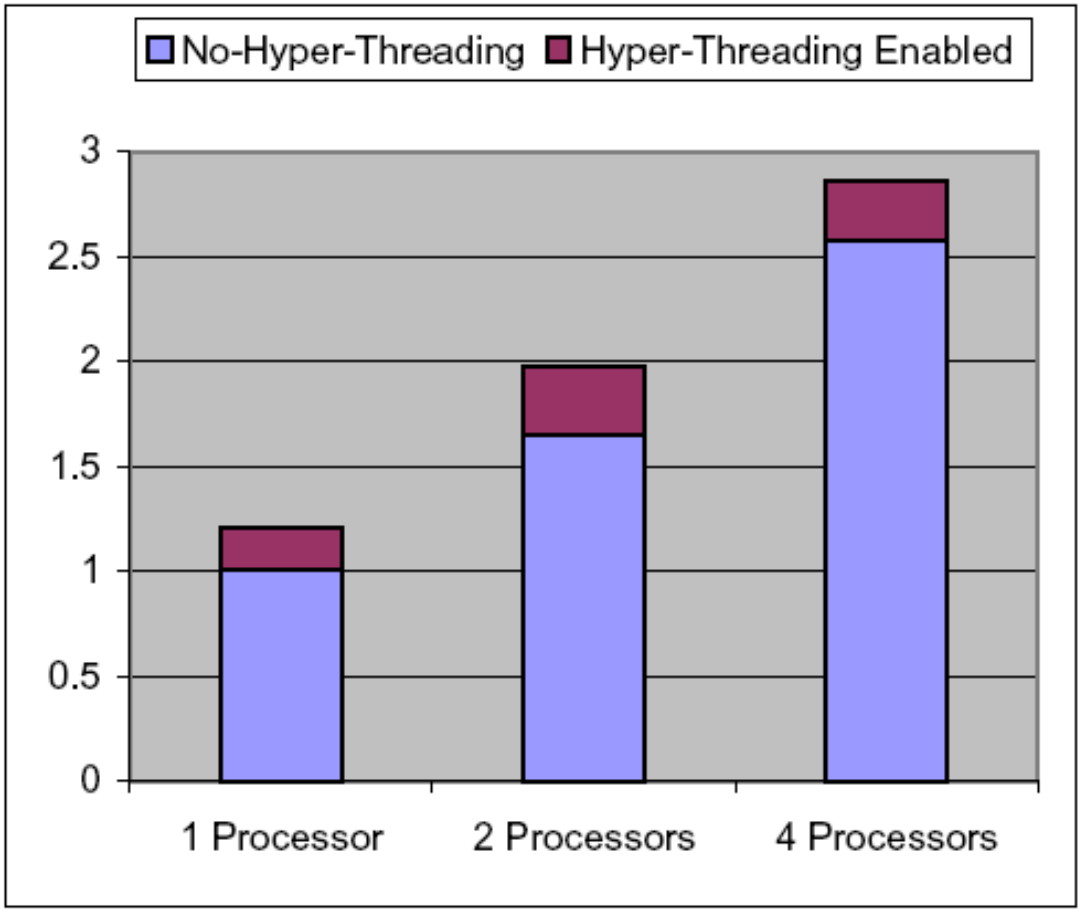
MP and HT Performance Enhancements

MP Without Hyper Threading

CPUs	S	S/CPU
2	1.7	0.85
4	2.6	0.65

Hyper Threading Without MP

CPUs	S	S/CPU
2	1.2	0.60

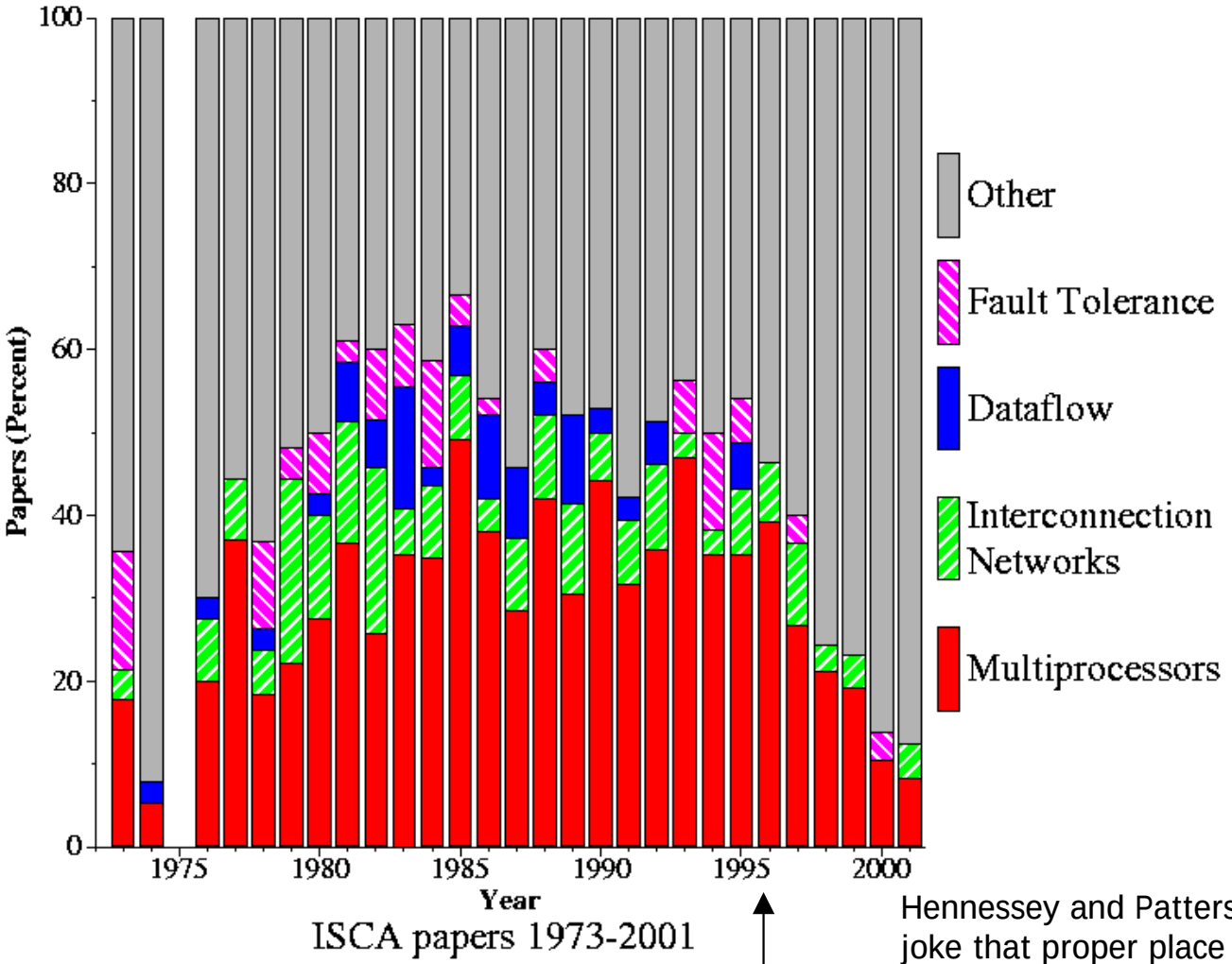


Speed-up (S) for On Line Transaction Processing (OLTP) Workload

Rise and Fall of Multiprocessor R&D

Topics of papers submitted to ISCA 1973 to 2001

Sorted as percent of total



ISCA papers 1973-2001

ISCA — International Symposium on Computer Architecture

Hennessey and Patterson joke that proper place for multiprocessing in their book is Chapter 11 (a section of US business law on bankruptcy)

Ref: Mark D. Hill and Ravi Rajwar, "The Rise and Fall of Multiprocessor Papers in the International Symposium on Computer Architecture (ISCA)", http://pages.cs.wisc.edu/~markhill/mp2001.html

Basic Interprocess Communication Models

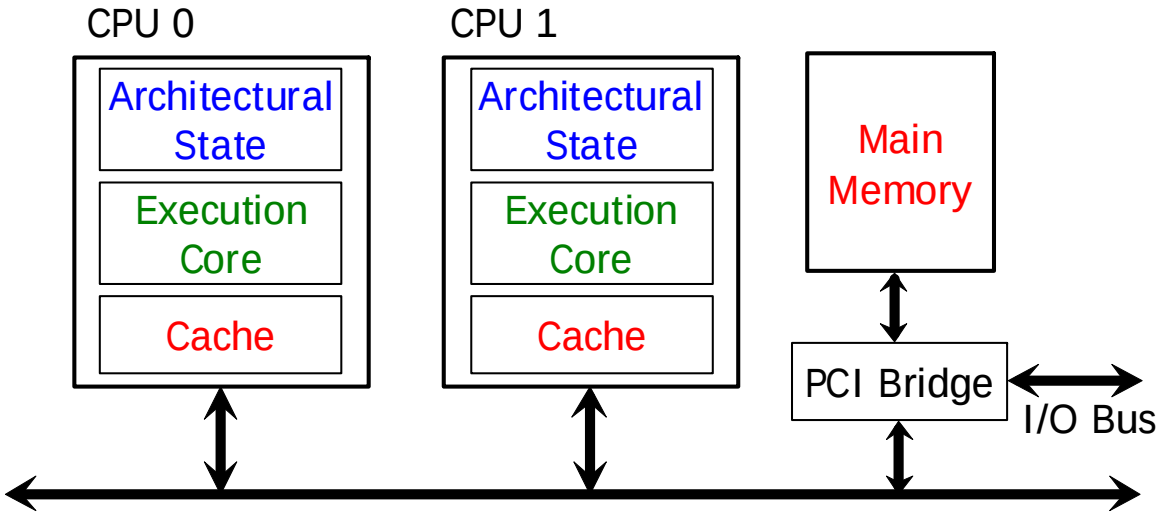
Shared memory system

- Interprocess communication — write/read shared memory location
  - Single shared address space
- Sequential coherence enforced by cache snooping
  - Bus imposes write / read order
  - Cache coherency overhead

Message passing system

- Interprocess communication — send/receive structured messages
  - Send / request data
  - Provide requested data or status
- Sequential coherence enforced by message content + synchronization
  - No snooping or snooping overhead
  - Message management contributes overhead

Multiprocessor Shared Memory Multi-Threading



One or more physical microprocessors

Architectural state

Registers, including stack pointers and program counter

Execution core

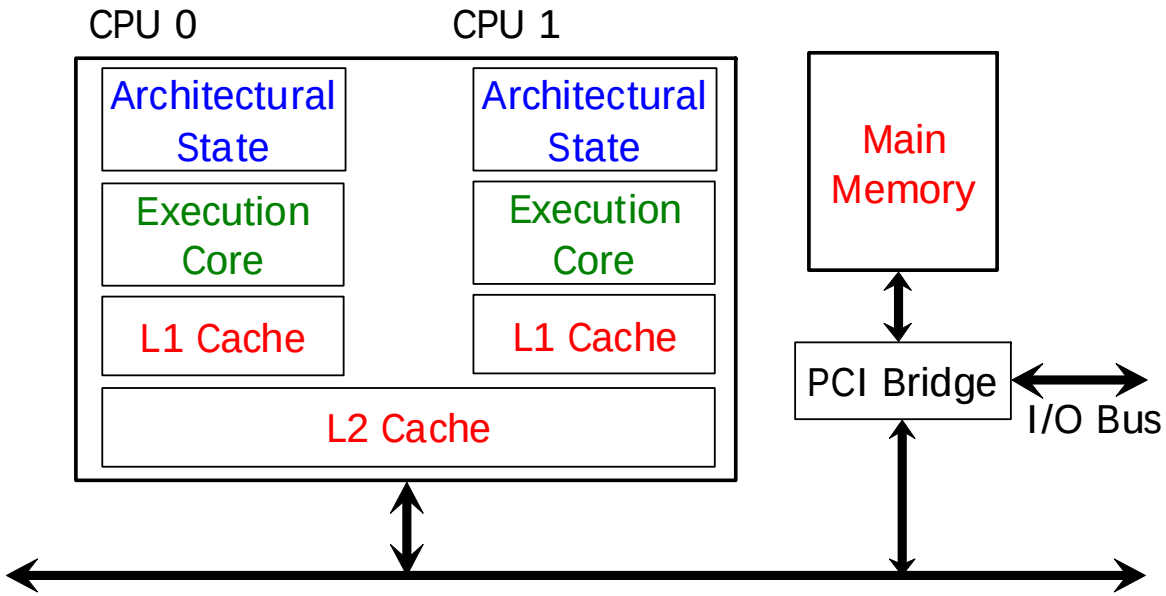
Integer ALUs, FPUs, vector processors, memory access

OS assigns a thread to each processor

Each thread runs independently

On long stall (page fault) a CPU can switch threads

Multi-Core Shared Memory Multi-Threading



Multiprocessor system on one physical chip

Cheaper than multi-microprocessor system

Can be bottleneck at memory bus

- Both processors need to update cache simultaneously
- One processor must wait

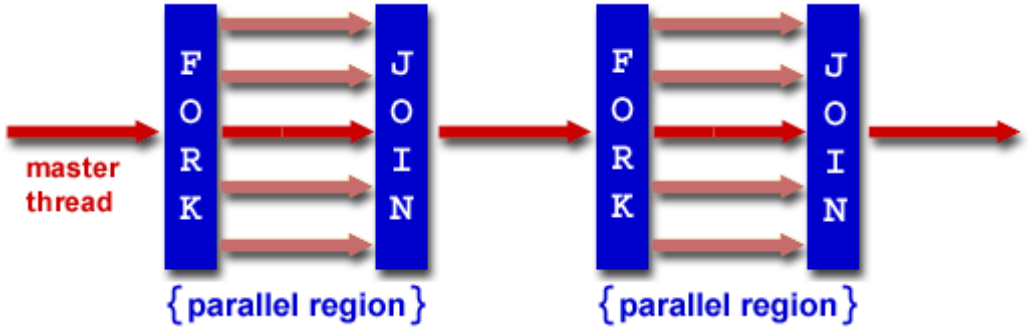
Open MP for Shared Memory Systems

Application Program Interface (API) for multiprocessing

- Supports shared memory applications in C/C++ and Fortran
- Provides directives for explicit thread-based parallelization
- Simple programming models on shared memory machines

Fork — Join Model

- Master thread (consumer thread)
  - Programs initiate as single thread
  - Executes sequentially until parallel construct is encountered
- Fork (producer thread)
  - Master thread creates team of parallel threads
  - Program statements in parallel construct execute in parallel
- Join
  - Team threads complete
  - Synchronize and terminate
  - Master thread continues
- Nesting
  - Forks can be defined within parallel sections



Ref: https://computing.llnl.gov/tutorials/openMP/

General Code Structure

```
#include <omp.h>
main ()  {
int var1, var2, var3;
/* Serial code */
     ...
#pragma omp parallel private(var1, var2) shared(var3)
   {
  /* Parallel section executed by all threads */
     ...
  /* All threads join master thread and disband */
   }
/* Resume serial code */
     ...
}
```

Variables private to each thread
Each thread has private copy

Variables shared among all threads
One copy accessed by all threads

"Hello Worlds" Program

```
#include <omp.h>
main ()  {
int nthreads, tid;
/* Fork team of threads with private variables */
#pragma omp parallel private(tid)
  {
  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);
  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
}
```

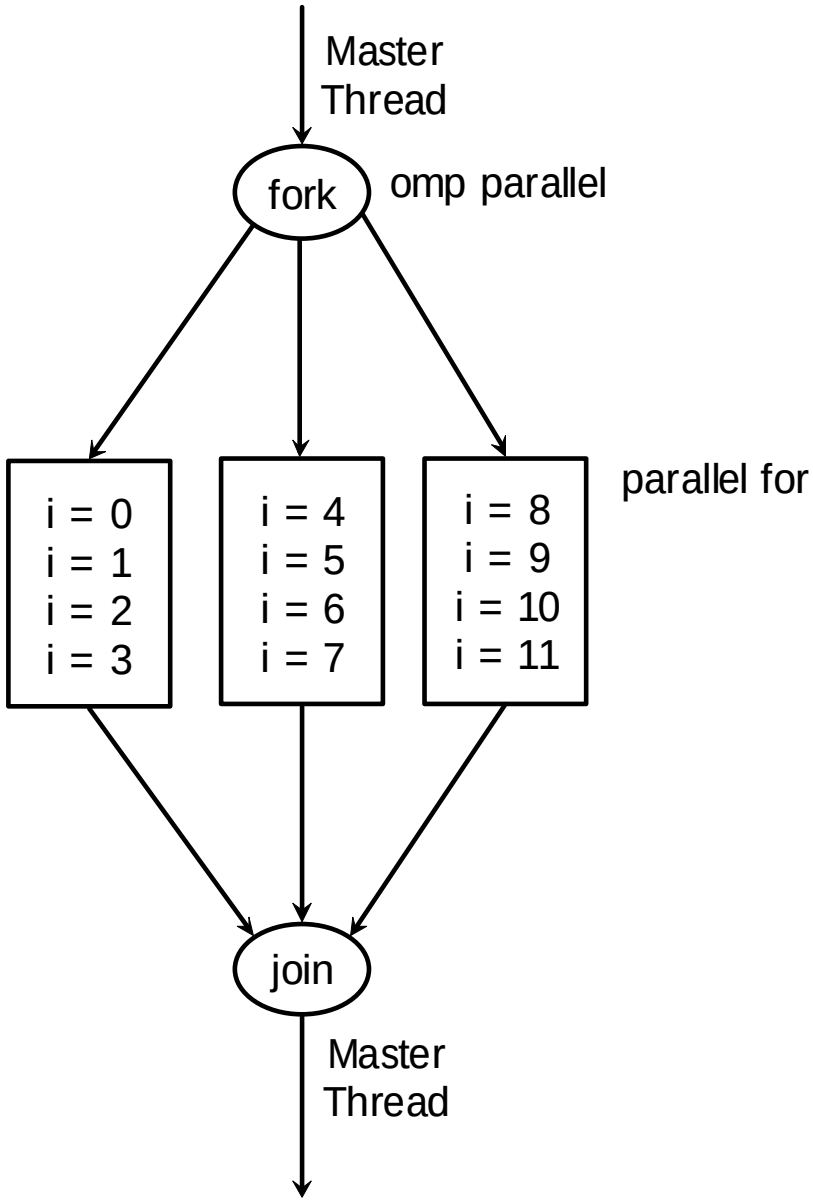
Parallel For

```
#pragma omp parallel
{
#pragma omp for
        for (i = 0; i < 12; i++)
                c[i] = a[i] + b[i];
}
```

Data decomposition

12 loop iterations divided among 3 CPU cores

Each core executes 4 loop iterations in parallel



Sections

```
#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp sections
    {

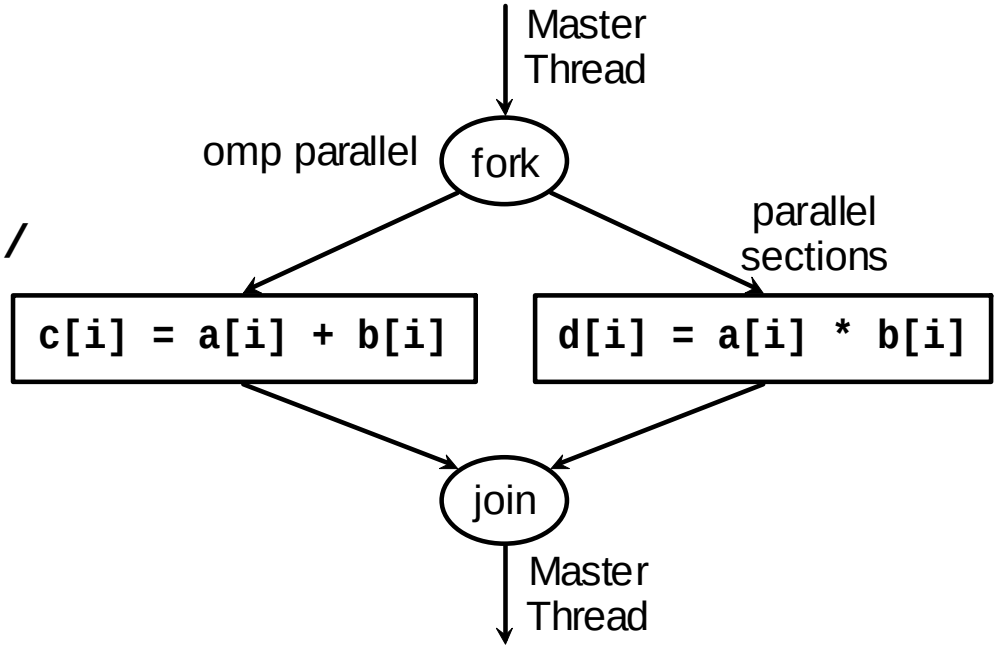
    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */
  }  /* end of parallel section */
}
```

Functional decomposition

Enclosed sections of code divided among threads in team



Message Passing Example — Vector Product

Compute $\sum_{i=0}^{3} a[i]*b[i]$ from data pre-distributed to nodes

P0	P1	P2	P3
load R_a, a	load R_a, a	load R_a, a	load R_a, a
load R_b, b	load R_b, b	load R_b, b	load R_b, b
$R_a \leftarrow R_a * R_b$	$R_a \leftarrow R_a * R_b$	$R_a \leftarrow R_a * R_b$	$R_a \leftarrow R_a * R_b$
send P1, R_a →	recv P0, R_b	send P3, R_a →	recv P2, R_b
	$R_a \leftarrow R_a + R_b$		$R_a \leftarrow R_a + R_b$
	send P3, R_a →		recv P1, R_b
			$R_a \leftarrow R_a + R_b$
			store p, R_a

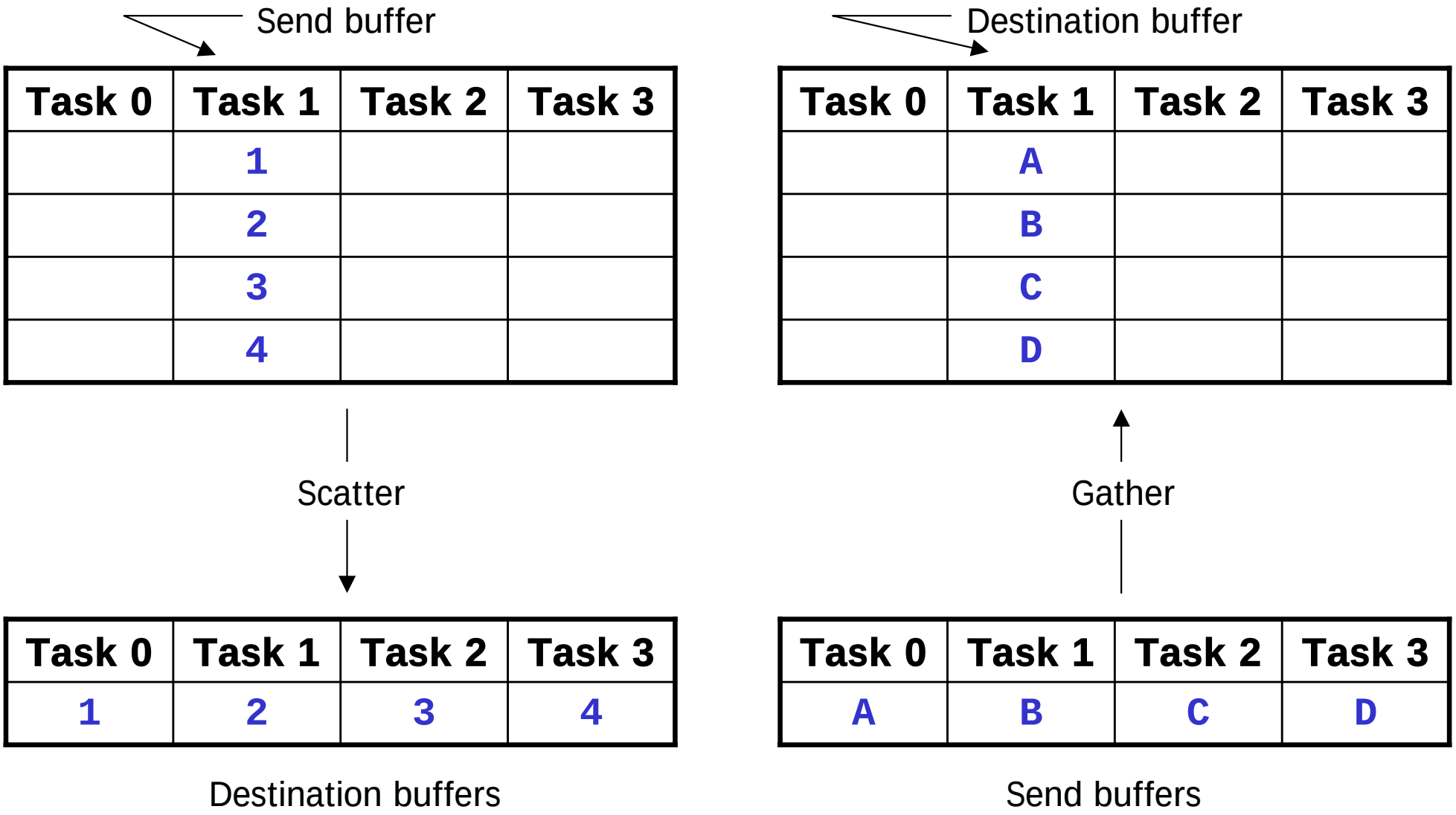
Message overhead

- Source or destination
- Time of creation

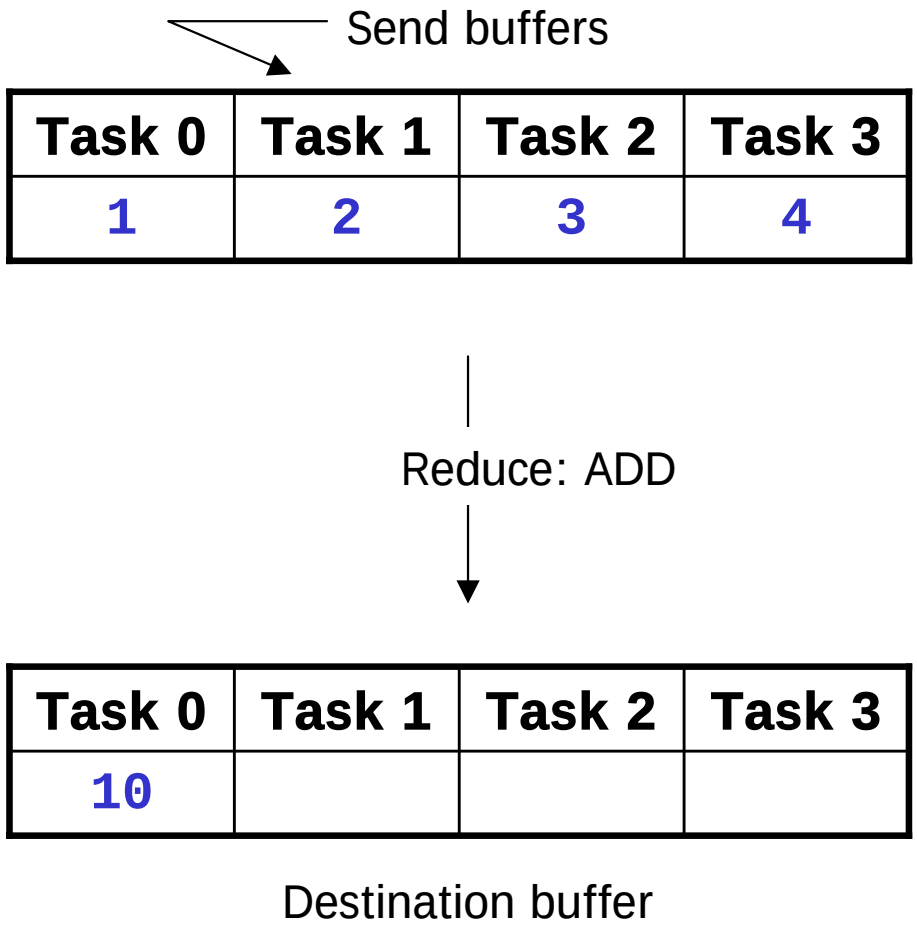
Sequential consistency guaranteed by message overhead

- P3 distinguishes two reads (receives) from P1 and P2 by source address
- No data hazard

Scatter and Gather



Reduce


Send buffers
Task 0
Task 1
Task 2
Task 3
1
2
3
4
Reduce: ADD
Task 0
Task 1
Task 2
Task 3
10
Destination buffer

MPI "Hello World"

```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
  char message[20];
  int myrank;              /* myrank = this process number */
  MPI_Status status;       /* MPI_Status = error flags */
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
                           /* MPI_COMM_WORLD = list of active MPI processes */
  if (myrank == 0)         /* code for process zero */
  {
    strcpy(message,"Hello, there");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else                     /* code for process one */
  {
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    printf("received :%s:\n", message);
  }
  MPI_Finalize();
}
```

Ref: "MPI: A Message-Passing Interface Standard Version 1.3"