

Advanced OpenMP

Swarnendu Biswas

ACM Winter School 2019 on High Performance Computing

Content influenced by many excellent references, see References slide for acknowledgements.

Warm up!

```
    cout << "This is serial code";
#pragma omp parallel
{
#pragma omp single
    cout << omp_get_num_threads();
}
cout << "This is serial code";
omp_set_num_threads(4);
#pragma omp parallel
{
#pragma omp single
    cout << omp_get_num_threads();
}
```

```
#pragma omp parallel num_threads(12)
{
#pragma omp single
    cout << omp_get_num_threads();
}
cout << "This is serial code";
cout << omp_get_nested();
omp_set_nested(1);
omp_set_num_threads(2);
#pragma omp parallel
{
#pragma omp parallel
{
    cout << omp_get_thread_num();
} } }
```

Distributing Work

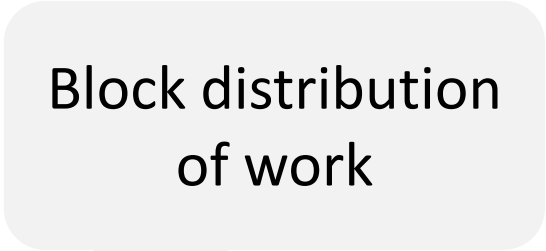
- Threads can perform disjoint work division using their thread ids and knowledge of total # threads

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int t_id = omp_get_thread_num();
    for (int i = t_id; i < 1000; i += omp_get_num_threads()) {
        A[i] = foo(i);
    }
}
```

Cyclic distribution
of work

Distributing Work

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int t_id = omp_get_thread_num();
    int num_thrs = omp_get_num_threads();
    int b_size = 1000 / num_thrs;
    for (int i = t_id*b_size; i < (t_id+1)*b_size; i += num_thrs) {
        A[i]= foo(i);
    }
}
```

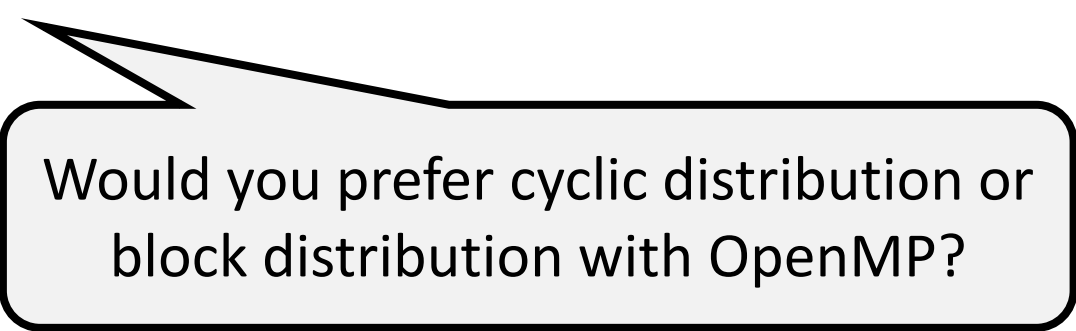


Block distribution
of work

Cyclic vs Block Distribution of Work

- Say I have a computation like

```
for (int i = 0; i < N; i++) {  
    A[i] = B[i] + C[i];  
}
```



Would you prefer cyclic distribution or block distribution with OpenMP?

Use of `nowait` clause

Can be useful if the two loops are independent

```
# pragma omp for nowait
for ( /* ... */ ) {
    // .. first loop ..
}
```

```
# pragma omp for
for ( /* ... */ ) {
    // .. second loop ..
}
```

Use of `nowait` clause

Can be useful if the two loops are independent

```
# pragma omp for nowait
for ( /* ... */ ) {
    // .. first loop ..
}
```

```
# pragma omp for
for ( /* ... */ ) {
    // .. second loop ..
}
```

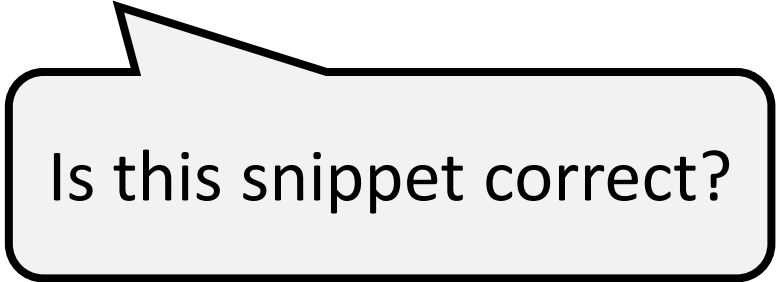
```
# pragma omp for nowait
for (int i=0; i<N; i++ ) {
    a[i] = b[i] + c[i];
}
```

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

Data Scope

```
int n = 10;  
std::vector<int> vector(n);  
int a = 10;
```

```
#pragma omp parallel for default(none) shared(n, vector)  
for (int i = 0; i < n; i++) {  
    vector[i] = i*a;  
}
```



Is this snippet correct?

Explicit Tasks

Explicit Task Constructs in OpenMP

- Not all programs have simple loops that OpenMP can parallelize
- OpenMP can only parallelize loops in a basic standard form with loop counts known at runtime

Explicit Task Constructs in OpenMP

- Not all programs have simple loops OpenMP can parallelize
- OpenMP can only parallelize loops in a basic standard form with loop counts
- Consider a program to traverse a linked list

```
p=head;
while (p) {
    dowork(p);
    p = p->next;
}
```

One Potential Solution

1

```
while (p != NULL) {  
    p = p->next;  
    count++;  
}
```

2

```
p = head;  
for (i=0; i<count; i++) {  
    parr[i] = p;  
    p = p->next;  
}
```

3

```
#pragma omp parallel  
{  
    #pragma omp for schedule(static,1)  
    for (i=0; i<count; i++)  
        dowork(parr[i]);  
}
```

One Potential Solution

1

```
while (p != NULL) {  
    p = p->next;  
    count++;  
}
```

3

```
#pragma omp parallel  
{  
    (static,1)
```

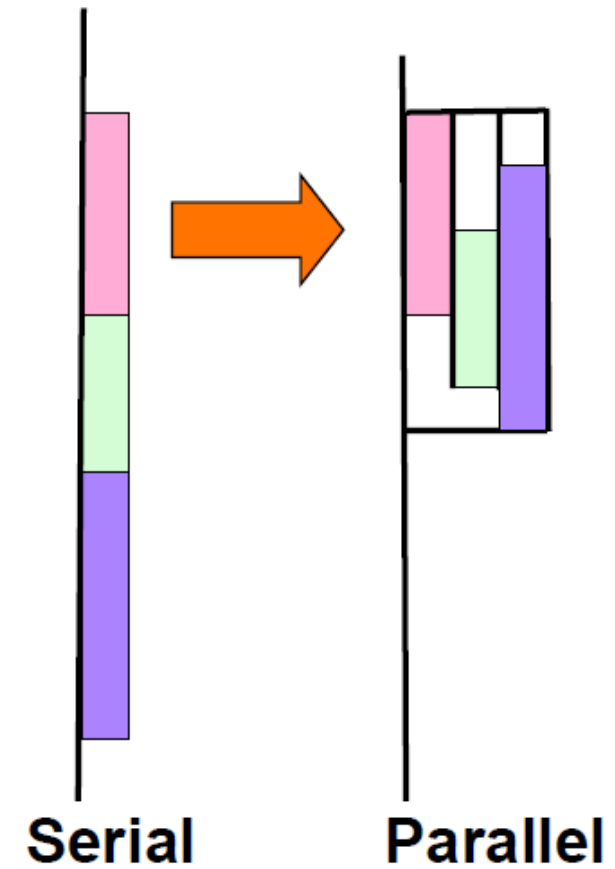
2

```
p = head;  
for (i=0; i<count; i++) {  
    parr[i] = p;  
    p = p->next;  
}
```

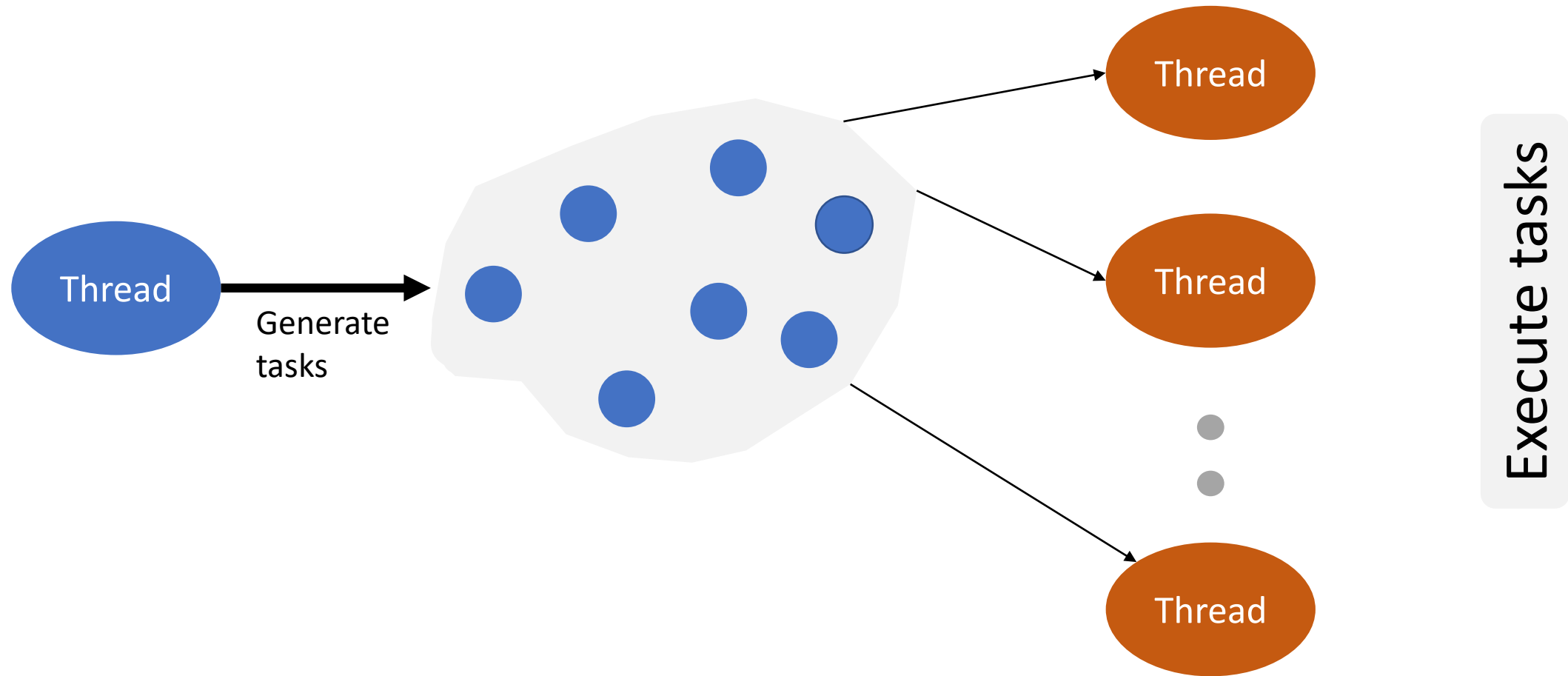
This works, but is inelegant (had to use a vector or array as an intermediate storage) and is inefficient (requires multiple passes over the data)

Tasks in OpenMP

- Explicit tasks were introduced in OpenMP 3.0
- Tasks are independent units of work, composed of
 - code to execute
 - data to compute
 - control variables
- Threads are assigned to perform the work of each task
- Runtime system decides when tasks are executed
 - Tasks may be deferred or executed immediately

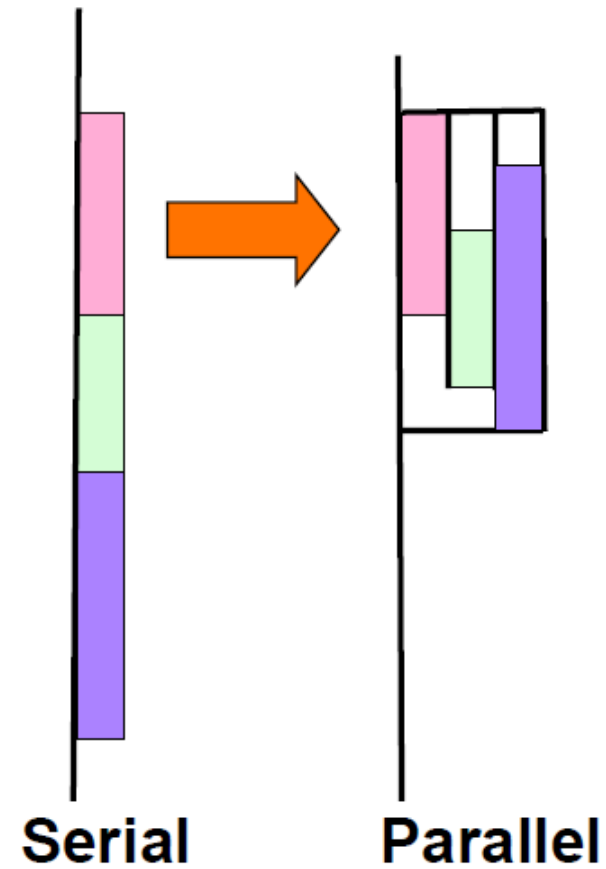


The Tasking Concept in OpenMP



Tasks in OpenMP

- The task construct includes a structured block of code
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- Tasks can be nested
 - A task may itself generate tasks



Task Directive

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        fred();
        #pragma omp task
        daisy();
        #pragma omp task
        billy();
    }
}
```

Thread 0 packages data

Tasks executed by some
thread in some order

All tasks complete
before this barrier ends

Task Completion

- `#pragma omp barrier`
 - All tasks created by any thread in the current team are guaranteed to be completed at the barrier exit
- `#pragma omp taskwait`
 - Wait for child tasks to complete
 - Applies only to children, not descendants

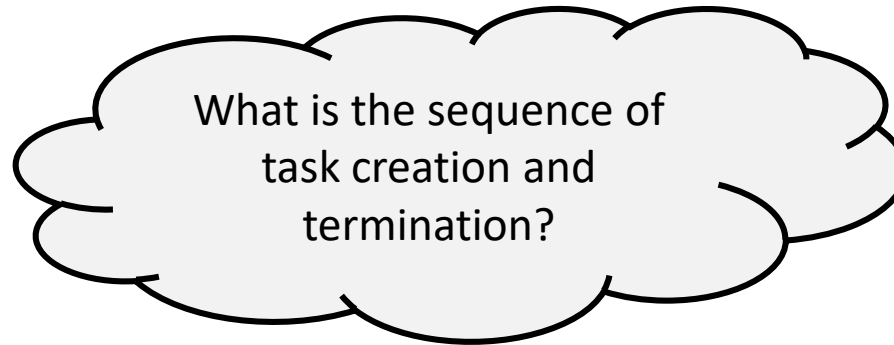
Example of Tasks

```
#pragma omp parallel
{
    #pragma omp single
    {
        cout << "A ";
    }
    #pragma omp task
    cout << "race ";
    #pragma omp task
    cout << "car ";
    cout << "is fun to watch!";
}
}
```

```
#pragma omp parallel
{
    #pragma omp single
    {
        cout << "A ";
    }
    #pragma omp task
    cout << "race ";
    #pragma omp task
    cout << "car ";
    #pragma omp taskwait
    cout << "is fun to watch!";
}
}
```

Task Synchronization

```
#pragma omp parallel num_threads(n)
{
    #pragma omp task
    funcA();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        funcB();
    }
}
```



An Example with OpenMP Tasks

```
#pragma omp parallel
{
    #pragma omp single private(p)
    {
        p = listhead ;
        while (p) {
            #pragma omp task firstprivate(p)
            {
                process (p);
            }
            p=next (p) ;
        } } }
```

Binary Tree Traversal

```
void postorder(node *p) {  
    if (p->left) {  
        postorder(p->left);  
    }  
    if (p->right) {  
        postorder(p->right);  
    }  
    process(p->data);  
}
```

Scheduling of Tasks


- Tasks are tied to the thread that first executes them
 - Need not necessarily be the creator thread
- Scheduling constraints
 - Only the thread a task is tied to can execute it
 - A task can be suspended at task scheduling points
 - Task creation and finish, taskwait, barrier, taskyield, ...

Scheduling of Tasks

- When a thread encounters a task scheduling point, it may do one of the following
 - Begin execution of a tied task bound to the current team
 - Resume any suspended task region, bound to the current team, to which it is tied
 - Begin execution of an untied task bound to the current team
 - Resume any suspended untied task region bound to the current team
- If more than one of the above choices is available, it is unspecified as to which will be chosen

Scheduling of Tasks

- Tasks created with the untied clause are never tied
 - Resume at task scheduling points possibly by different threads
 - May provide better load balancing

An orange speech bubble with a tail pointing towards the top-left, containing the text "Great! But is that all?".

Great! But is
that all?

Example

```
#define LARGE_NUMBER 10000000  
double item[LARGE_NUMBER];  
extern void process(double);
```

```
int main() {  
    #pragma omp parallel  
    {  
        #pragma omp single  
        {  
            int i;  
            #pragma omp task untied  
            // i is firstprivate, item is shared  
            {  
                for (i=0; i<LARGE_NUMBER; i++) {  
                    #pragma omp task  
                    process(item[i]);  
                } } } } }
```

Scheduling of Tasks

- Tasks created with the untied clause are never tied
 - Resume at task scheduling points possibly by different threads
 - May provide better load balancing
- Things to remember!
 - Avoid threadprivate variables
 - Avoid dependence of thread ids
 - Be careful while using critical regions and locks

The taskyield Directive

- Specifies that the current task can be suspended in favor of execution of a different task
 - Hint to the OpenMP runtime

```
void foo(omp_lock_t* lock, int n) {  
    for(int i = 0; i < n; i++)  
#pragma omp task  
    {  
        something_useful();  
        while(!omp_test_lock(lock)) {  
            #pragma omp taskyield  
        }  
        something_critical();  
        omp_unset_lock(lock);  
    }  
}
```

Other Details in Tasking

Other clauses

- `priority` – Hint to the runtime system for task execution order
- `depends` – Task dependence is fulfilled when the predecessor task has completed
- `final` – Stop task decomposition at a certain depth for recursive problems, exposes enough parallelism but reduces overhead for small data sizes

SIMD Programming

SPMD Programming

Single Program Multiple Data

- Each thread runs same program
- Selection of data, or branching conditions, based on thread id
 - General and common parallel programming paradigm

OpenMP implementation

- Perform work division in parallel loops
- Query `thread_id` and `num_threads`
- Partition work among threads

Different Levels of Parallelism in Hardware

Instruction-level Parallelism

- Microarchitectural techniques like pipelining, OOO execution, and superscalar

Vector-level Parallelism

- Use Single Instruction Multiple Data (SIMD) vector processing instructions and units

Thread-level Parallelism

- Hyperthreading

Vectorization

- Process of transforming a scalar operation on single data elements at a time (SISD) to an operation on multiple data elements at once (SIMD)
- Loop vectorization transforms a program so that the same operation is performed at the same time on several vector elements

Vectorization

```
double *a, *b, *c;  
for (int i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

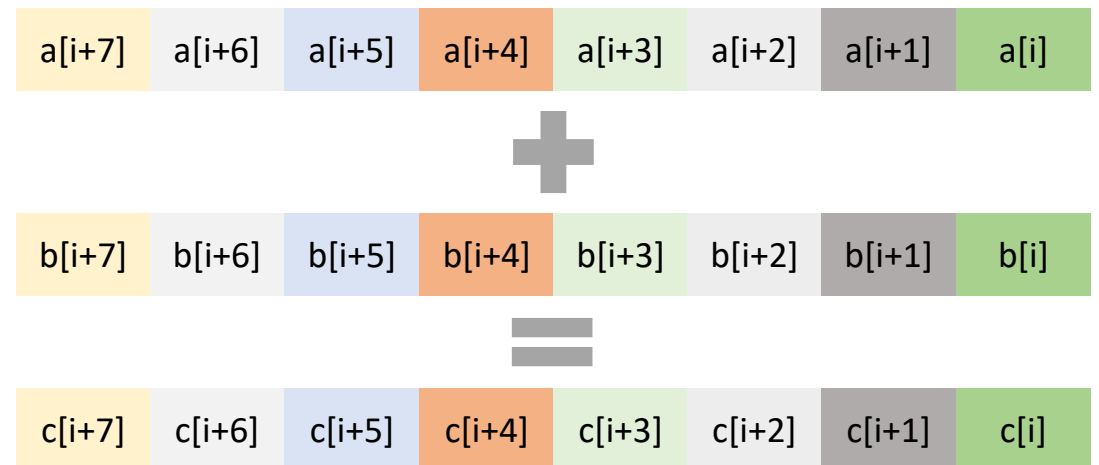
Scalar mode

- One instruction produces one result
 - vaddsd/vaddss



Vector mode

- One instruction can produce multiple results
 - vaddpd/vaddps



Vectorization

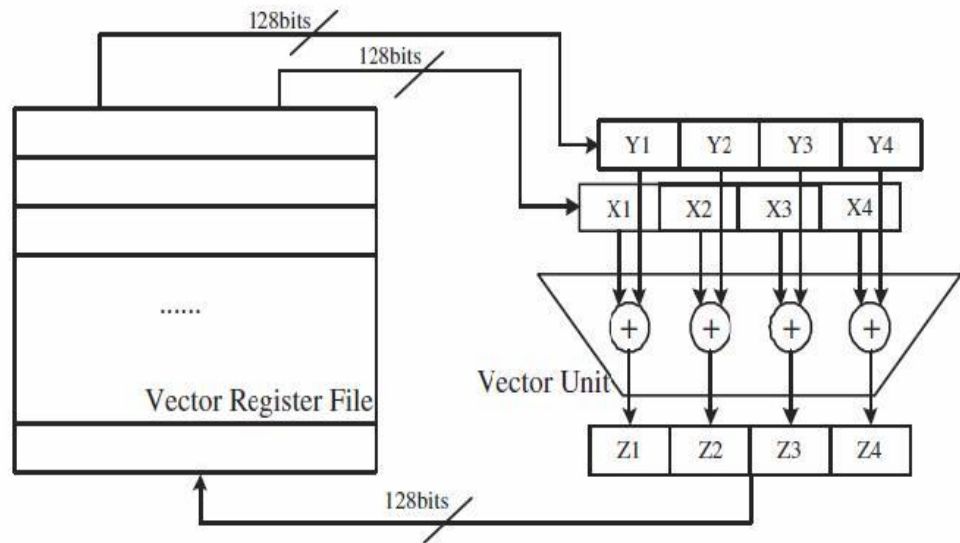
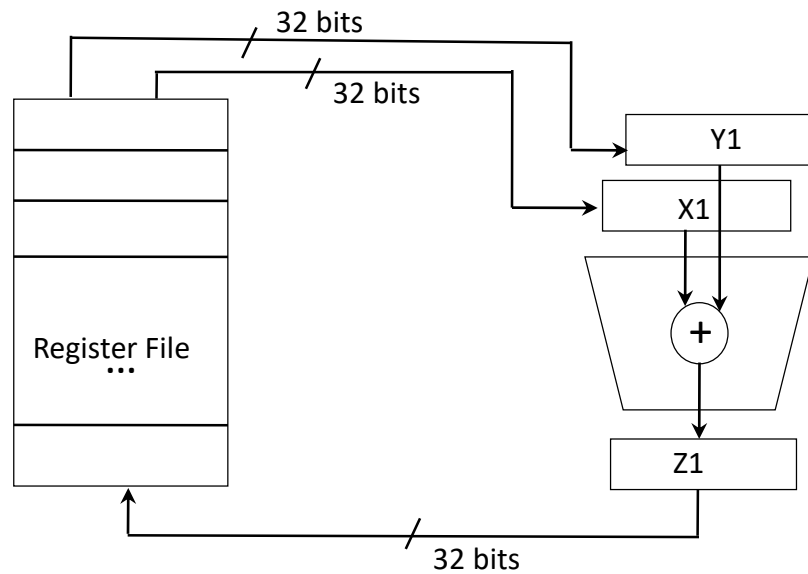
n
times

```
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st r3, addr3
```

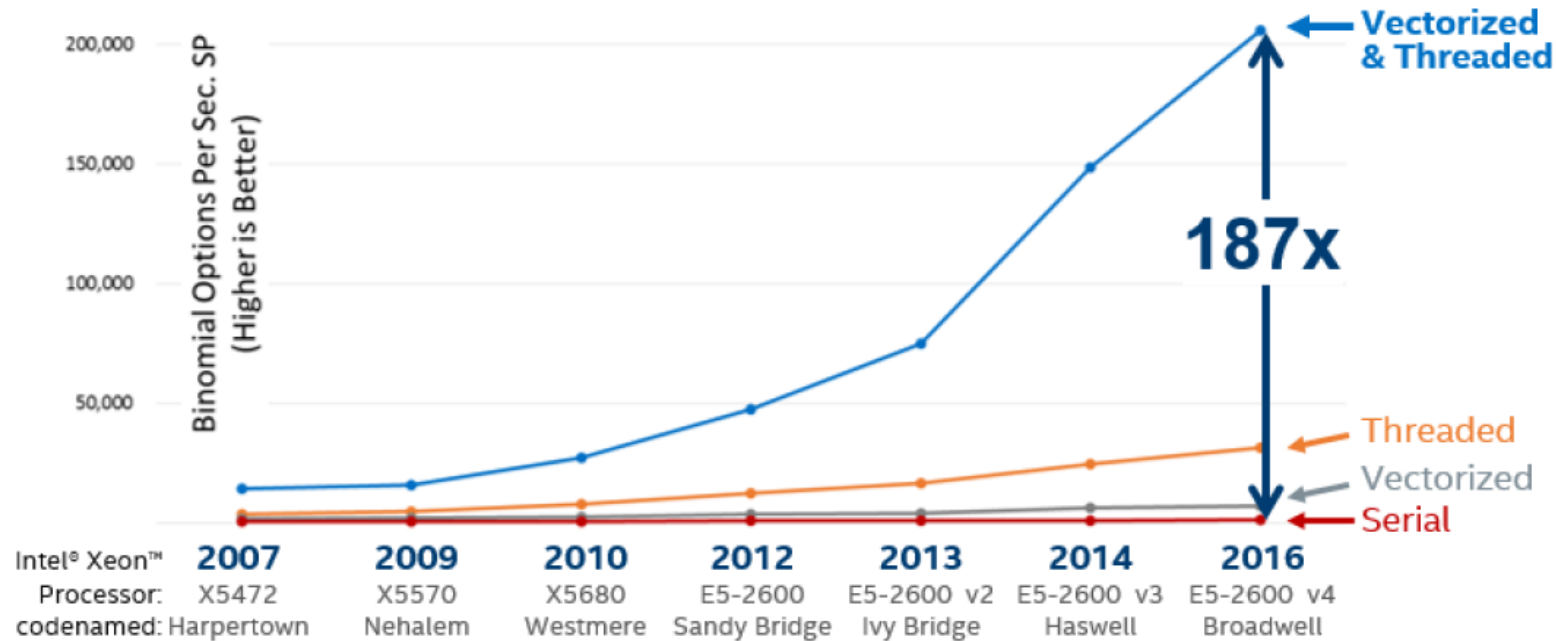
```
for (i=0; i<n; i++)  
  c[i] = a[i] + b[i];
```

n/4
times

```
ldv vr1, addr1
ldv vr2, addr2
addv vr3, vr1, vr2
stv vr3, addr3
```



The combined effect of vectorization and threading



The Difference Is Growing With Each New Generation of Hardware

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to <http://www.intel.com/performance> [Configurations](#) at the end of this presentation.

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



Intel-Supported SIMD Extensions

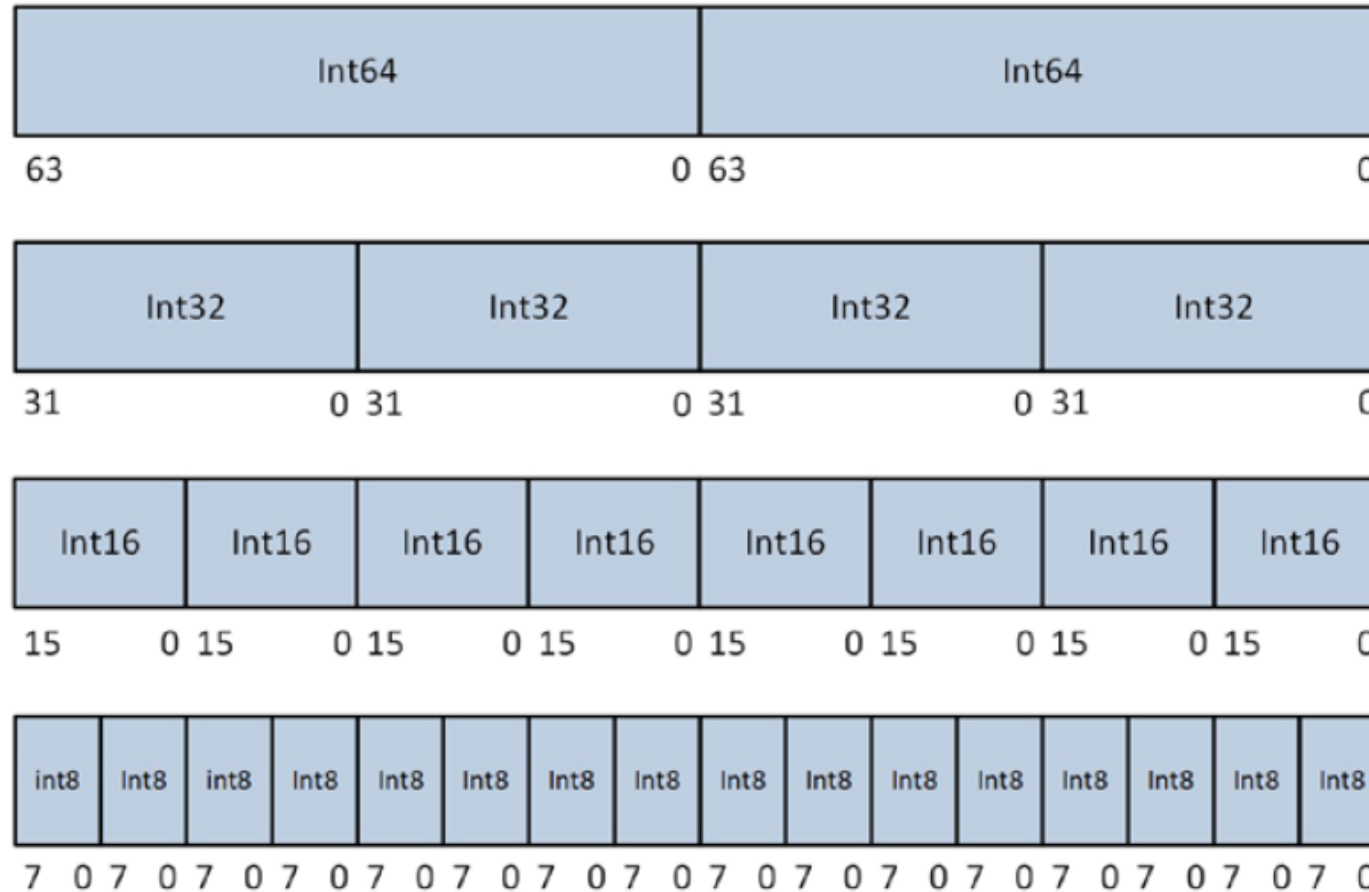
SIMD extensions	Width (bits)	Dual precision (64 bit) calculations	Single precision (32 bit) calculations	Introduced
SSE2/SSE3/SSE4	128	2	4	~2001-2007
AVX/AVX2	256	4	8	~2011-2015
AVX-512	512	8	16	~2017

Other platforms that support SIMD have different extensions

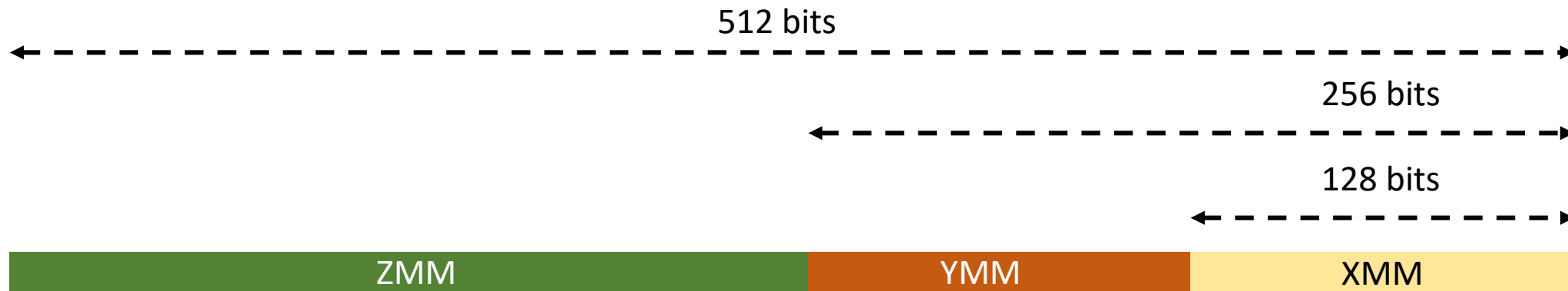
SIMD Vectorization

- Use of SIMD units can speed up the program
- Intel SSE has 128-bit vector registers and functional units
 - 4 32-bit single precision floating point numbers
 - 2 64-bit double precision floating point numbers
 - 4 32-bit integer numbers
 - 2 64 bit integer
 - 8 16-bit integer or shorts
 - 16 8-bit bytes or chars
- Assuming a single ALU, these SIMD units can execute 4 single precision floating point number or 2 double precision operations in the time it takes to do only one of these operations by a scalar unit

128-bit wide operands using integers



Intel-Supported SIMD Extensions



64-bit architecture		
SSE	XMM0-XMM15	
AVX	YMM0-YMM15	Low- order 128 bits of each YMM register are aliased to a corresponding XMM register
AVX-512	ZMM-ZMM31	low-order 256 and 128 bits are aliased to registers YMM0-YMM31 and XMM0-XMM31 respectively

x86-64 Vector Operations

[] – required
() - optional

- Example instructions
 - Move: (V)MOV[A/U]P[D/S]
 - Comparing: (V)CMP[P/S][D/S]
 - Arithmetic operations: (V)[ADD/SUB/MUL/DIV][P/S][D/S]
- Instruction decoding
 - V – AVX
 - P, S – packaged, scalar
 - A, U – aligned, unaligned
 - D, S – double, single
 - B, W, D, Q – byte, word, doubleword, quadword integers

x86-64 Vector Operations

Instruction

Explanation

`movss xmm2, xmm1`

`xmm2 = xmm2 + xmm1 (SSE/SSE2)`

`vmovapd ymmword ptr [edi], ymm1`

Move aligned packed double-precision floating-point values from ymm2/mem to ymm1.

AVX Scalar Floating-Point Instruction Examples

Instruction

Explanation

`vaddss xmm0, xmm1, xmm2`

$\text{xmm0}[31:0] = \text{xmm1}[31:0] + \text{xmm2}[31:0]$
 $\text{xmm0}[127:32] = \text{xmm1}[127:32]$
 $\text{ymm0}[255:128] = 0$

`vaddsd xmm0, xmm1, xmm2`

$\text{xmm0}[63:0] = \text{xmm1}[63:0] + \text{xmm2}[63:0]$
 $\text{xmm0}[127:64] = \text{xmm1}[127:64]$
 $\text{ymm0}[255:128] = 0$

Cumulative (app.) # of Vector Instructions



Copyright © 2015 Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.

Optimization Notice



Vectorize Code

- Auto-vectorizing compiler
- Vector intrinsics
- Assembly language

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

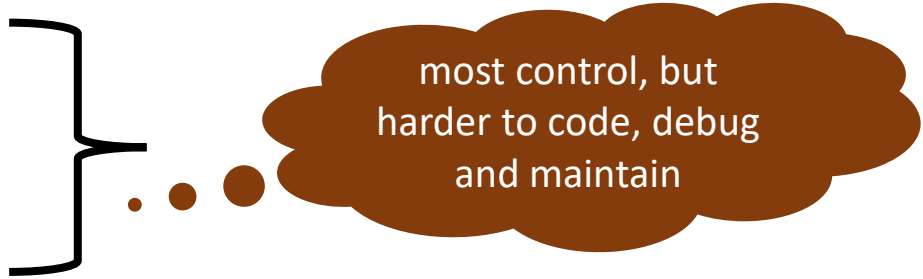
```
void example(){  
    __m128 rA, rB, rC;  
    for (int i = 0; i < LEN; i+=4){  
        rA = _mm_load_ps(&a[i]);  
        rB = _mm_load_ps(&b[i]);  
        rC = _mm_add_ps(rA,rB);  
        _mm_store_ps(&c[i], rC);  
    }  
}
```

```
..B8.5  
    movaps    a(,%rdx,4), %xmm0  
    addps     b(,%rdx,4), %xmm0  
    movaps    %xmm0, c(,%rdx,4)  
    addq      $4, %rdx  
    cmpq      %rdi, %rdx  
    jl        ..B8.5
```

easy, but low
control

hard, but
most control

Vectorize Code

- Auto-vectorization
 - Compiler vectorizes automatically – No code changes
 - Semi auto-vectorization – Use pragmas as hints to guide compiler
 - Explicit vector programming – OpenMP SIMD pragmas
 - SIMD/Vector intrinsics
 - Inline assembly language
- 
- Use SIMD-capable libraries like Intel Math Kernel Library (MKL)

Auto-Vectorization

Transparent to programmers

Compilers can apply other transformations

Portability of code across architectures

- Vectorization instructions may differ but compilers take care of it

Auto-Vectorization

Transparent to programmers

Compilers can apply other transformations

Portability of code across architectures

- Vectorization instructions may differ but compilers take care of it

Compilers may fail to vectorize

- Programmers may give hints to help the compiler
- Programmers may have to manually vectorize their code

Why Auto Vectorizers Fail?

- Data dependences
- Unaligned accesses
- Function calls in loop block
- Complex control flow, conditional branches
- Loop not “countable”, e.g., upper bound not a runtime constant
- Loop body too complex (register pressure)
- Vectorization seems inefficient

How about SIMD support?

- Support in older versions of OpenMP (< 4.0) required vendor-specific extensions
 - Programming models (e.g., Intel Cilk Plus)
 - Compiler pragmas (e.g., `#pragma vector`)
 - Low-level constructs or intrinsics (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + 10;
}
```

simd Construct

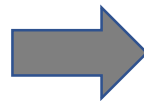
- `#pragma omp simd ...`
 - Can be applied to a loop to indicate that the loop can be transformed to a SIMD loop
 - Use SIMD instructions
 - Partition loop into chunks that fit a SIMD vector register
 - **Does not parallelize the loop body**
- `#pragma omp declare simd`
 - Applied to a function to enable creation of one or more versions to allow for SIMD processing

SIMD Function Vectorization

```
#pragma omp declare simd ...  
function-definition-or-declaration
```

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

```
#pragma omp declare simd  
float min(float a, float b) {  
    return a < b ? a : b;  
}
```



```
// Vector version  
vec8 min_v(vec8 a, vec8 b) {  
    return a < b ? a : b;  
}
```

simd Worksharing Construct

- `#pragma omp for simd ...`
- Parallelize and vectorize a loop nest
 - Distribute a loop's iteration space across a thread team
 - Subdivide loop chunks to fit a SIMD vector register

OpenMP Example

```
uint64_t seq_sum = 0;
for (int i = 0; i < N; i++) {
    seq_sum += x[i];
}

uint64_t vec_sum = 0;
double start_time, end_time;
start_time = omp_get_wtime();
#pragma omp simd // num_threads(4)
for (int i = 0; i < N; i++) {
    vec_sum += x[i];
}
end_time = omp_get_wtime();
assert(seq_sum == vec_sum);
```

```
cout << "SIMD time: " << (end_time-
start_time) << " seconds\n";

uint64_t for_sum = 0;
start_time = omp_get_wtime();
#pragma omp parallel for simd reduction(+ :
for_sum) num_threads(4)
for (int i = 0; i < N; i++) {
    for_sum += x[i];
}
end_time = omp_get_wtime();
assert(seq_sum == for_sum);
cout << "Parallel SIMD time: " << (end_ti
me-start_time)<< " seconds\n";
```

OpenMP Example

```
uint64_t seq_sum = 0;
for (int i = 0; i < N; i++) {
    seq_sum += x[i];
}
```

```
cout << "SIMD time: " << (end_time-
start_time) << " seconds\n";
```

```
uint64_t for_sum = 0;
```

```
swarnendu@DESKTOP-U56QRQ5: ~/iitk-workspace/parallel-computing/build
```

```
swarnendu@DESKTOP-U56QRQ5:~/iitk-workspace/parallel-computing/build$ ./bin/omp-simd
```

```
SIMD time: 0.164496 seconds
```

```
Parallel SIMD time: 0.0712165 seconds
```

```
swarnendu@DESKTOP-U56QRQ5:~/iitk-workspace/parallel-computing/build$
```

```
vec_sum += x[i];
}
end_time=omp_get_wtime();
assert(seq_sum == vec_sum);
```

```
assert(seq_sum == for_sum);
cout << "Parallel SIMD time: " << (end_t
ime-start_time)<< " seconds\n";
```

OpenMP Memory Model

Busy-Wait Paradigm

```
Object X = null;  
boolean done= false;
```

Thread T1

```
X = new Object();  
done = true;
```

Thread T2

```
while (!done) {}  
if (X != null)  
    X.compute();
```


Thread T1

```
X = new Object();
```

```
done = true;
```

Thread T2

```
temp = done;  
while (!temp) {}
```



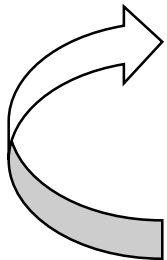
Infinite loop



Thread T1

```
done = true;
```

```
X = new Object();
```



Thread T2

```
while (!done) {}  
X.compute();
```

NPE



What Value Can a Read Return?

`X = 0`
`done = 0`

Core C1

```
S1: store X, 10  
S2: store done, 1
```

Core C2

```
L1: load r1, done  
B1: if (r1 != 1) goto L1  
L2: load r2, X
```

Reordering of Accesses by Hardware

Different
addresses!

Store-store

Load-load

Load-store

Store-load

Reordering of Accesses by Hardware

Different
addresses!

Store-store

Correct in a single-threaded context

Non-trivial in a multithreaded context

Store-load

What values can a load return?

Return the “last” write

Uniprocessor: program order

Multiprocessor: ?

Memory Consistency Model

Set of rules that govern how systems process memory operation requests from multiple processors

- Determines the order in which memory operations appear to execute

Specifies the allowed behaviors of multithreaded programs executing with shared memory

- Both at the hardware-level and at the programming-language-level
- There can be multiple correct behaviors

Importance of Memory Consistency Models

Determines what optimizations are correct

Contract between the programmer and the hardware

Influences ease of programming and program performance

Impacts program portability

Dekker's Algorithm

```
flag1 = 0  
flag2 = 0
```

Core C1

```
S1: store flag1, 1  
L1: load r1, flag2
```

Core C2

```
S2: store flag2, 1  
L2: load r2, flag1
```

Can both r1 and r2 be set to zero?

Sequential Consistency

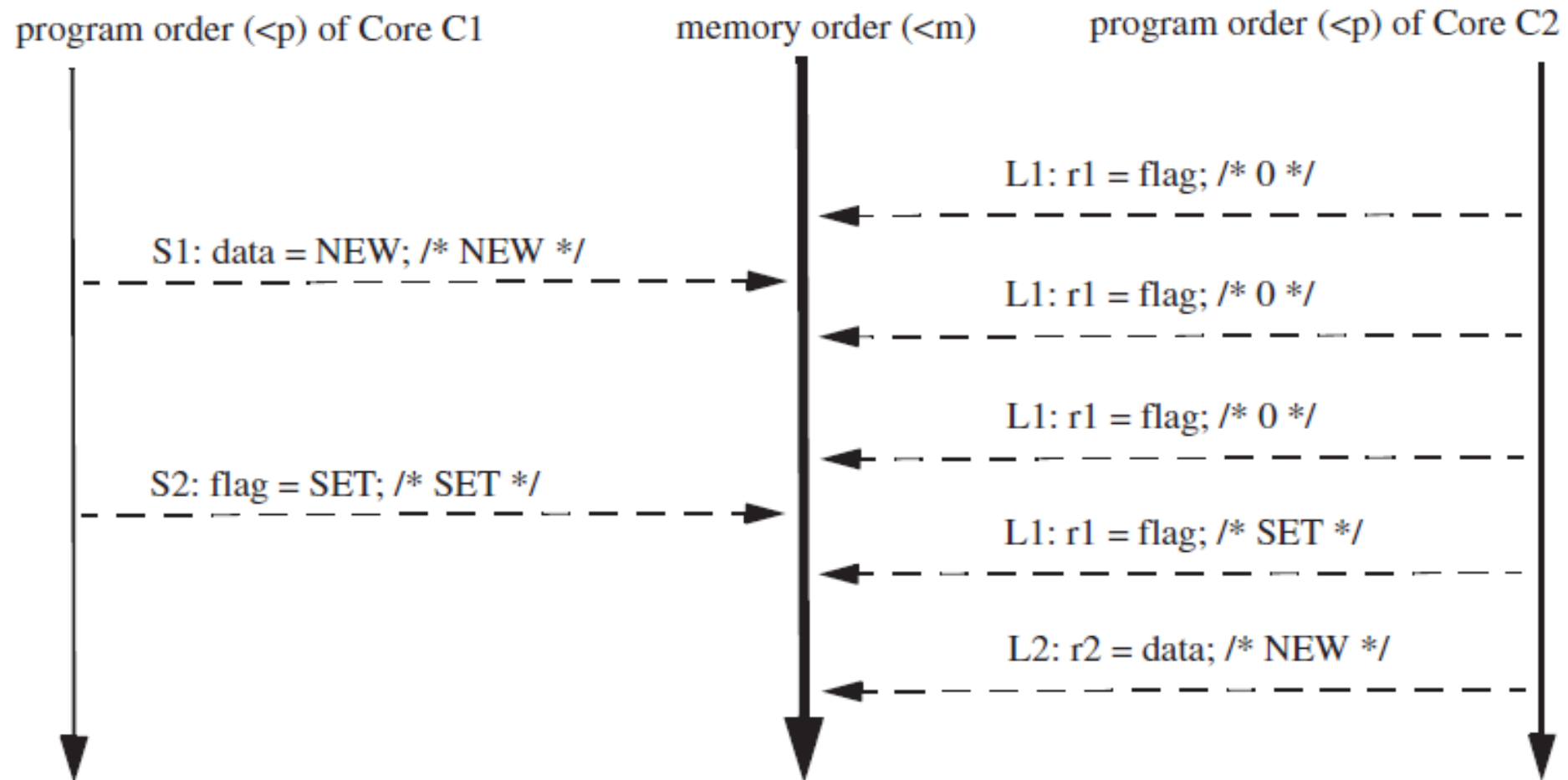
A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in **some sequential order**, and the operations of each individual processor appear in **the order specified by the program**.

Interleavings with SC

TABLE 3.1: Should r2 Always be Set to NEW?

Core C1	Core C2	Comments
S1: Store data = NEW; S2: Store flag = SET;	L1: Load r1 = flag; B1: if (r1 \neq SET) goto L1; L2: Load r2 = data;	/* Initially, data = 0 & flag \neq SET */ /* L1 & B1 may repeat many times */

Interleavings with SC



SC Formalism

Every load gets its value from the last store before it (in global memory order) to the same address

SC Rules

Suppose we
have two
addresses a
and b

- $a == b$ or $a \neq b$

Constraints

- if $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
- If $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
- If $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$
- If $S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$

End-to-end SC

Simple memory model that can be implemented both in hardware and in languages

Performance can take a hit

- Naive hardware
- Maintain program order - expensive for a write

Existing Memory Consistency Models

Hardware

- Sequential Consistency (SC)
- Total Store Order (TSO)
- Partial Store Order (PSO)
- Weak Ordering (WO)
- ...

Programming Languages

- Java
- C++ and OpenMP
- ...

Total Store Order

Allows reordering stores to loads

Can read own write early, not other's writes

Conjecture: widely-used x86 memory model is equivalent to TSO

TSO Rules

$a == b$ or $a != b$

- If $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
- If $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
- If $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$
- ~~If $S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$~~ /* Enables FIFO Write Buffer */

Every load gets its value from the last store before it to the same address

Support for FENCE Operations in TSO

If $L(a) <_p \text{FENCE} \Rightarrow L(a) <_m \text{FENCE}$

If $S(a) <_p \text{FENCE} \Rightarrow S(a) <_m \text{FENCE}$

If $\text{FENCE} <_p \text{FENCE} \Rightarrow \text{FENCE} <_m \text{FENCE}$

If $\text{FENCE} <_p L(a) \Rightarrow \text{FENCE} <_m L(a)$

If $\text{FENCE} <_p S(a) \Rightarrow \text{FENCE} <_m S(a)$

If $S(a) <_p \text{FENCE} \Rightarrow S(a) <_m \text{FENCE}$

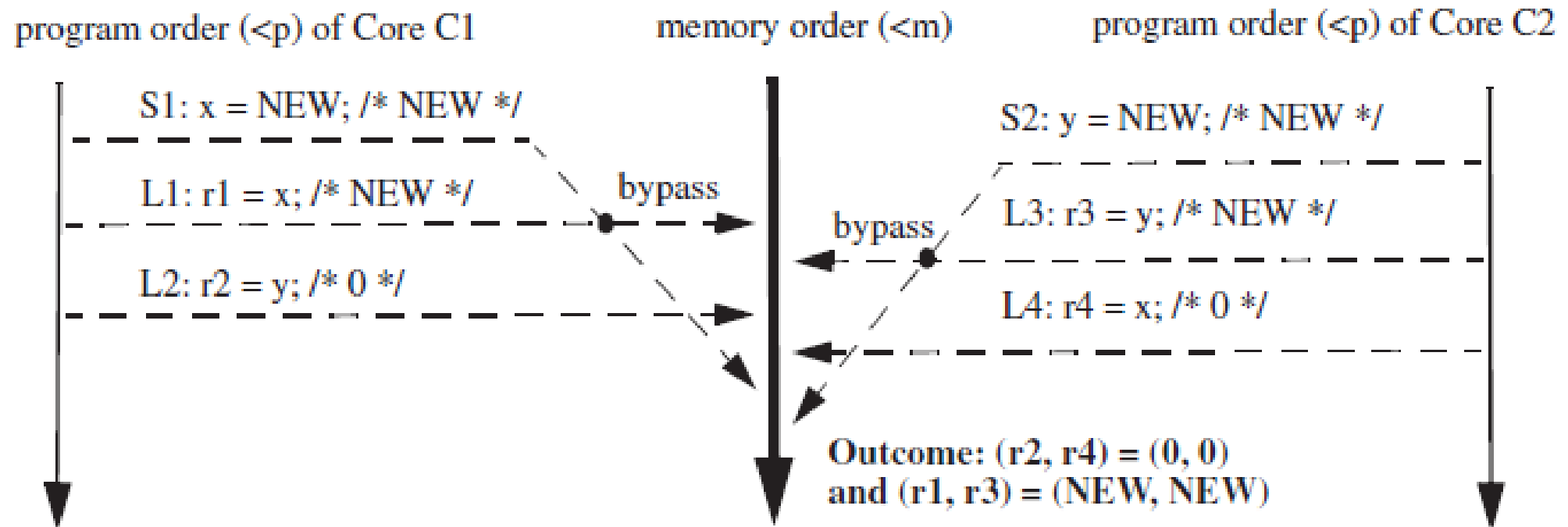
If $\text{FENCE} <_p L(a) \Rightarrow \text{FENCE} <_m L(a)$

Possible Outcomes with TSO

TABLE 4.3: Can r1 or r3 be Set to 0?

Core C1	Core C2	Comments
S1: x = NEW; L1: r1 = x; L2: r2 = y;	S2: y = NEW; L3: r3 = y; L4: r4 = x;	/* Initially, x = 0 & y = 0 */ /* Assume r2 = 0 & r4 = 0 */

Possible Outcomes with TSO



Partial Store Order (PSO)

- Allows reordering of store to loads and stores to stores
- Writes to **different** locations from the same processor can be pipelined or overlapped and are allowed to reach memory or other cached copies out of program order
- Can read own write early, not other's writes

Opportunities to Reorder Memory Operations

TABLE 5.1: What Order Ensures r2 & r3 Always Get NEW?

Core C1	Core C2	Comments
S1: data1 = NEW; S2: data2 = NEW; S3: flag = SET;	L1: r1 = flag; B1: if (r1 ≠ SET) goto L1; L2: r2 = data1; L3: r3 = data2;	/* Initially, data1 & data2 = 0 & flag ≠ SET */ /* spin loop: L1 & B1 may repeat many times */

Reorder Operations Within a Synchronization Block

TABLE 5.2: What Order Ensures Correct Handoff from Critical Section 1 to 2?

Core C1	Core C2	Comments
A1: acquire(lock) /* Begin Critical Section 1 */ Some loads L1i interleaved with some stores S1j /* End Critical Section 1 */ R1: release(lock)	A2: acquire(lock) /* Begin Critical Section 2 */ Some loads L2i interleaved with some stores S2j /* End Critical Section 2 */ R2: release(lock)	/* Arbitrary interleaving of L1i's & S1j's */ /* Handoff from critical section 1 */ /* To critical section 2 */ /* Arbitrary interleaving of L2i's & S2j's */

Optimization Opportunities

Non-FIFO coalescing write buffer

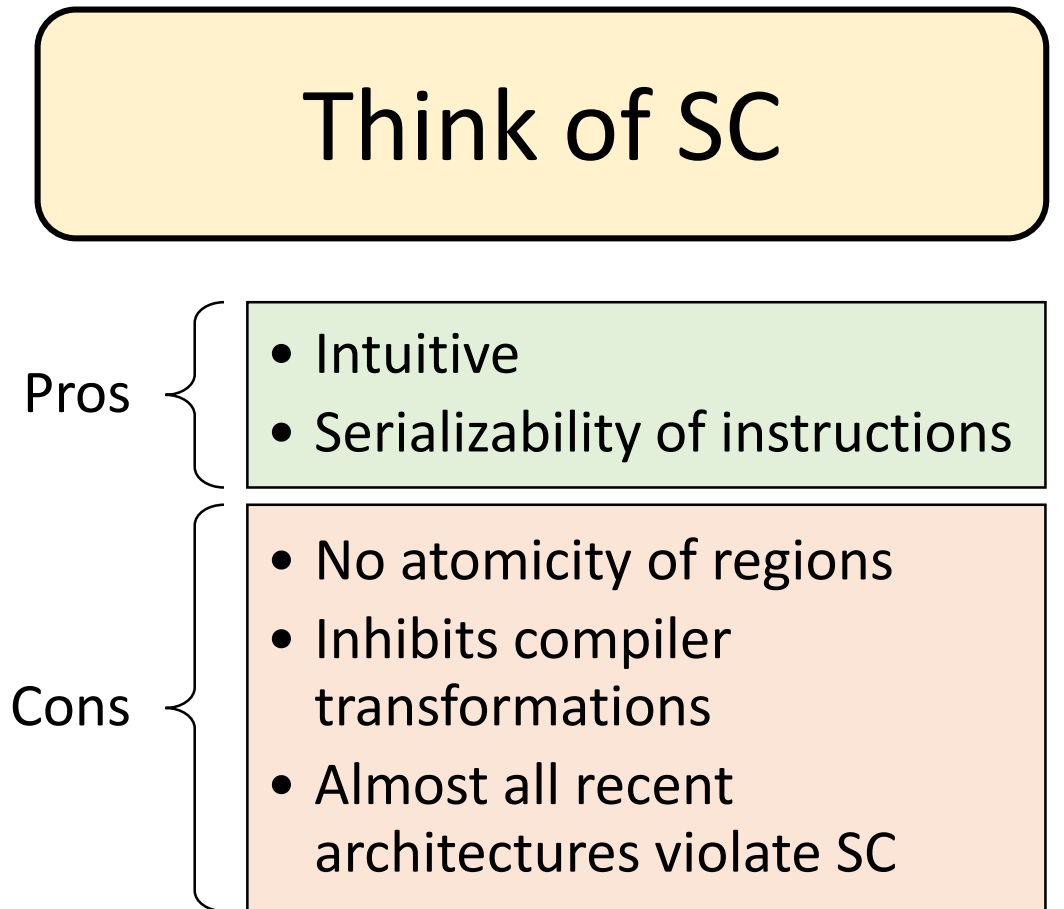
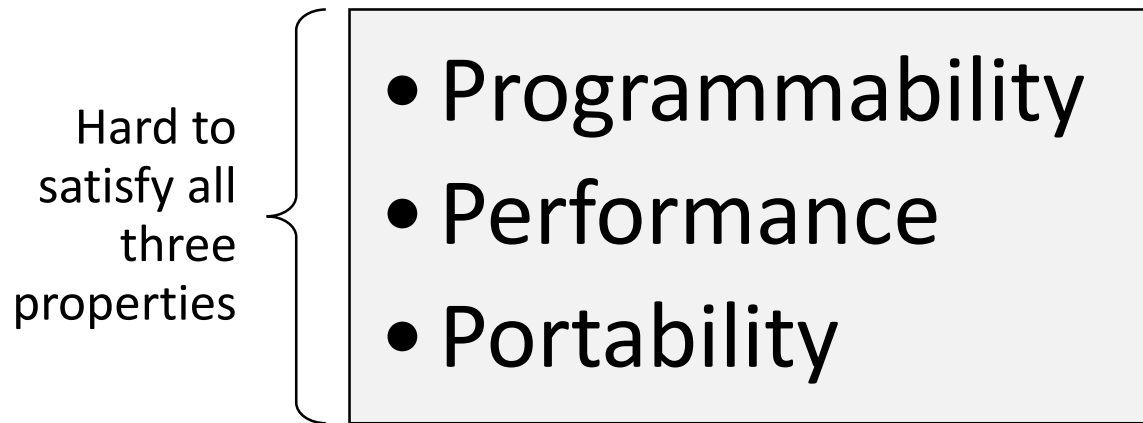
Support non-blocking reads

- Hide latency of reads
- Use lockup-free caches and speculative execution

Simpler support for speculation

- Need not compare addresses of loads to coherence requests
- For SC, need support to check whether the speculation is correct

Desirable Properties of a Memory Model



Relaxed Consistency Memory Model

- OpenMP supports a relaxed consistency shared memory model
 - Closely related to the weak ordering model
- Threads can maintain a temporary view of shared memory that is not consistent with other threads
- These temporary views are made consistent only at certain points in the program
- The operation that enforces consistency is called the flush operation

Synchronization Construct: `flush`

- `#pragma omp flush (list)`
- Identifies a point at which a thread is guaranteed to see a consistent view of memory with respect to the variables in “`list`”
 - Flush forces data to be updated in memory so other threads see the most recent value
- In the absence of a list, all shared objects are synchronized

Synchronization Construct: `flush`

- If `list` contains a pointer, the pointer is flushed, not the object referred to by the pointer
- It is recommended not to use flushes, excepting certain cases where you want to implement say your own spin lock
- Flushes are expensive, since they require compilers to generate memory fences

Semantics of the flush Operation

- A flush is a sequence point at which a thread is guaranteed to see a consistent view of memory
 - All previous read/writes by this thread have completed and are visible to other threads
 - No subsequent read/writes by this thread have occurred
- A flush operation is analogous to a fence in other shared memory APIs

Potential Benefits with Relaxed Consistency

- Relaxed memory model allows flexibility to OpenMP implementations
- Write to A
 - May complete immediately
 - May complete after the execution marked “...”

A = 1

...

...

#pragma omp flush(A)

Flush and Synchronization

- A flush operation is implied by OpenMP synchronizations
 - at entry/exit of parallel, critical, and ordered regions
 - at implicit and explicit barriers
 - at entry/exit of parallel worksharing regions
 - during lock APIs
 -

Flush and Synchronization

- This means if you are mixing reads and writes of a variable across multiple threads, you cannot assume the reading threads see the results of the writes unless:
 - The writing threads follow the writes with a construct that implies a flush
 - The reading threads precede the reads with a construct that implies a flush

Visibility of Data

- In order for a write of a variable on one thread to be guaranteed visible and valid on a second thread, the following operations must occur in the following order:
 1. Thread A writes the variable
 2. Thread A executes a flush operation
 3. Thread B executes a flush operation
 4. Thread B reads the variable

Reordering Example

```
1. a = ...;  
2. b = ...;  
3. c = ...;  
  
4. #pragma omp flush(c)  
5. #pragma omp flush(a, b)  
  
6. ...= a...b...;  
7. ...c...;
```

- 1 and 2 may not be moved after 5
- 4 and 5 maybe interchanged at will
- 6 may not be moved before 5

OpenMP Example

```
#pragma omp parallel sections
{
    // Producer
#pragma omp section
{
    // produce data
    flag = 1;
}
    // Consumer
#pragma omp section
{
    while (flag == 0 ) {}
    // consume data
}
}
```

```
#pragma omp parallel sections
{
#pragma omp section
{
    // produce data
#pragma omp flush
#pragma omp write
    flag = 1;
#pragma omp flush(flag)
}
#pragma omp section
{
    while (1) {
#pragma omp flush(flag)
#pragma omp atomic read
        flag_read = flag
        if (flag_read) break;
    }
#pragma omp flush
    // consume data
}
}
```

OpenMP Optimizing Compiler

- Can reorder operations freely inside a parallel region
 - No guarantees about the ordering of operations during a parallel region excepting around flush operations
 - Parallel region contains implicit flushes
 - Cannot move operations outside of the parallel region or around synchronization operations
 - Presence of flush operations make the OpenMP memory model a variant of weak ordering

More Rules

- If the intersection of the flush-sets of two flushes performed by two different threads is non-empty, then the two flushes must be completed as if in some sequential order, seen by all threads
- If the intersection of the flush-sets of two flushes performed by one thread is non-empty, then the two flushes must appear to be completed in that thread's program order
- If the intersection of the flush-sets of two flushes is empty, then the threads can observe these flushes in any order

References

- Tim Mattson et al. The OpenMP Common Core: A hands on exploration. SC 2018.
- Ruud van der Pas. OpenMP Tasking Explained. SC 2013.
- Blaise Barney. OpenMP. <https://computing.llnl.gov/tutorials/openMP/>
- C. Terboven and M. Klemm. Advanced OpenMP Tutorial. OpenMPCon & IWOMP 2017.