ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Final Project Related Issues
Variable Sharing in OpenMP
OpenMP synchronization issues
OpenMP performance issues

November 6, 2015
Lecture 23

Quote of the Day

"I have never let my schooling interfere with my education."

-- Mark Twain, 1835 - 1910

Before We Get Started

- Issues covered last time:
  - Data sharing in OpenMP, wrap up
  - OpenMP synchronization issues
  - OpenMP optimization issues
- Today's topics
  - Final Project related issues
  - Open MP optimization issues, wrap up
  - MPI, introduction
- Other issues:
  - HW08, due on Wd, Nov. 10 at 11:59 PM
  - Dan/Ang/Hammad to update the starting point to eliminate references to deprecated functionality

Final Project Related

- Proposal Issues:
  - Two pages long (shorter, if it makes sense)
  - PDF file
  - Due on 11/13 at 11:59 pm (Learn@UW dropbox)
  - Structure:
    - Statement of the problem
    - One paragraph explanation of why you chose the problem (motivation)
    - Goal of project (what will be accomplished)
    - How you want to go about it
    - How you'll demonstrate that you reached your proposed goal
    - Management issues
      - Who the team members are, and who does what
      - Project timeline (for "reality check" purposes)
- I will try to provide feedback on all proposals within seven days

Final Project Issues

- Project can be individual or team-based
  - Teams can have up to four students
    - Multi-student proposals: need to spell out who does what
- Final Project Presentation
  - 66 students
  - Ten minute per student
  - Example: team of 3 students gets 30 mins for final presentation
- Presentations scheduled through doodle, Dan to look into this
- Final Project presentation window: Wednesday 7 AM through Friday 7 PM (Dec. 16 through 18, Finals week)

Three Default Projects

- Project 1:
  - Solve two types of systems on multiple GPUs:
    - Banded lower triangular linear system
    - Banded upper triangular linear systems
      - Size of system: up to tens of millions
      - Size of band: up to 1000
- Project 2:
  - Charm++ multi-node parallel implementation of granular dynamics
- Project 3:
  - Benchmarking for performance database

Project 1

- Solve AX=B, where

$$\mathbf{A} = \begin{bmatrix} 1 &&&&&&&&& \\ 5 & 1 &&&&&&&& \\ 6 & 2 & 1 &&&&&&& \\ 4 & 7 & 1 & 1 &&&&&& \\ & 2 & 5 & 9 & 1 &&&&& \\ && 6 & 1 & 6 & 1 &&&& \\ &&& 7 & 8 & 2 & 1 &&& \\ &&&& 3 & 0 & 7 & 1 && \\ &&&&& 9 & 7 & 0 & 1 & \\ &&&&&& 8 & 3 & 2 & 1 \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} x_{11} & & x_{1,3} \\ \vdots & \vdots & \vdots \\ x_{10,1} & & x_{10,3} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -2 & 12 & 0 \\ 3 & 3 & -6 \\ 5 & 9 & 7 \\ 7 & 3 & 2 \\ 2 & 7 & 8 \\ 4 & 11 & 3 \\ 10 & 3 & 22 \\ 22 & 9 & 61 \\ -9 & 7 & 19 \\ 0 & 6 & 33 \end{bmatrix}$$

- In this example, N=10, K=2, and M=3
  - N – dimension of matrix A (tens of millions)
  - K – width of bandwidth (up to 1000)
  - M – number of RHS vectors (up to 1000)
- Note: example shows band lower triangular case – the band upper triangular case needs to be solved as well

Project 2: Granular Dynamics

- Uses Charm++
- Example: granular material dynamics
- Test problem: filling up bucket with 100 million spheres
- Bodies are spheres

- Challenging, on three accounts:
  - Charm++ is not straightforward, steep learning curve
  - Support on Euler is limited
  - Handling the dynamics of the problem not trivial

Project 3: Benchmarking Project

- Build on 2025 MS Thesis of ECE student who took ME759

- Basic idea behind this project:
  - Pick up a problem of interest and find out how much it takes to solve the problem using various HW and/or SW solutions

  - Example Problem: perform “scan” operation fast
    - Step 1: Use MPI, Charm++, OpenMP, OpenCL, straight CUDA, thrust, cub, etc. to find out how effectively various HW and/or SW choices implement scan
    - Step 2: Refine existing PerfDB that Omkar started
    - Step 3: Improve its web interface for other people to be able to query this PerfDB

The “Non-default Project” Option

- Do something tied to your research or something that you are interested in/curious about
- OK to propose something that is a small part of big undertaking that would be too large to accomplish within one month
  - This is fine as long as
    - Project proposed represents a step towards something that is ambitious
    - You structure the Final Project so that progress can be measured/demonstrated
- IMPORTANT: Your score will reflect how well you accomplish what’s spelled out in Project Proposal, not the big task that can’t be done in one month

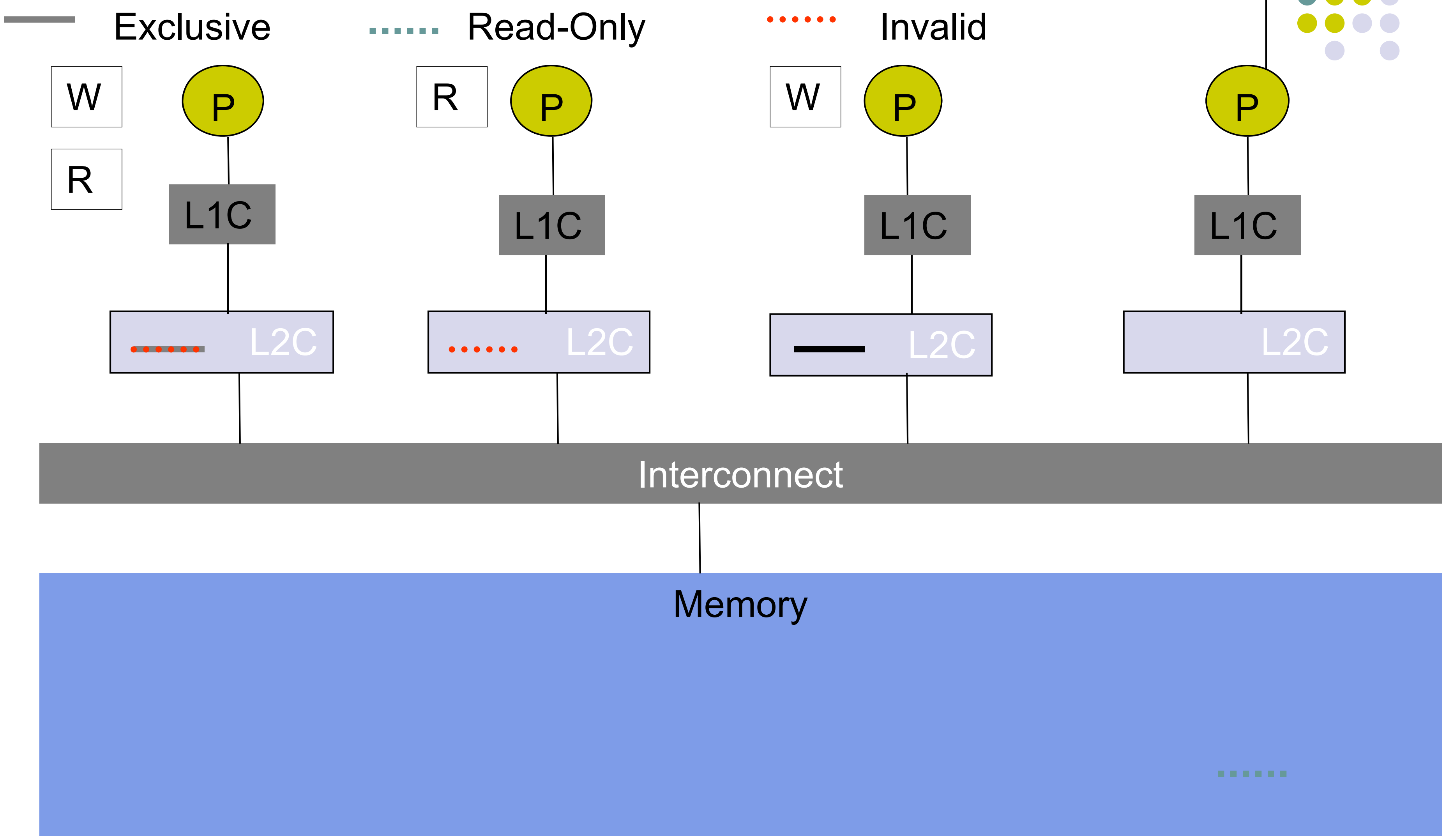
Loose Ends

- I hope to write two conference papers with students who come up with the best Project 1 and Project 3
  - Would require some extra work, after conclusion of the semester
- Project 2 involved, requires more effort before ready for prime time
- Be ambitious, yet propose something manageable
  - Failing is totally ok, if it comes despite hard work
- Most common issue: people propose things without realizing that they have other tasks to work on and can't allocate enough time
  - Remember that this needs to be wrapped up in one month. Recall that an assignment takes about one week to complete

Final Project Deliverables

- Three items need to be delivered:
  - Final Project report (PDF document)
    - Has two parts:
      - "Part 1": your Final Project Proposal that is due on November 13
      - "Part 2": summarizes the work done and results obtained in conjunction with the proposed work in "Part 1"
    - Due no later than Dec. 21 at 11:59 PM
  - PowerPoint doc sent to Dan at least 6 hours prior to making your presentation
  - Code that can be used to verify results reported in Final Project
    - Due no later than Dec. 21 at 11:59 PM

Back to Usual Program: OpenMP Code Optimization Aspects



Coherency example

Credit: Alan Real

False sharing

- Cache lines consist of several words of data
  - It’s common for one cache line to store 8 double precision values
- What happens when two processors are both writing to different words on the same cache line?
  - Each write will invalidate the other processors copy
  - Lots of remote memory accesses
- Symptoms:
  - Poor execution time
  - High, non-deterministic numbers of cache misses
  - Mild, non-deterministic, unexpected load imbalance

Credit: Alan Real

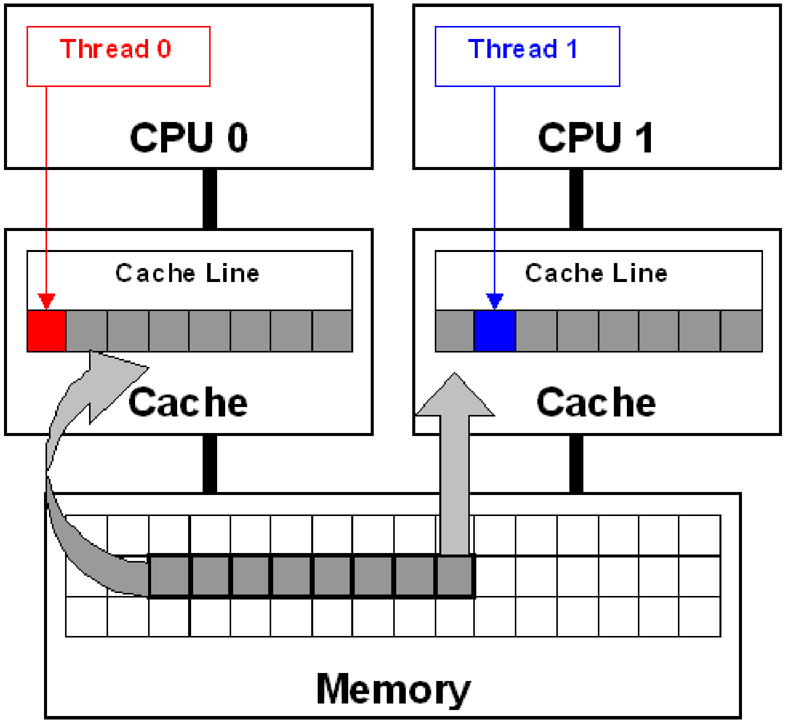
False Sharing Example

Assume NUM_THREADS is 8
Assume N is 16000

```
double sum = 0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;

#pragma omp for
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];

#pragma omp atomic
    sum += sum_local[me];
}
```



Think about this: could you fix this with a “firstprivate”?

Credit: Intel

Sometimes This Fixes It

- Reduce the frequency of false sharing by using thread-local copies of data.
  - The thread-local copy is read and modified frequently
  - When complete, copy the result back to the data structure.

```
struct ThreadParams
{
    // struct encapsulates info required by thread to figure out its work order
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; //Subject to frequent read/write access variable
    unsigned long start;
    unsigned long end;
};

void threadFunc(void *parameter)
{
    ThreadParams *p = (ThreadParams*)parameter;
    // local copy for read/write access variable
    unsigned long local_v = p->v;

    for (unsigned long local_dummy = p->start; local_dummy < p->end; local_dummy++)
    {
        // Functional computation, read/write the “v” member.
        // Keep reading/writing local_v instead
    }

    p->v = local_v; // Update shared data structure only once
}
```


[Intel]→

Another Way to Fix This
[Ugly + Architecture Dependent]

- When using an array of data structures, pad the structure to the end of a cache line to ensure that the array elements begin on a cache line boundary.
  - If you cannot ensure that the array is aligned on a cache line boundary, pad the data structure to twice the size of a cache line.

```
struct ThreadParams
{
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; // Frequent read/write access variable
    unsigned long start;
    unsigned long end;

    // expand to 64 bytes to avoid false-sharing
    // (4 unsigned long variables + 12 padding)*4 = 64
    int padding[12];
};

__declspec (align(64)) struct ThreadParams Array[10];
```

Credit: Intel

Concluding Remarks on the OpenMP API

Attractive Features of OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Code size grows only modestly
- Expression of parallelism flows clearly, code is easy to read
- Single source code for OpenMP and non-OpenMP
  - Non-OpenMP compilers simply ignore OMP directives
- Cross-platform compatibility
  - Linux, Windows, OSX

[Rebecca Hartman-Baker]→

OpenMP, Some Caveats

- OpenMP threads are heavy
  - Good for handling parallel tasks
  - Not so good at handling fine large scale grain parallelism
  - The model embraced is not that of hardware oversubscription

- There is a lag between the moment a new specification is released and the time a compiler is capable of handling all of its aspects
  - Intel's compiler is probably most up to date
  - Visual Studio 2015 does not support OpenMP 3.0
    - No support for tasks, for instance

OpenMP Issues Not Discussed

- Two issues not discussed in ME759 but important if you want to squeeze everything out of OpenMP
  - Nested parallelism and OpenMP support
  - The SMP vs. NUMA model and thread affinity implications

Further Reading, OpenMP

- Michael Quinn (2003) Parallel Programming in C with MPI and OpenMP
- Chapman, Barbara, Gabrielle Jost, and Ruud van der Pas. (2008) Using OpenMP, Cambridge, MA: MIT Press.
- Kendall, Ricky A. (2007) Threads R Us, http://www.nccs.gov/wp-content/training/scaling_workshop_pdfs/threadsRus.pdf
- Mattson, Tim, and Larry Meadows (2008) SC08 OpenMP "Hands-On" Tutorial, http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
- LLNL OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/
- OpenMP.org, http://openmp.org/
- OpenMP 3.0 API Summary Cards:
  - Fortran: http://openmp.org/mp-documents/OpenMP-4.0-Fortran.pdf
  - C/C++: http://openmp.org/mp-documents/OpenMP-4.0-C.pdf
- http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Parallelism, as Expressed at Various Levels

Cluster	Group of computers communicating through fast interconnect
Coprocessors/Accelerators	Special compute devices attached to the local node through special interconnect
Node	Group of processors communicating through shared memory
Socket	Group of cores communicating through shared cache
Core	Group of functional units communicating through registers
Hyper-Threads	Group of thread contexts sharing functional units
Superscalar	Group of instructions sharing functional units
Pipeline	Sequence of instructions sharing functional units
Vector	Single instruction using multiple functional units

Have discussed already →

Haven't discussed yet →

Have discussed, but little direct control →

[Intel]→

Instruction Set Architecture (ISA) Extensions

- Extensions to the base x86 ISA: One way that x86 has evolved over the years
  - Extensions for vectorizing math
    - SSE, AVX, SVML, IMCI
    - F16C - half precision floating point (called FP16 in CUDA)
  - Hardware Encryption/Security extensions
    - AES, SHA, MPX
  - Multithreaded Extensions
    - Transactional Synchronization Extensions - TSX (Intel)
    - Advanced Synchronization Facility - ASF (AMD)

[Hammad]→

CPU SIMD

- We have some “fat” registers for math
  - 128 bit wide, 256 bit wide, 512 bit wide
- Pack floating point values into these registers
  - 4 floats or 2 doubles in a single 128 bit register
- Perform math on these registers
  - Ex: One add instructions can add 4 floats
- This concept is known as “vectorizing” your code

[Hammad]→

CPU SIMD via Vectorization

- Comes in many flavors
- Most CPUs support 128 bit wide vectorization
  - SSE, SSE2, SSE3, SSE4
- Newer CPUs support AVX
  - 128 bit wide and some 256 bit wide instructions
- Haswell supports AVX2
  - full set of 256 bit wide instructions
- Skylake, Xeon Phi 2nd Gen, will support AVX-512
  - 512 bit wide instructions

[Hammad]→

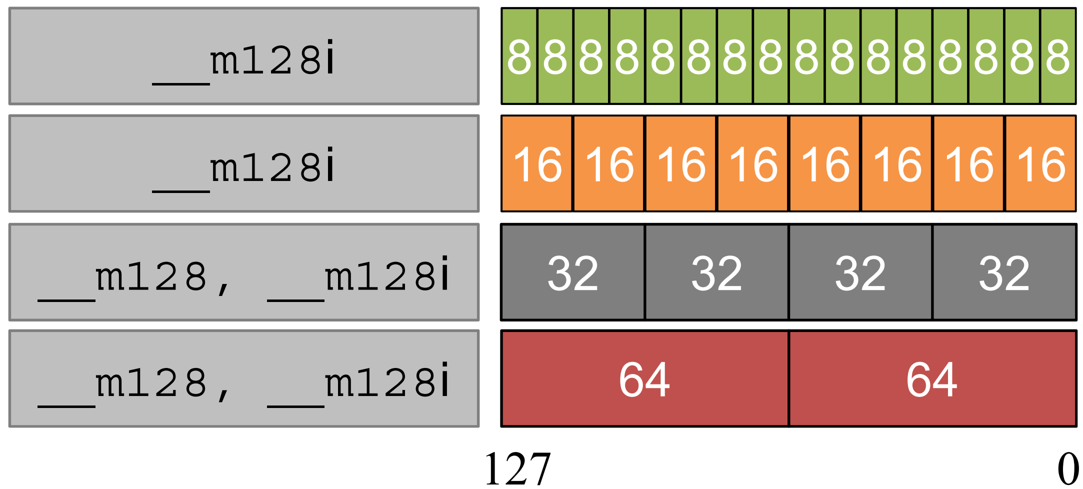
Streaming SIMD Extensions (SSE)

- Implemented using a set of 8 new 128 bit wide registers
  - Called: xmm0, xmm1,..., xmm7
  - SSE operations can only use these registers
- SSE supported storing 4 floats in each register
  - Basic load/store and math
- SSE2 expanded that to 2 `doubles`, 8 short `integers` or 16 `chars`
  - SSE2 implements operations found in MMX spec
- SSE3
  - Horizontal operations
- SSE4
  - Lots of new instructions like Dot Product, Min, Max, etc.

[Hammad]→

An Introduction to SSE

- New types: `__m128` (**float**), `__m128i` (**int**)
- Constructing:



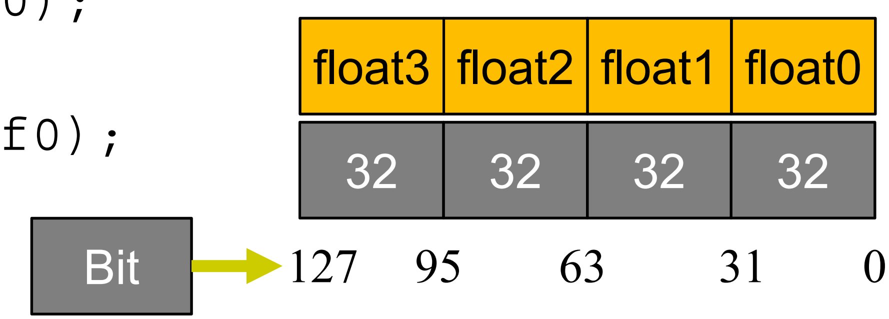

```
__m128 m =
_mm_set_ps(f3,f2,f1,f0);
_mm_set_pd(d1,d0);

__m128i mi =
_mm_set_epi64(e1, e0) //2 64 bit ints
_mm_set_epi32(e3,e2,e1,e0) //4 32 bit ints
_mm_set_epi16(e7,e6,e5,e4,e3,e2,e1,e0) //8 16 bit shorts
_mm_set_epi8 (e15,e14,e13,e12,e11,e10,e9 ,e8,
                e7 ,e6 ,e5 ,e4 ,e3 ,e2 ,e1 ,e0) //16 chars
```


[Hammad]→

An Introduction to SSE

- Intrinsics ending with
  - `ps` operate on single precision values
  - `pd` operate on double precision values
  - `i8` operate on chars
  - `i16` operate on shorts
  - `i32` operate on 32 bit integers
  - `i64` operate on 64 bit integers
- Conventions
  - Bits are specified from 0 at the right to the highest value at the left
- Note the order in set functions
  - `_mm_set_ps(f3,f2,f1,f0);`
- For reverse order use
  - `_mm_setr_ps(f3,f2,f1,f0);`



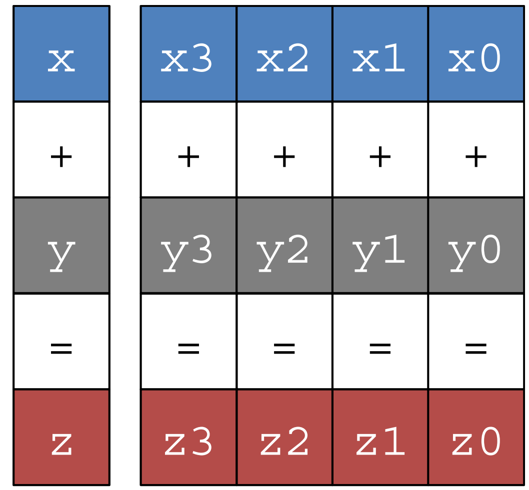
[Hammad]→

4 wide add operation (SSE 1.0)

C++ code

```
__m128 Add (const __m128 &x, const __m128 &y){
       return _mm_add_ps(x, y);
}

__mm128 z, x, y;
x = _mm_set_ps(1.0f,2.0f,3.0f,4.0f);
y = _mm_set_ps(4.0f,3.0f,2.0f,1.0f);
z = Add(x,y);
```



“gcc –S –O3 sse_example.cpp”

Assembly

```
    __Z10AddRKDv4_fS1_ __Z10AddRKDv4_fS1_:
        movaps  (%rsi), %xmm0   # move y into SSE register xmm0
        addps   (%rdi), %xmm0   # add x with y and store xmm0
    ret                         # xmm0 is returned as result
```


[Hammad]→

SSE Dot Product (SSE 4.1)

```
_m128 r = _mm_dp_ps (__m128 x, __m128 y, int mask)
```

- Dot product on 4 wide register

r	r3	r2	r1	r0	x	x3	x2	x1	x0	y	y3	y2	y1	y0

Bit #	7	6	5	4	3	2	1	0
mask	1	1	1	1	0	0	0	1

- Use mask to specify what entries are added
  - Bits 4-7 specify what entries are multiplied
  - Bits 0-3 specify where sum is stored
  - In this case: multiply all 4 entries in x and y and add them together. Store result in r1.

Normalize 4 wide Vector

C++ code

```
__m128 Normalize( const __m128 &x){
         const int mask = 0b11110001;
         return _mm_sqrt_ps(_mm_dp_ps(x, x, mask));
}
__mm128 z, x;
x = _mm_set_ps(1.0f,2.0f,3.0f,4.0f);
z = Normalize(x);
```

“gcc –S –O3 sse_example.cpp”

Assembly

```
__Z9NormalizeRKDv4_f __Z9NormalizeRKDv4_f:
    movaps  (%rdi), %xmm0 # load x into SSE register xmm0
    # perform dot product, store result into first 32 bits of xmm0
    dpps    $241, %xmm0, %xmm0
    sqrtps  %xmm0, %xmm0 # perform sqrt on xmm0, only first 32 bits contain data
    ret                  # return xmm0
```


[Hammad]→

Intrinsics vs. Assembly

- Intrinsics map c/c++ code onto x86 assembly instructions
  - Some intrinsics map to multiple instructions
- Consequence: it's effectively writing assembly code in C++
  - Without dealing with verbosity of assembly
    - `In c++ _mm_add_ps becomes addps`
    - `In c++ _mm_dp_ps becomes  dpps`
- `It's always possible to write c++ code that generates optimal assembly`

[Hammad]→

Types of SSE/AVX operations

- Data movement instructions
  - Unaligned, Aligned, and Cached loads
- Arithmetic instructions
  - Add, subtract, multiply, divide, …
- Reciprocal instructions
  - `1.0/x, 1.0/sqrt(x)`
- Comparison
  - Less than, greater than, equal to, …
- Logical
  - `and, or, xor,` …
- Shuffle
  - Reorder packed data

[Hammad]→

Memory operations

- Load one cache line from system memory into cache
  - `void _mm_prefetch(char * p , int i );`
- Uncached load (does not pollute cache)
  - `_mm_stream_ps(float * p , __m128 a );`
- Aligned load and store
  - `__m128 _mm_load_ps (float const* mem_addr)`
  - `void _mm_store_ps (float* mem_addr, __m128 a)`
- Unaligned load and store
  - `__m128 _mm_loadu_ps (float const* mem_addr)`
  - `void _mm_storeu_ps (float* mem_addr, __m128 a)`

[Hammad]→

Shuffle Operations

- Move/reorganize data between two __m128 values

```
_mm_shuffle_ps(__m128 x, __m128 y, int mask)
```

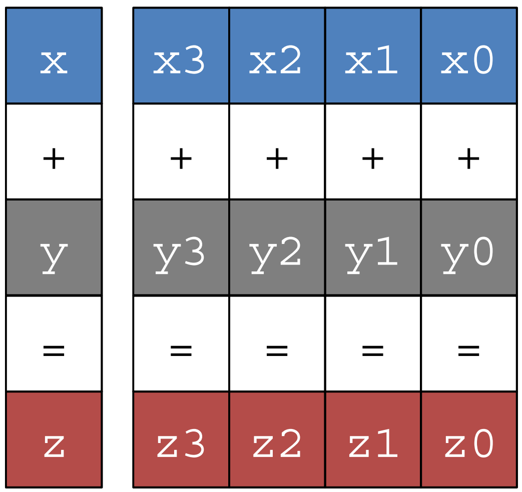
- Every two bits in mask represent one output entry
  - Bits 0-3 represent first 4 bits represent 2 entries from x
  - Bits 4-7 represent 2 entries from y

Bit #	7	6	5	4	3	2	1	0
entry	3		2		1		0	
example	0	0	0	0	0	0	0	0
example	1	0	0	1	0	1	0	1
example	1	1	0	0	0	0	1	0

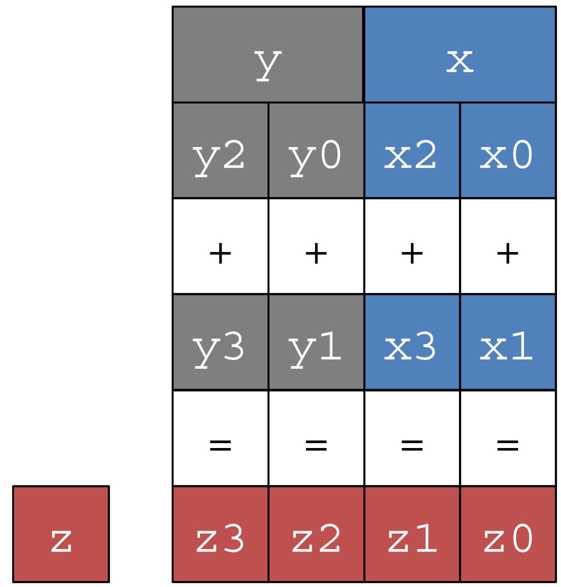
x	x3	x2	x1	x0
y	y3	y2	y1	y0
r	y0	y0	x0	x0
r	y2	y1	x1	x1
r	y3	y0	x0	x2

[Hammad]→

Horizontal Operators (SSE 3)



Traditional Add



Horizontal Add

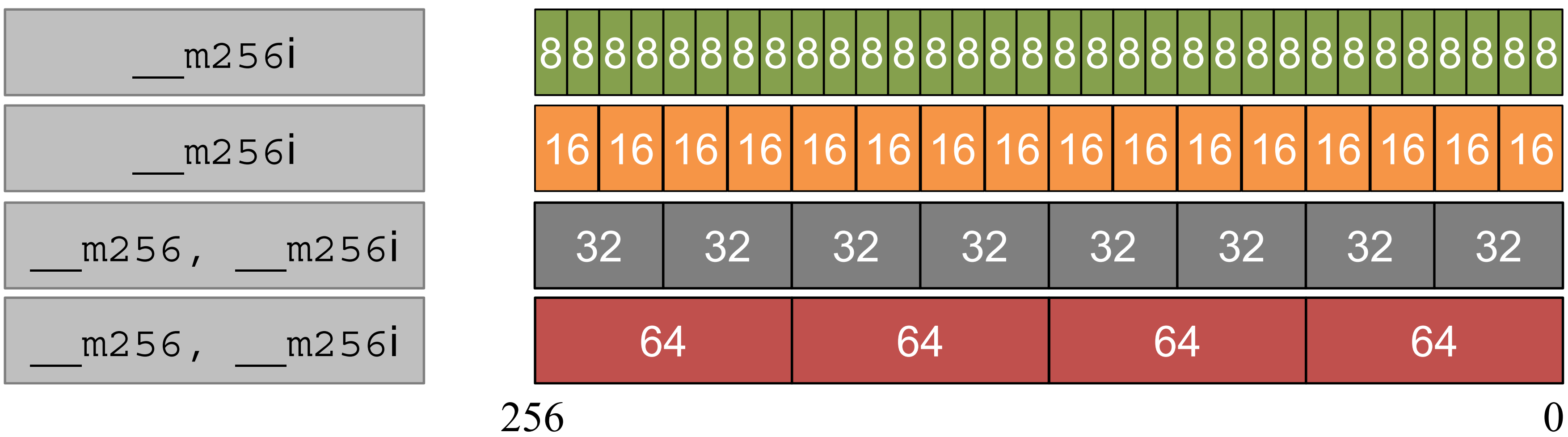
- Horizontal operators for addition, subtraction
  - 32 and 64 bit floating point values
  - 8, 16, 32, 64 bit integers

- Used, for example, in small matrix-matrix multiplication

[Hammad]→

Advanced Vector Extensions AVX

- Similar to SSE but wider, 32 registers, each 256 bit wide
  - SSE has 8 128 bit registers



Examples:

- Add operation

```
__m256 _mm256_add_ps (__m256 a, __m256 b)
```

- Dot product

```
__m256 _mm256_dp_ps (__m256 a, __m256 b, const int imm8)
```

[Hammad]→

Header File Reference

- `#include<mmintrin.h>   //MMX`
- `#include<xmmintrin.h>  //SSE`
- `#include<emmintrin.h>  //SSE2`
- `#include<pmmintrin.h>  //SSE3`
- `#include<tmmintrin.h>  //SSSE3`
- `#include<smmintrin.h>  //SSE4.1`
- `#include<nmmintrin.h>  //SSE4.2`
- `#include<immintrin.h>  //AVX`

[Hammad]→

History

- MMX (1996) – First Widely Adopted standard
- 3DNow (1998) – Used for 3D graphics processing on CPUs
- SSE (1999) – Designed by Intel, initially used by Intel only
- SSE2 (2001) – AMD jumps in at this point, adds support to their chips
- SSE3(2004)
- SSSE3 (2006) – Supplemental SSE3 instructions
- SSE4 (2006)
- SSE5 (2007) – Introduced by AMD but dropped in favor of AVX
  - Split SSE5 into-> XOP, CLMUL, FMA extensions
- AVX (2008) - Introduced by Intel with Sandy Bridge (AMD supports)
- AVX2 (2012) – Introduced by Intel with Haswell
- AVX-512 (~2016) - Skylake

[Hammad]→

Resources

- Excellent guide covering all SSE/AVX intrinsics
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide/#
- SSE Example code
  - http://www.tommesani.com/index.php/simd/42-mmx-examples.html
- Assembly analysis of SSE optimization
  - http://www.intel.in/content/dam/www/public/us/en/documents/white-papers/ia-32-64-assembly-lang-paper.pdf

[Hammad]→

Parallel Computing as Supported by MPI

Acknowledgments

- Parts of MPI material covered draws on a set of slides made available by the Irish Centre for High-End Computing (ICHEC) - www.ichec.ie
  - These slides will contain "ICHEC" at the bottom
  - In turn, the ICHEC material was based on the MPI course developed by Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh

- Individual or institutions are acknowledged at the bottom of the slide, like

[A. Jacobs]→

MPI: Textbooks, Further Reading…

- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18,1997)
- **MPI: The Complete Reference**, Marc Snir and William Gropp et al., The MIT Press, 1998 (2-volume set)
- **Using MPI: Portable Parallel Programming With the Message-Passing Interface** and **Using MPI-2: Advanced Features of the Message-Passing Interface.** William Gropp, Ewing Lusk and Rajeev Thakur, MIT Press, 1999 – also available in a single volume ISBN 026257134X.
- **Parallel Programming with MPI**, Peter S. Pacheco, Morgan Kaufmann Publishers, 1997 - very good introduction.
- **Parallel Programming with MPI**, Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti. Training handbook from EPCC
  - http://www.epcc.ed.ac.uk/computing/training/document_archive/mpi-course/mpi-course.pdf

Shared Memory Systems

- Memory resources are shared among processors
  - Typical scenario, on a budget: one node with four CPUs, each with 16 cores
- Relatively easy to program since there is a single unified memory space
- Two issues:
  - Scales poorly with system size due to the need for cache coherence
  - Most often, you need more memory than available on the typical multi-core node
- Example:
  - Symmetric Multi-Processors (SMP)
    - Each processor has equal access to RAM
- Traditionally, this represents the hardware setup that supports OpenMP-enabled parallel computing

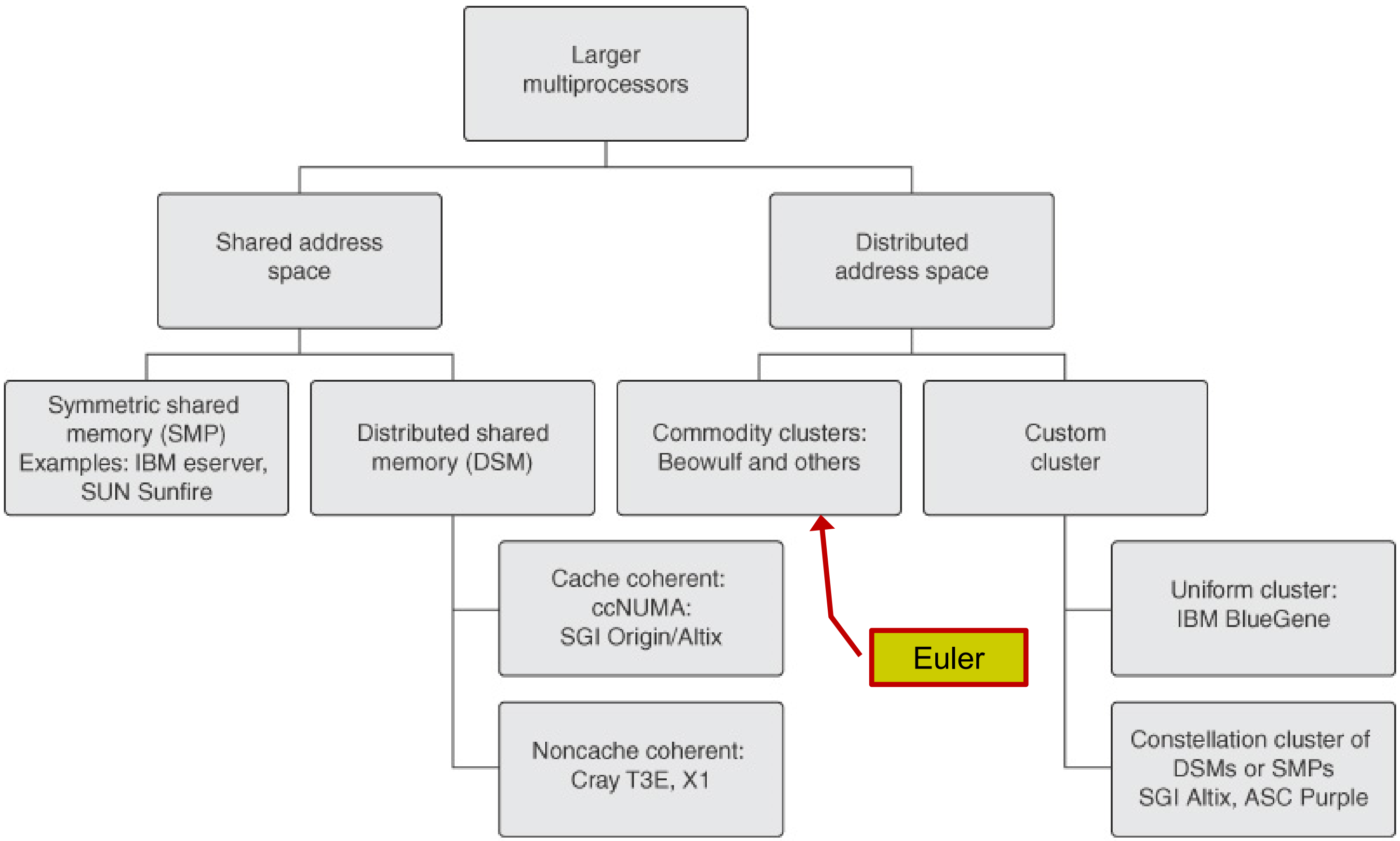
[A. Jacobs]→

Distributed Memory Systems

- Individual nodes consist of a CPU, RAM, and a network interface
  - A hard disk is typically not necessary; mass storage can be supplied using NFS
- Information is passed between nodes using the network
- No cache coherence and no need for special cache coherency hardware
- Software development: more difficult to write programs for distributed memory systems since the programmer must keep track of memory usage
- Traditionally, this represents the hardware setup that supports MPI-enabled parallel computing

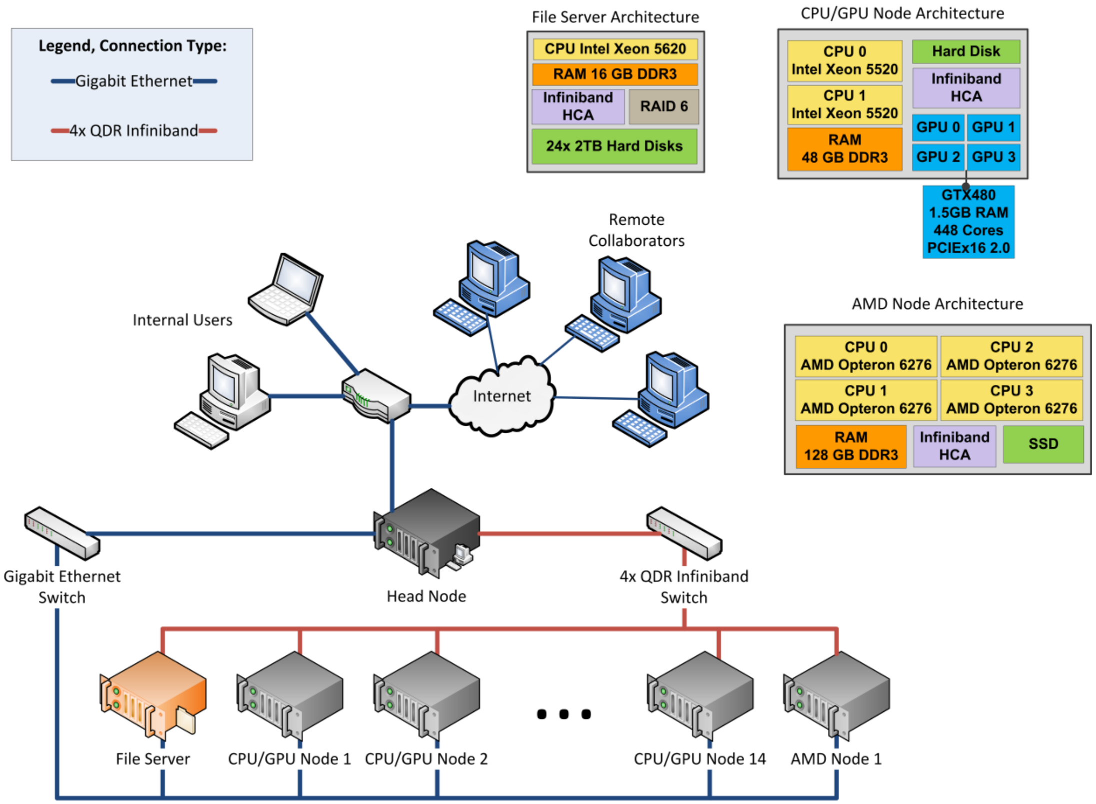
[A. Jacobs]→

Overview of Large Multiprocessor Hardware Configurations



Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition

Euler
~ Hardware Configurations ~



Hardware Relevant in the Context of MPI

Two Components of Euler that are Important

- **CPU**: AMD Opteron 6274 Interlagos 2.2GHz
  - 16-Core Processor (four CPUs per node → 64 cores/node)
  - 8 x 2MB L2 Cache per CPU
  - 2 x 8MB L3 Cache per CPU
  - Thermal Design Power (TDP): 115W

- **HCA**: 40Gbps Mellanox Infiniband interconnect
  - Bandwidth comparable to PCIe2.0 x16 (~32Gbps), yet the latency is rather poor (~1microsecond)
  - Ends up being the bottleneck in cluster computing

MPI: The 30,000 Feet Perspective

- The same program is launched for execution independently on a collection of cores
- Each core executes the program
- What differentiates processes is their rank: processes with different ranks do different things (“branching based on the process rank”)
  - Very similar to GPU computing, where one thread did work based on its thread index