VGP393 – Week 1

- Agenda:
  - Course road-map
  - Types of parallel architectures
  - Parallel programming / threading terminology

16-July-2008

What should you already know?

- C++ and object oriented programming
  - Most assignments will include some boot-strap code in the form of C++ classes
- Fundamental data structures
  - A lot of we cover will require working knowledge of trees, lists, stacks, and queues
  - Understanding of some STL types will be helpful, but is not required
- Some knowledge of linear algebra / vector math
  - Many of the problems that are intersting to impelment on parallel computers originate in linear algebra

What will you learn?

- Types of parallel computers and system organization
  - SIMD, MIMD, multi-core, multi-threaded, etc.
- Multi-threading primitives and their use
  - Threads, mutexes, semaphores, etc.
- Debugging parallel programs
  - Detecting and avoiding deadlock will be key
- Measuring performance
  - If we double the number of processors, how much speed-up is realized?

How will you be graded?

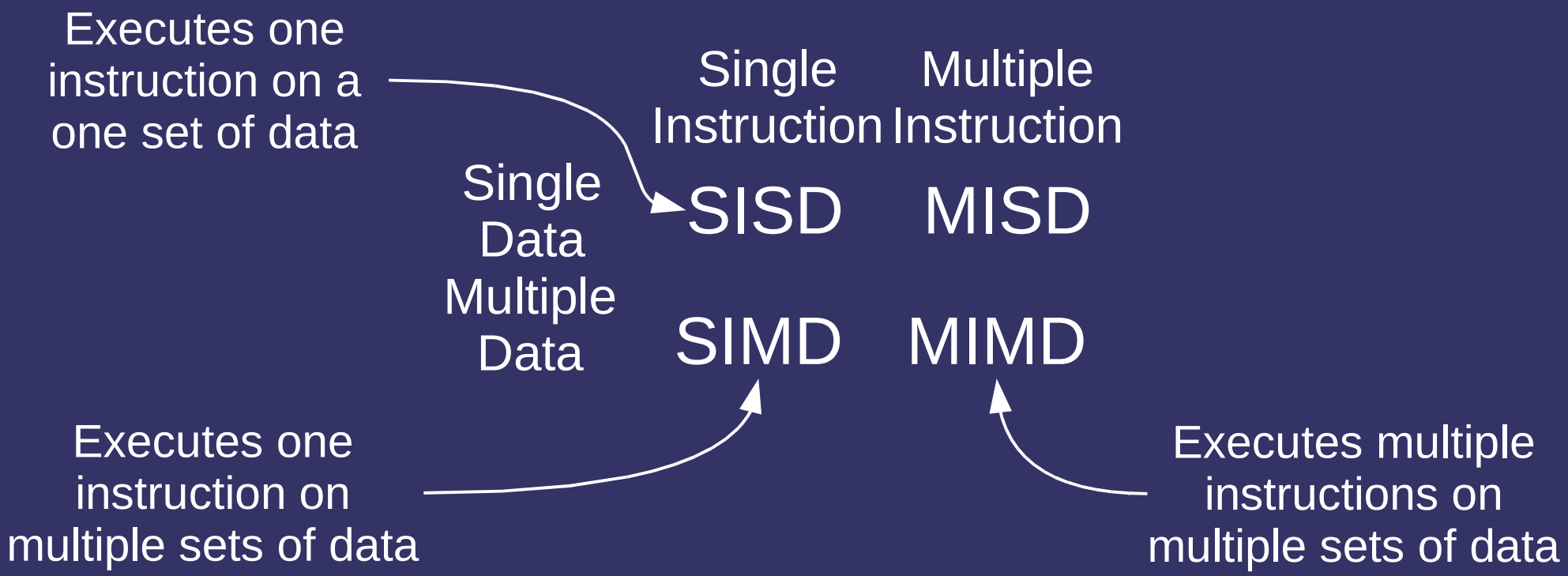
- Four, bi-weekly quizzes worth 5 points each
- Final exam worth 50 points
- Four programming assignments
  - One worth 10 points
  - Three worth 30 points each

How will programs be graded?

- First and foremost, does the program produce the correct output?
- Are appropriate algorithms and data-structures used?
- Is the code readable and clear?

Types of Parallel Computers

- Flynn's taxonomy of sequential and parallel computers provides a high-level model:



16-July-2008

Types of Parallel Computers

- SISD isn't parallel at all
  - *Single instruction, single data* is just another way of saying *sequential* or *scalar*
- Processor may internally execute instructions in parallel, but the programming model is sequential
  - This is called *superscalar*

SISD Architecture

- Superscalar architectures rely on *instruction level parallelism*
  - The ability execute multiple sequential instructions in parallel
  - Branch instructions and data dependencies between instructions make this difficult
    - Instructions can be executed *out of order* to avoid stalls
    - Exponentially more hardware is needed to keep a linear number of execution units busy
  - Cache misses and memory latency cause additional stalls
  - As a result, many execution units may site idle

Types of Parallel Computers

- MISD is nonsense
  - No *multiple instruction, single data* systems have ever been built

Types of Parallel Computers

- *Single instruction, multiple data* (SIMD) computers execute the same instruction stream on different data in parallel
  - Popularized by Cray in the 70's and 80's
  - Also known as *vector processors*
- How to add two arrays of 1,000 numbers?
  - Scalar computers would loop 1,000 times
  - Vector computers would loop 1,000 / *n* times
    - *n* is the number of processing elements in the array

Types of Parallel Computers

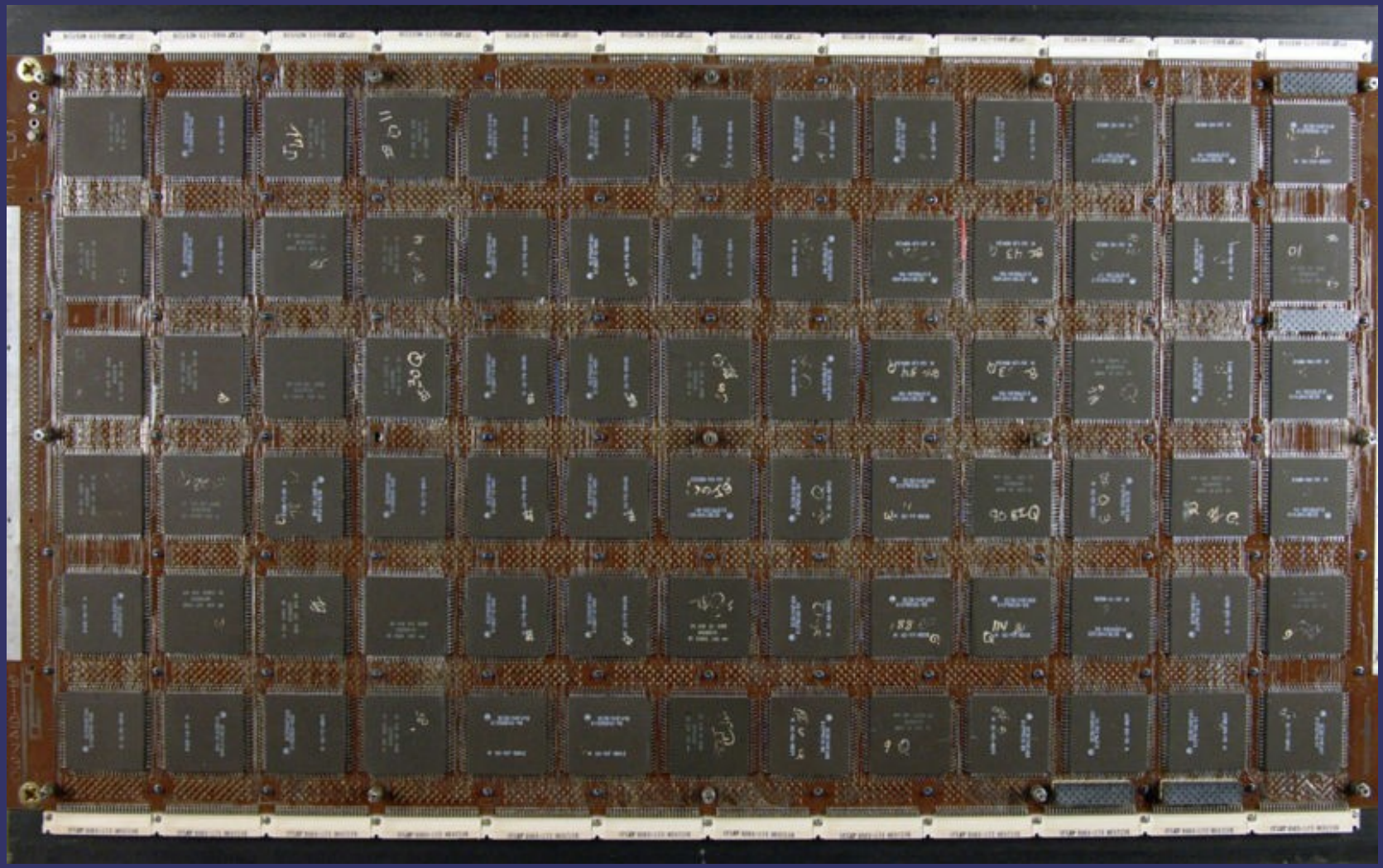
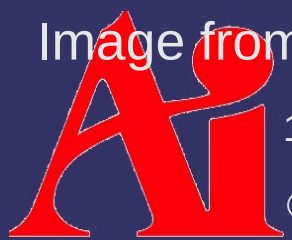


Image from http://en.wikipedia.org/wiki/Image:Processor_board_cray-1_hg.jpg



16-July-2008

Types of Parallel Computers

- Where do we see SIMD today?
  - Instruction “extensions” to general purpose processors
    - Altivec for PowerPC, MMX and SSE for x86, etc.
  - DSPs
  - GPUs
    - DX9 class vertex and fragment shaders are generally implemented as arrays of processing elements that execute the same instructions on different vertices or fragments
    - Any GPU that lacks dynamic branching (Geforce5 / Radeon X800 or earlier) is in this category

Types of Parallel Computers

- *Multiple instruction, multiple data* (MIMD) computers multiple instructions streams on multiple sets of data
  - Wide variety of MIMD architectures
  - May not be as fast at repetitive data manipulations as a comparable vector computer
  - Generally, most flexible and most complicated to write code for
  - Naturally, this where we will spend most of our time this term :)

MIMD Architectures - SMP

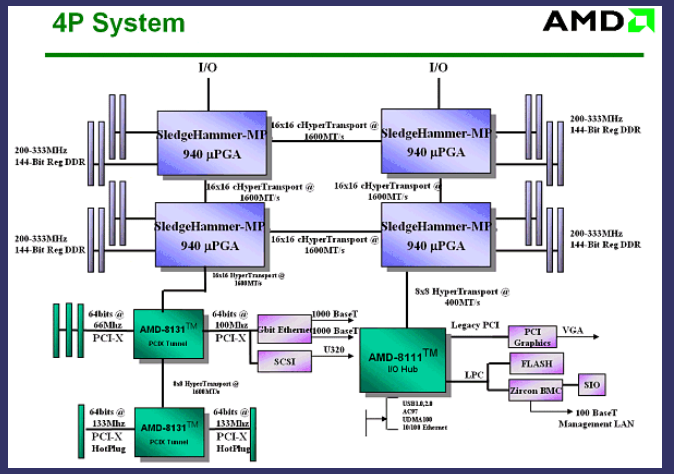
- Symmetric multiprocessor (SMP)
  - Collection of identical processors
  - All processors can access the same memory in, generally, the same amount of time
  - With a few exceptions, SMPs don't scale well beyond 4 processors
    - Memory bandwidth becomes the limiting factor
    - Some exotic memory architectures solve this, but they are *very* expensive and difficult to produce
    - Sequent Symmetry systems shipped with 30 Pentium processors in the mid 90's

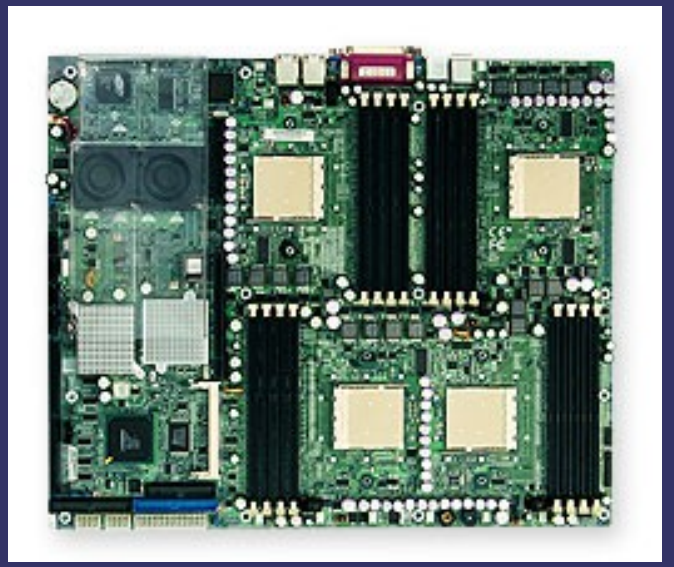
MIMD Architectures - NUMA

- Non-uniform memory access (NUMA) architectures solve the SMP bandwidth problem
  - Processors and memory are grouped in nodes
  - Multiple nodes connected via fast interconnect to create a large system with a flat memory space
    - Access to memory on the same node is fastest
    - All CPUs can see all memory
  - Sequent and SGI shipped high-end NUMA systems in the late 90's
    - Sequent systems were initially based on nodes of 4 PentiumPro processors
  - Multiprocessor Opteron systems are also NUMA

MIMD Architectures - NUMA

- AMD Opteron systems use NUMA
  - Each processor contains a memory controller and cache
  - Each processor also contains links to two other processors
  - A particular memory location may be up to two “hops” away





Images from:
http://www.supermicro.com/Aplus/motherboard/Opteron/nForce/H8QC8+.cfm
http://www.ixbt.com/cpu/amd-hammer-family.shtml

16-July-2008

MIMD Architectures - SMT

- *Simultaneous Multi-threading* (SMT) puts otherwise idle units to work
  - The processor is equipped with multiple sets of "process state"
    - Registers, instruction pointer, stack pointer, etc.
  - Instructions from multiple threads are issued as execution units become available
    - Linear increase in hardware to make use of linear increase in execution units...
    - ...at the expense of increased programming complexity

MIMD Architectures - SMT

- Where do we see SMT today?
  - Developed by Digital for the Alpha 21464
  - Various Intel processors (called Hyper-Threading Technology):
    - Pentium 4 (Northwood and later cores)
    - Atom
  - Various PowerPC processors:
    - POWER Processing Element (PPE) of the Cell processor
    - POWER5 (and later)
    - Xenon (Xbox 360)
  - Nvidia G80 and AMD R600 GPUs

MIMD Architectures - SMT

- Atom, Cell PPE, and Xenon are important special cases
  - Both CPUs are simple, in-order, scalar processors
  - Multi-threading is *necessary* to get reasonable performance!
  - This type of architecture will likely be more common in the future

MIMD Architectures - CMP

- *Chip Multi-processing* (CMP) puts multiple independent processors on a single die
  - Processors may or may not share cache or memory controller
- First shipped by IBM in the POWER4
  - AMD ships 2, 3, and 4 core Athlon64
  - Intel ships dual and quad core Core 2
  - IBM ships 3 core Xenon

MIMD Architectures - CMP

- Growing area of processor development
  - Dual core PCs are universal, quad core is “high end”
    - Intel's Nehalem will have 4 cores w/2 threads per core[1]
  - IBM's Cell contains 8 specialized cores
    - POWER7 is rumored to have 2 chips per module, 8 cores w/ 4 threads per core on each chip[2]
  - Sun's UltraSPARC T1 contains up to 8 cores
  - Expect 10's, 100's, or even 1,000's of cores in the future[3]

[1] http://en.wikipedia.org/wiki/Nehalem_(microarchitecture)
[2] http://www.theregister.co.uk/2008/07/11/ibm_power7_ncsa/
[3] http://news.cnet.com/8301-13924_3-9981760-64.html

MIMD Architectures

- Other MIMD architectures do exist
  - Clusters of commodity systems
  - Supercomputers with message based interconnects
  - Etc.
- As game developers, you're not likely to encounter these types of systems
  - Because of this, we're not going to cover them this term

Types of Parallel Computers

- What's the take-home message?

Types of Parallel Computers

- What's the take-home message?
  - Hardware architectures that were once eccentric are now common place
    - Future performance gains on desktops, consoles, and mobile devices will come from these techniques
  - Software development practices that were once the domain of supercomputer researchers must be taken up by application developers

Amdahl's Law

- Consider a total program run-time given by:

$$T_{total}(1) = T_{setup} + T_{compute} + T_{finalization}$$

- Distributing the work across P PEs results in:

$$T_{total}(P) = T_{setup} + \frac{T_{compute}(1)}{P} + T_{finalization}$$

Amdahl's Law

- The *speed up* from running on P PEs is:

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)}$$

- Portions of the code that cannot be run in parallel are called *serial terms*
  - The *serial fraction*, γ, is the fraction of running time spent in the serial terms
  - The parallelizable part is given by $1 - \gamma$

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)}$$

Amdahl's Law

- Rewrite $T_{total}(P)$ in terms of $T_{total}(1)$ and γ:

$$T_{total}(P) = \gamma T_{total}(1) + \frac{(1-\gamma) T_{total}(1)}{P}$$

- Rewrite S:

$$S(P) = \frac{T_{total}(1)}{\left(\gamma + \frac{1-\gamma}{P}\right) T_{total}(1)}$$

$$S(P) = \frac{1}{\left(\gamma + \frac{1-\gamma}{P}\right)}$$

16-July-2008

Amdahl's Law

- What does it *mean*?

$$S(P) = \frac{1}{\left(\gamma + \frac{1-\gamma}{P}\right)}$$

Amdahl's Law

- What does it *mean*?

$$S(P) = \frac{1}{\left(\gamma + \frac{1-\gamma}{P}\right)}$$

  - The value of γ tells us how much effort to put into optimizing or parallelizing the $(1-\gamma)$ portion
  - It also tells us how much we can increase the workload for a given number of PEs without appreciably affecting the run-time

Threading Terminology

- A *process* "...consists of (1) system resources that are allocated to it, (2) a section of memory, (3) security attributes (such as its owner and its set of permissions) and (4) the processor state.[1]"
  - Processor state includes register contents, the program counter / instruction pointer, stack, and physical memory addressed.

[1] http://www.linfo.org/process.html

Threading Terminology

- A *thread* "...has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history...[1]"

[1] http://www.informit.com/articles/article.aspx?p=25075

Threading Terminology

- What is the difference between a *thread* and a *process*?

Threading Terminology

- What is the difference between a *thread* and a *process*?
  - Processes consist of processor state, system resources (memory, files, etc.) and security attributes
  - Threads consist of processor state
- We can think of a process as a collection of resources, security attributes and *at least one thread*
  - This model is used by most current operating systems
  - *Unit of execution* (UE) is a generic term for a thread or a process

Threading Terminology

- A *processing element* (PE) is a generic term for a piece of hardware that can execute program instructions
  - Processors in an SMP
  - Cores in a CMP
  - Processor contexts / threads in a SMT

Threading Terminology

- *Load balance* is the measure of how even work is distributed among the available PEs
  - *Load balancing* is the process of evenly distributing work to the PEs

Threading Terminology

- *Synchronization* "refers to the coordination of simultaneous [UEs] to complete a task in order to get correct runtime order and avoid unexpected race conditions.[1]"
  - Two events are *synchronous* if they must happen at the same time
  - Otherwise, two events are *asynchronous*

[1] http://en.wikipedia.org/wiki/Synchronization_(computer_science)

Threading Terminology

- A *race condition* a programming error where the "...result of the process is unexpectedly and critically dependent on the sequence or timing of other events.[1]"

[1] http://en.wikipedia.org/wiki/Race_condition

Threading Terminology

- *Deadlock* occurs when forward progress is halted because every task is waiting for some other task to complete some action

  > "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

[1] *A Treasury of Railroad Folklore*, B.A. Botkin & A.F. Harlow, p. 381 by way of http://en.wikipedia.org/wiki/Deadlock

Next week...

- Synchronization
  - Critical sections
  - Deadlock
  - Synchronization primitives
- Win32 / MFC threading API, part 1
  - Creating / destroying threads
  - Events
  - Semaphores
  - Mutexes
  - Critical sections
  - Compiling / linking multi-threaded programs

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.

16-July-2008