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 9, 2015
Lecture 24

Quote of the Day

"Without music to decorate it, time is just a bunch of boring production deadlines or dates by which bills must be paid."

-- Frank Zappa, Musician

1940 - 1993

Before We Get Started

- Issues covered last time:
  - Final Project discussion
  - Open MP optimization issues, wrap up

- Today’s topics
  - SSE and AVX quick overview
  - Parallel computing w/ MPI

- Other issues:
  - HW08, due on Wd, Nov. 10 at 11:59 PM

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 the 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 or 256 bit wide or more recently 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 Support

- 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
  - Mature set of 256 bit wide instructions
- Skylake, Xeon Phi 2nd Gen, will support AVX-512
  - 512 bit wide instructions

[Hammad]→

Streaming SIMD Extensions (SSE) How Support Evolved Over Time

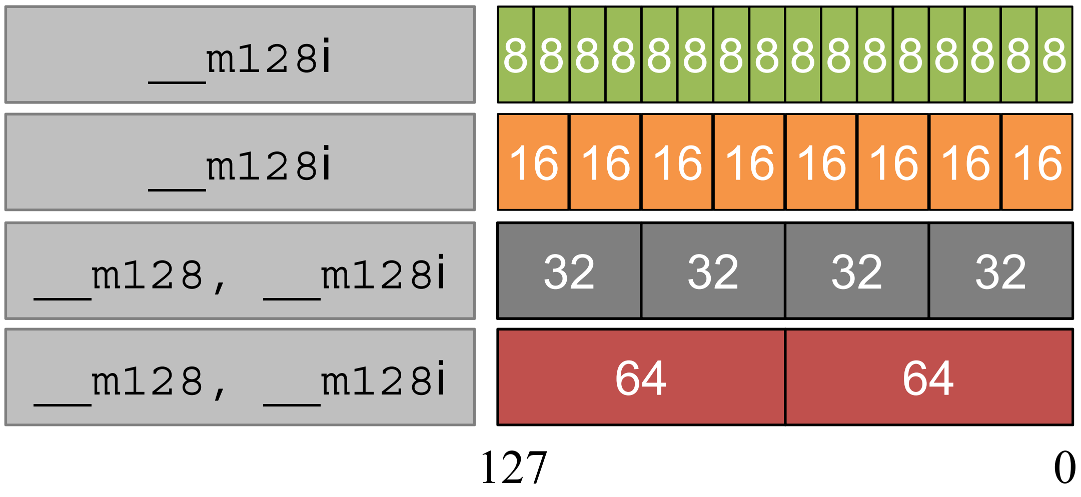
- Implemented using a set of 8 new 128 bit wide registers
  - Called: xmm0, xmm1,..., xmm7
  - SSE operations can only use these registers
- SSE supported storing 4 floats in each register
  - Basic load/store and math
- SSE2 expanded that to 2 `doubles`, 8 short `integers` or 16 `chars`
  - SSE2 implements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

[Hammad]→

SSE: Packing Data into Fat Registers

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

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



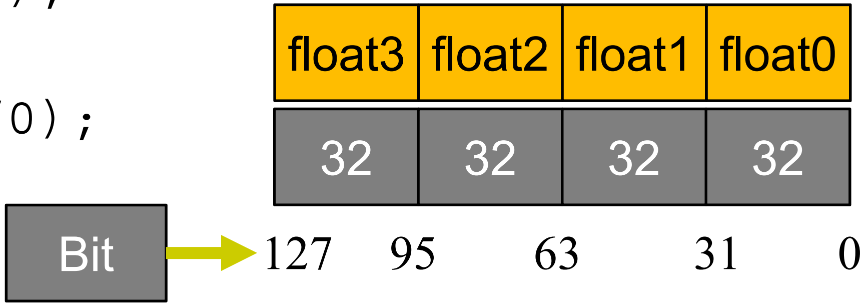
```
__m128i mi =
_mm_set_epi64(e1, e0) //2 64 bit ints
_mm_set_epi32(e3,e2,e1,e0) //4 32 bit ints
_mm_set_epi16(e7,e6,e5,e4,e3,e2,e1,e0) //8 16 bit shorts
_mm_set_epi8 (e15,e14,e13,e12,e11,e10,e9 ,e8,
                e7 ,e6 ,e5 ,e4 ,e3 ,e2 ,e1 ,e0) //16 chars
```

This “p” comes from “packed”

[Hammad]→

SSE: Nomenclature/Convention Issues

- 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);`



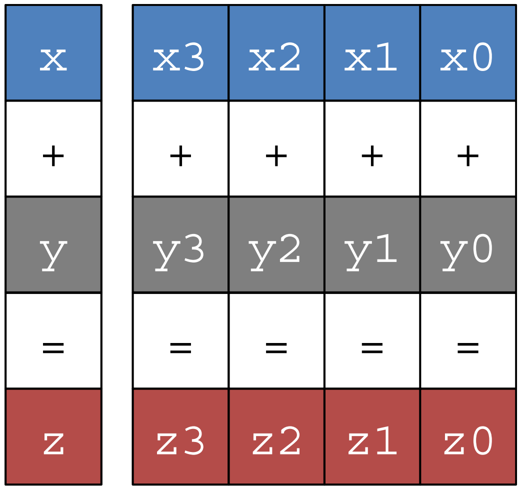
[Hammad]→

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

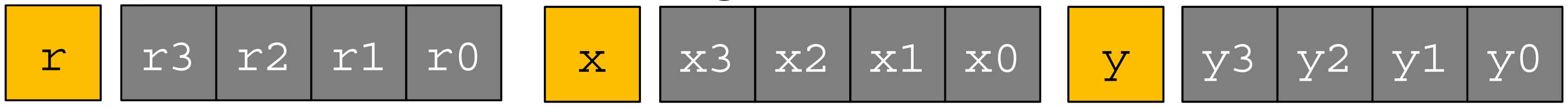
```
    __Z10AddRKDv4_fS1_ __Z10AddRKDv4_fS1_:
        movaps  (%rsi), %xmm0   # move y into SSE register xmm0
        addps   (%rdi), %xmm0   # add x with y and store xmm0
    ret                         # xmm0 is returned as result
```

[Hammad]→

SSE Dot Product (SSE 4.1)

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

- Dot product on 4 wide register



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

- Mask used 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`

- Convenience of writing C++ code that yet generates optimal assembly

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)
```

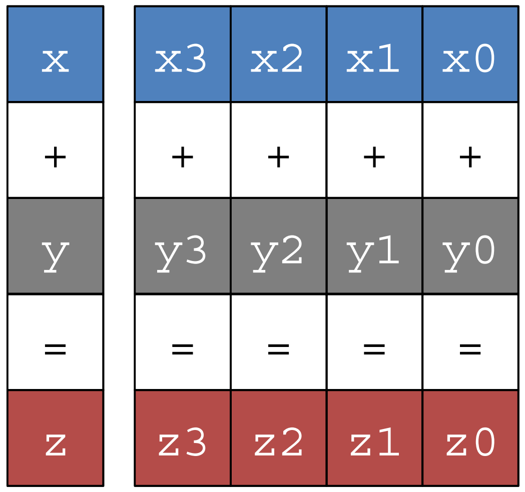
- Every two bits in mask represent one output entry
  - Bits 0-3 deal with two entries from x
  - Bits 4-7 deal with two entries from y

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

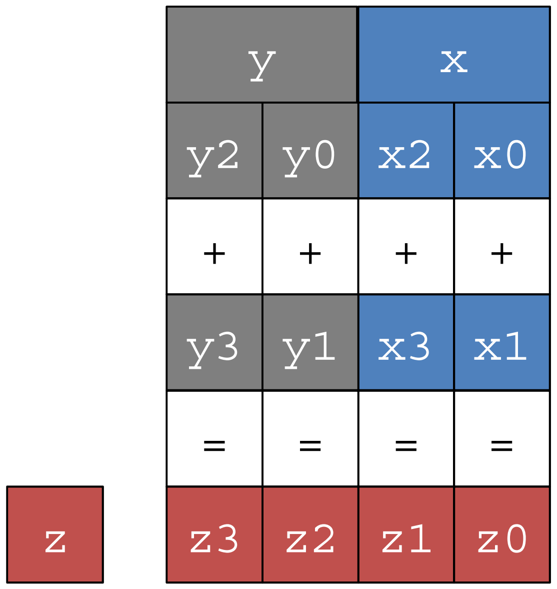
x	x3	x2	x1	x0
y	y3	y2	y1	y0
r	y0	y0	x0	x0
r	y2	y1	x1	x1
r	y3	y0	x0	x2

[Hammad]→

Horizontal Operators (SSE 3)



Traditional Add



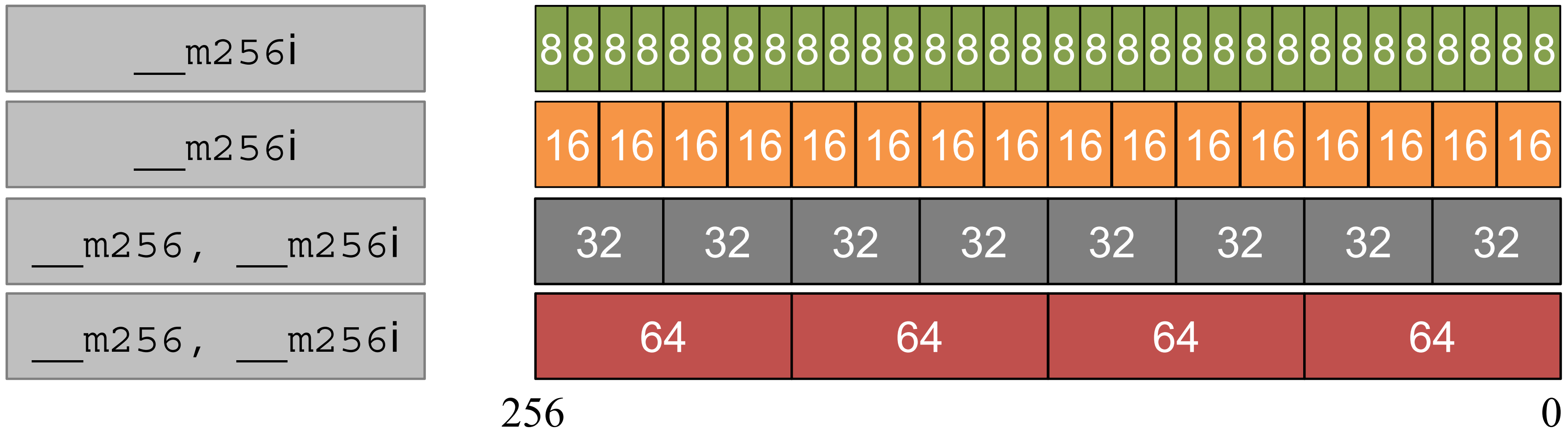
Horizontal Add

- 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 has 32 registers, each 256 bit wide
  - Note: SSE has eight 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

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
  - You might need more system memory than available on the typical multi-core node

- Example:
  - Symmetric Multi-Processors (SMP)
    - Each processor has equal access to RAM

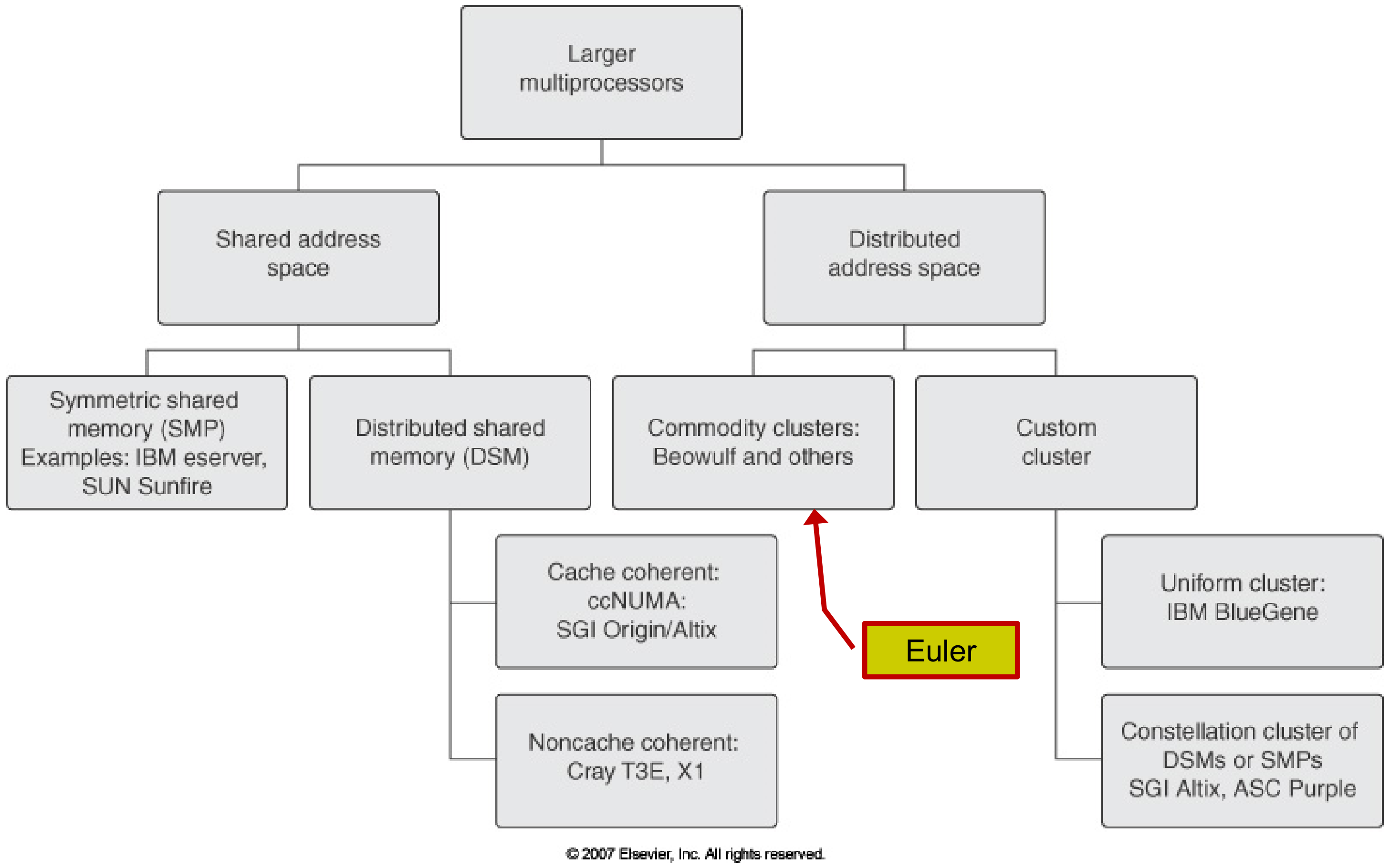
- Traditionally, this represents the hardware setup that supports OpenMP-enabled parallel computing

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]→

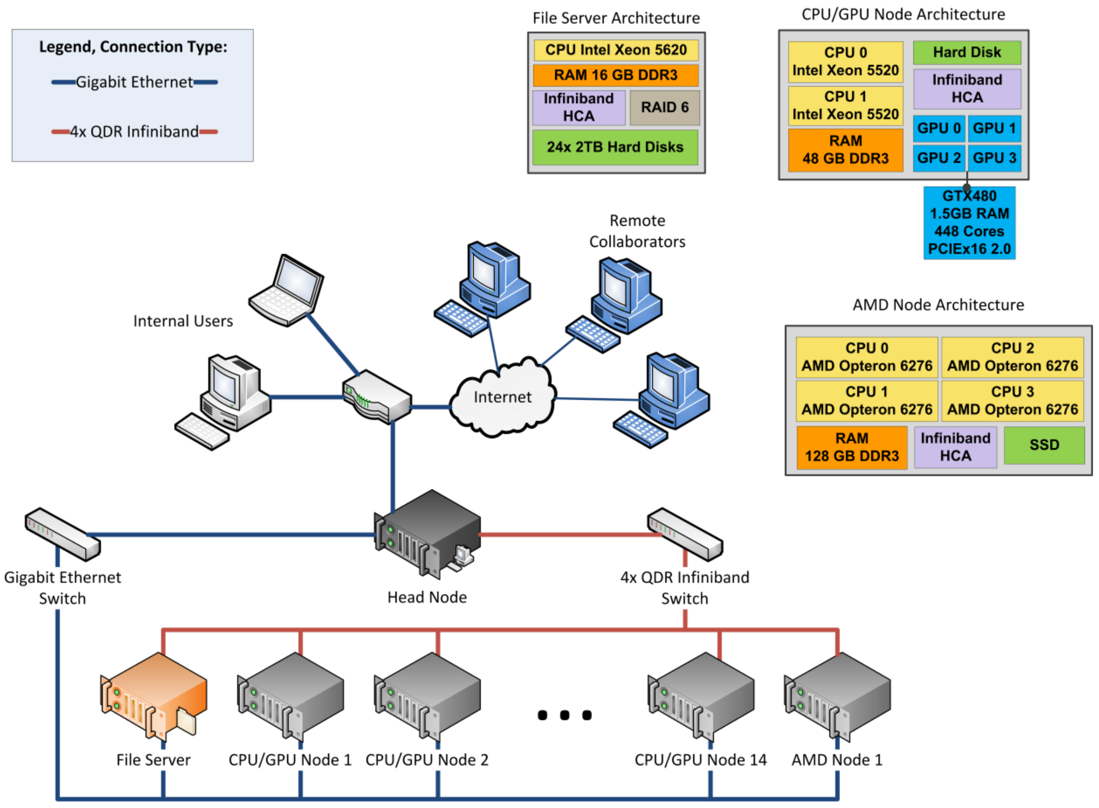
Overview of Large Multiprocessor Hardware Configurations



Larger multiprocessors
Shared address space
Distributed address space
Symmetric shared memory (SMP) Examples: IBM eserver, SUN Sunfire
Distributed shared memory (DSM)
Commodity clusters: Beowulf and others
Custom cluster
Cache coherent: ccNUMA: SGI Origin/Altix
Noncache coherent: Cray T3E, X1
Euler
Uniform cluster: IBM BlueGene
Constellation cluster of DSMs or SMPs SGI Altix, ASC Purple
© 2007 Elsevier, Inc. All rights reserved.

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

Euler
~ Hardware Configurations ~



Hardware Relevant in the <u>Context of MPI</u>

Two Components of Euler that are Important

- <u>**CPU**</u>: 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

- <u>**HCA**</u>: 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

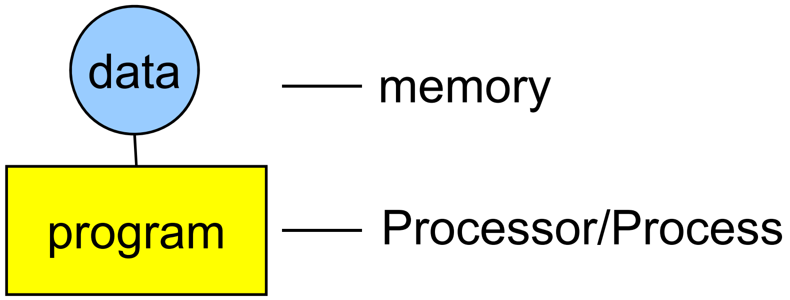
- What differentiates processes is their rank: processes with different ranks do different things ("branching based on the process rank")
  - Very similar to GPU computing, where one thread did work based on its thread index
  - Somewhat similar to OpenMP yet there the form of parallel computing promoted first and foremost drew on "work sharing": parallel for, parallel sections, parallel tasks
    - You can still have parallelism based on the id of the thread (not advertised that much)

The Message-Passing Model

- One starts many process on different cores but on each core the process is spawned by launching the same program
  - Process definition [in ME759]: “program counter + address space”
- Message passing enables communication among processes that have separate address spaces
- Inter-process communication calls for:
  - Synchronization, followed by…
  - … movement of data from one process’s address space to the others
- Execution paradigm embraced in MPI: Single Program Multiple Data (SPMD)

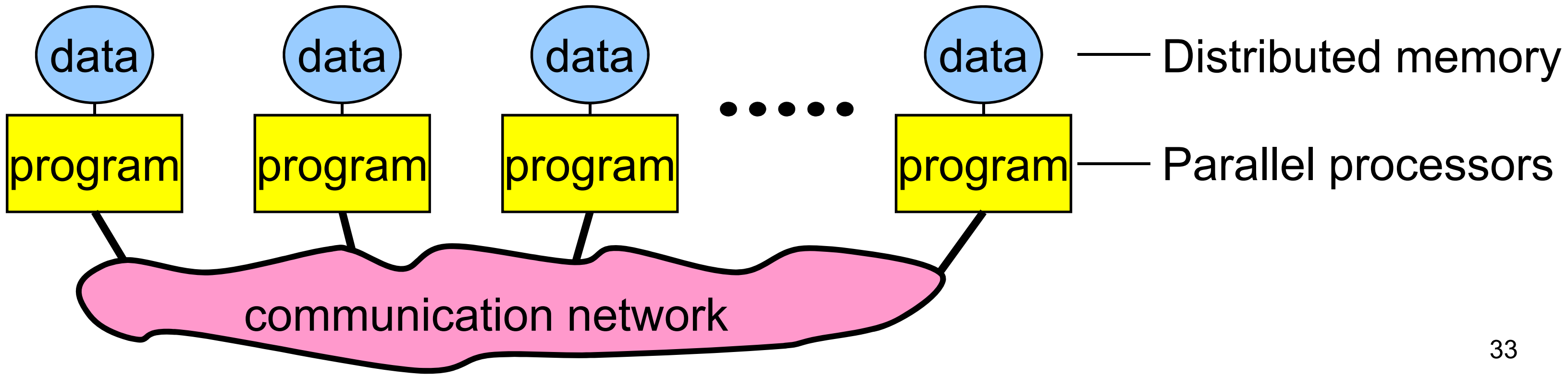
The Message-Passing Programming Paradigm

- Sequential Programming Paradigm



A processor may
run many processes

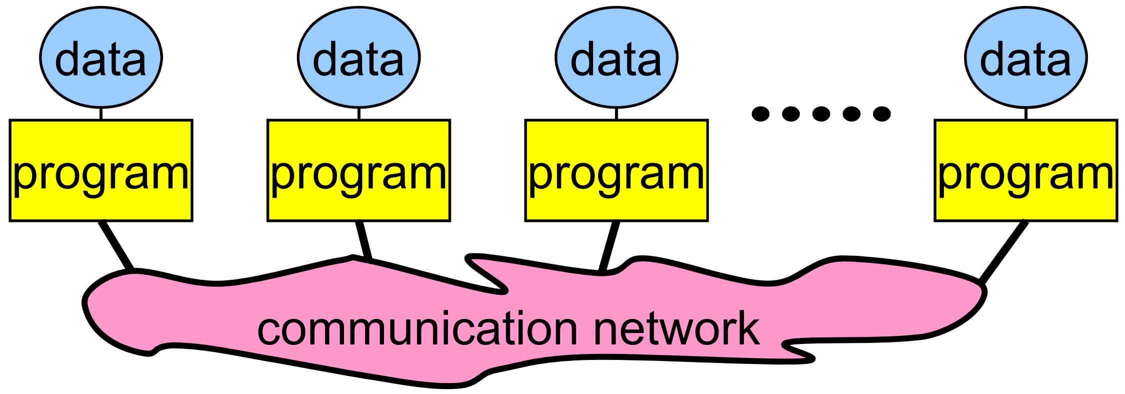
- Message-Passing Programming Paradigm



[ICHEC]→

Fundamental Concepts: Process/Processor/Program

- Our View: A **process** is a **program** performing a task on a **processor**
- Each processor/process in a message passing program runs a instance/copy of a **program:**
  - Written in a conventional sequential language, e.g., C or Fortran,
  - The variables of each sub-program have the same name but different locations (distributed memory) and different data!
  - Communicate via special send & receive routines (**message passing**)



[ICHEC]→

A First MPI Program

```
#include "mpi.h"
#include <iostream>


int main(int argc, char **argv) {
  int my_rank, n;
  char hostname[128];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &n);

  gethostname(hostname, 128);
  if (my_rank == 0) { /* master */
    printf("I am the master: %s\n", hostname);
  }
  else { /* worker */
    printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
  }

  MPI_Finalize();
  return 0;
}
```

Has to be called first, and once

Has to be called last, and once

[A. Snavely]→

Program Output

```
[negrut@euler04 CodeBits]$ mpiexec -np 8  ./a.out
I am a worker: euler04 (rank=1/7)
I am a worker: euler04 (rank=5/7)
I am a worker: euler04 (rank=6/7)
I am a worker: euler04 (rank=3/7)
I am a worker: euler04 (rank=4/7)
I am the master: euler04
I am a worker: euler04 (rank=2/7)
I am a worker: euler04 (rank=7/7)
[negrut@euler04 CodeBits]$
[negrut@euler04 CodeBits]$
```

Why Care about MPI?

- Today, MPI is what enables supercomputers to run at PFlops rates
  - Some of these supercomputers might use GPU acceleration though
- Examples of architectures relying on MPI for HPC:
  - IBM Blue Gene L/P/Q (Argonne National Lab – “Mira”)
  - Cray supercomputers (Oakridge National Lab – “Titan”, also uses K20X GPUs)
- MPI has FORTRAN, C, and C++ bindings
- MPI widely used in Scientific Computing
  - Department of Energy is an MPI champion

MPI is a Standard

- MPI is an API for parallel programming on distributed memory systems. Specifies a set of operations, but says nothing about the implementation
  - MPI is a standard

- Popular because it many vendors support (implemented) it, therefore code that implements MPI-based parallelism is very portable

- One of the early common implementations: MPICH
  - The CH comes from Chameleon, the portability layer used in the original MPICH to provide portability to the existing message-passing systems
  - OpenMPI: more recent, joint effort of three or four groups (Indiana University, Los Alamos, Tennessee, Europe)
  - Other implementation of the standard out of Ohio State University

Where Can Message Passing Be Used?

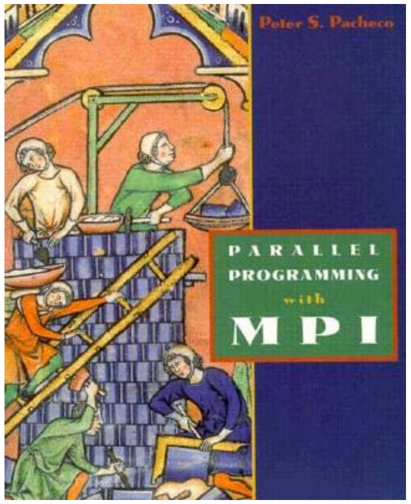
- Message passing can be used wherever it is possible for processes to exchange messages:
  - Distributed memory systems
  - Networks of Workstations
  - Even on shared memory systems ←
    - Pretty common when running jobs on Intel Xeon Phi
      - Example: Have five MPI ranks. Each rank has 12 OpenMP threads

MPI vs. CUDA

- When would you use CPU/GPU computing and when would you use MPI-based parallel programming?
  - Use CPU/GPU
    - If your data fits the memory constraints associated with GPU computing
    - You have parallelism at a fine grain so that you the SIMD paradigm applies
    - Example:
      - Image processing
  - Use MPI-enabled parallel programming
    - If you have a very large problem, with a lot of data that needs to be spread out across several machines
    - Example:
      - Solving large heterogeneous multi-physics problems
- In large scale computing the future likely to belong to heterogeneous architecture
  - A collection of CPU cores that communicate through MPI, each or which farming out work to an accelerator (GPU)

MPI: A Second Example Application

- Example out of Pacheco’s book:
  - “Parallel Programming with MPI”
  - Good book, newer edition available



```
/* greetings.c -- greetings program
 *
 * Send a message from all processes with rank != 0 to process 0.
 *    Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by process 0.
 *
 * See Chapter 3, pp. 41 & ff in PPMPI.
 */
```

MPI: A Second Example Application

[Cntd.]

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int          my_rank;       /* rank of process      */
    int          p;             /* number of processes  */
    int          source;        /* rank of sender       */
    int          dest;          /* rank of receiver     */
    int          tag = 0;       /* tag for messages     */
    char         message[100];  /* storage for message  */
    MPI_Status   status;        /* return status for receive  */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
```

Program Output

```
[negrut@euler CodeBits]$ mpiexec -np 8 ./greetingsMPI.exe
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
[negrut@euler CodeBits]$
```