Introduction to Parallel Computing

Piyush Mehrotra
NAS Division
NASA Ames Research Center
piyush.mehrotra@nasa.gov

2012 Summer Short Course for
Earth System Modeling and Supercomputing

Agenda

- Parallel Computing
- Parallel Hardware
- Parallel Programming
- Performance Measures

Why Parallel Computing?

- Reduce overall runtime
- Increase fidelity
- Handle larger data
- Support fault tolerance
- Because it is there
  - in nature
  - in systems

Parallelism: Basic Concepts

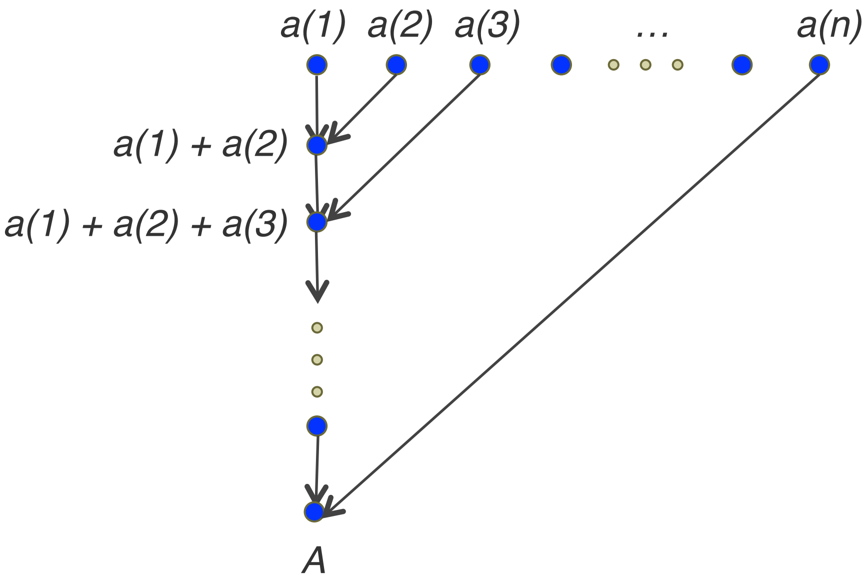
$$A = \sum a(i), \;\; i = 1..n$$

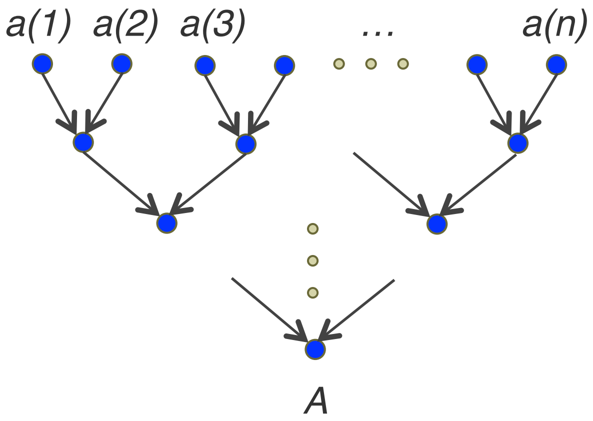
Serial:

- Single processing unit
- $(n - 1)$ steps

Parallel:

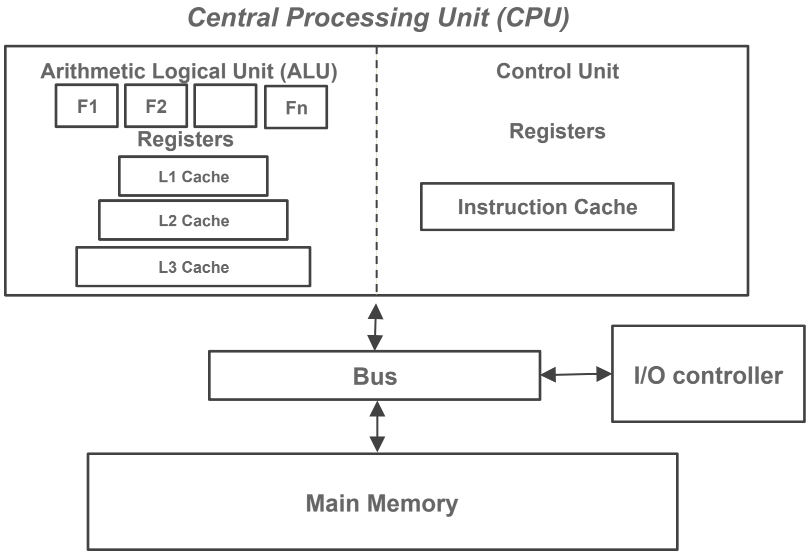
- $(n/2)$ processing units
- $\log_2 n$ steps





Von Neumann Model

- Basis for modern computer architecture
- Introduced the concept of stored program



Modifications to Von Neumann Model

- Cache (data and instruction)
  - Small amounts of low latency memory (at multiple levels)
  - Reduces memory bottleneck
- Multiple functional units – *Instruction Level Parallelism*
  - *Pipelining* – sub-operations performed simultaneously on a data stream
  - *Multiple Issue* – multiple operations executed simultaneously

Low level issues generally handled by the compiler though programmers can structure codes to aid the compiler and the runtime system

Agenda

- Parallel Computing
- Parallel Hardware
  - Taxonomy
  - SIMD
  - MIMD
    - Shared memory
    - Distributed memory
  - Interconnect
  - Hybrid
  - NASA's systems
- Parallel Programming
- Performance Measures

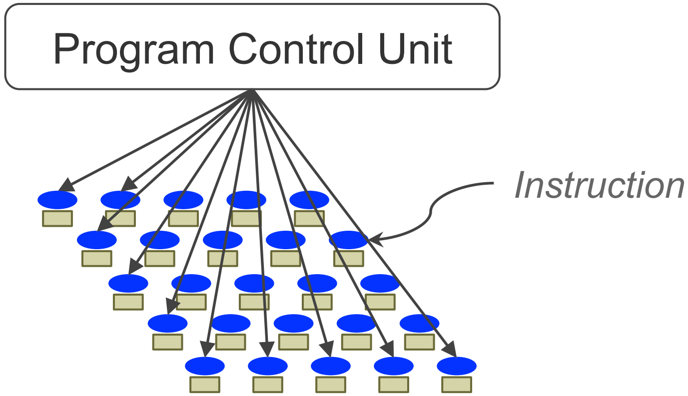
Flynn's Parallel Architecture Taxonomy (1966)

$$\begin{Bmatrix} S \\ M \end{Bmatrix} I \begin{Bmatrix} S \\ M \end{Bmatrix} D$$

- *SISD:* single instruction single data – traditional serial processing
- *MISD:* rare – multiple instructions on a single data item- e.g., for fault tolerance
- *SIMD:* single instruction on multiple data
  - Some old architectures with a resurgence in accelerators
  - Vector processors - pipelining
- *MIMD:* multiple instructions multiple data - almost all parallel computers

SIMD

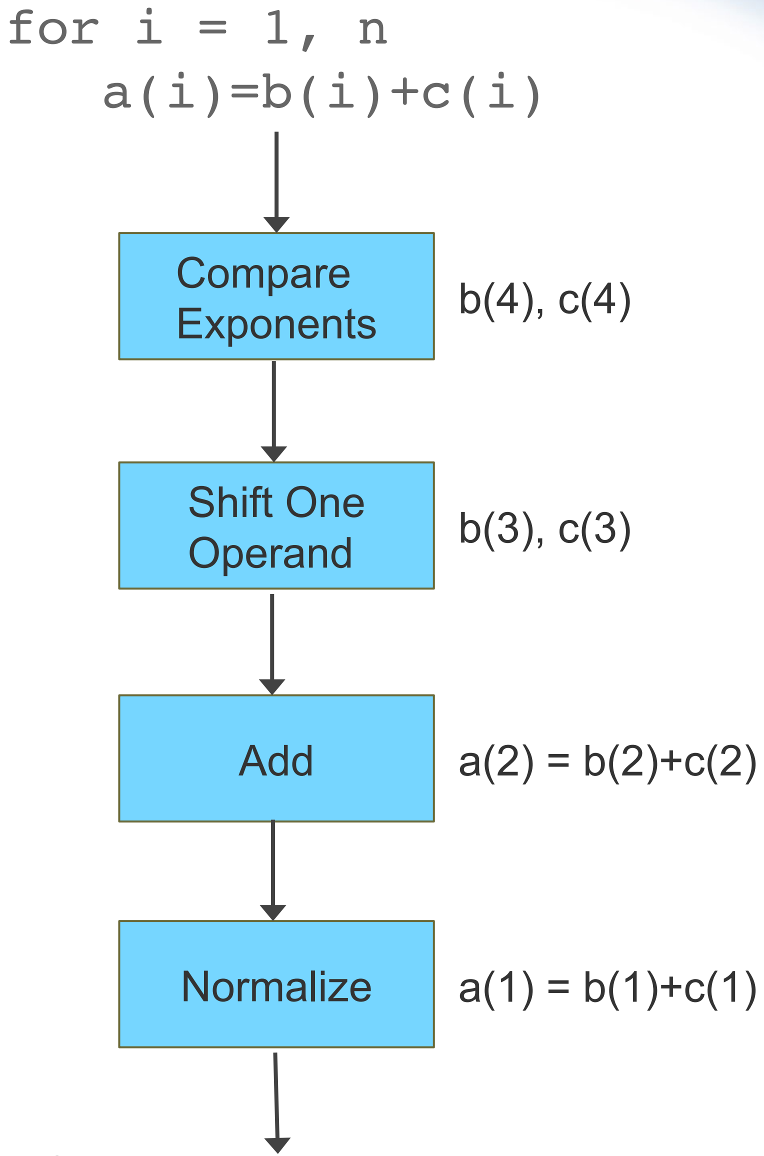
Many small processing cores executing instructions in lock step over multiple data streams



- Several machines built in the early 90's – *Connection Machine* from Thinking Machines
- Programming approach, also called *data parallelism:* apply the same instruction to many elements, e.g., array processing
- Difficult to use for complex algorithms (think conditionals)

Vector Processors

- Specialized form of SIMD
- Akin to factory assembly line – functional units in a vector processor performing different functions on a data stream
- Once the pipeline is full – a result every time unit
- Difficult to use with irregular structures and conditionals



```
for i = 1, n
   a(i)=b(i)+c(i)
```

Compare Exponents b(4), c(4)

Shift One Operand b(3), c(3)

Add a(2) = b(2)+c(2)

Normalize a(1) = b(1)+c(1)

MIMD

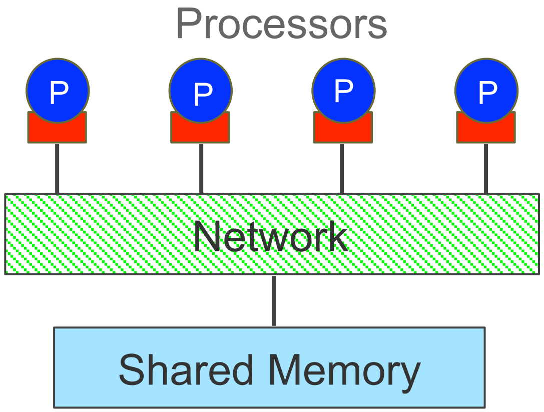
- The most general form of parallel architecture:
  Multiple Instructions Multiple Data

- Multiple processing units executing independent instruction streams on independent data streams

- MIMD Architecture types
  - Shared memory
  - Distributed memory
  - Hybrid

Shared Memory Systems

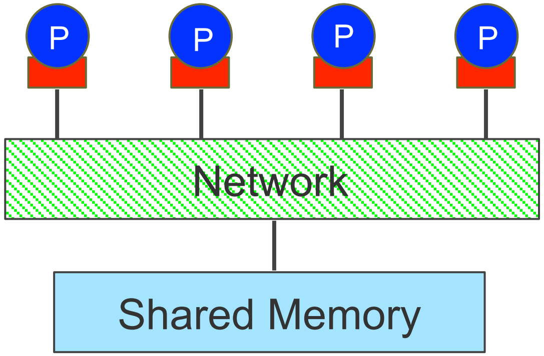
- Multiple processing units accessing global shared memory using a single address space



- Shared memory systems are easier to program
  - User responsible for synchronization of processors for correct data access and modification
- Scaling to large number of processors can be an issue

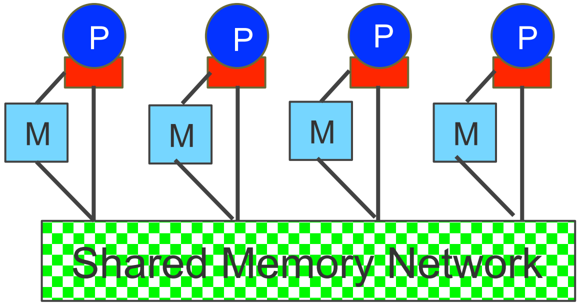
Shared Memory Access

Two types of shared memory systems based on access type:



UMA: Uniform Memory Access – all memory is "equidistant" from all processors

- Memory access can become a bottleneck

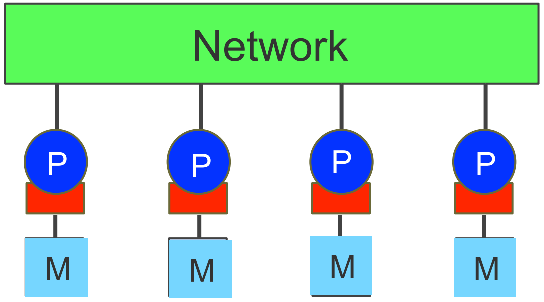


NUMA: Non-Uniform Memory Access – local memory versus distant memory

- Requires more complex interconnect hardware to support global shared memory
- Also called *Distributed shared memory systems*

Distributed Memory Systems

- Multiple processing units with independent local memory and address spaces



Network

P P P P

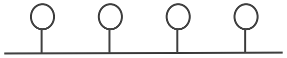
M M M M

Processors + Memory

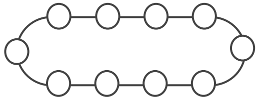
- Systems are easier to scale
- No implicit sharing of data – user is responsible for explicit communication of data amongst processors

Interconnect networks

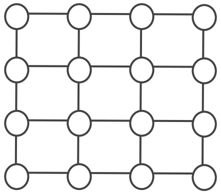
- Topologies:



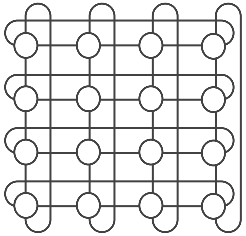
Bus



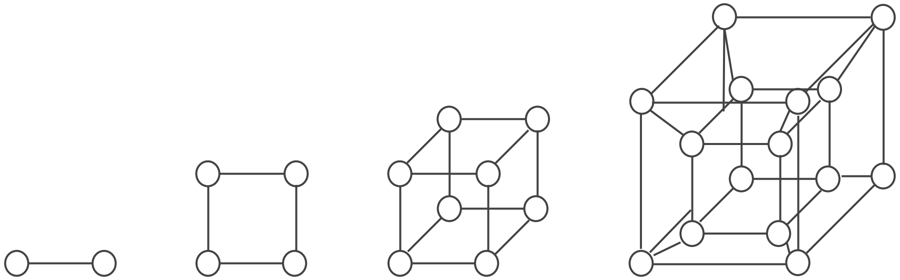
Ring



Mesh

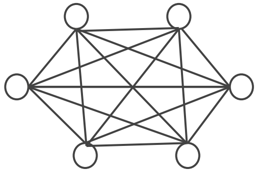


Toroidal Mesh



1d *2d* *3d* *4d*

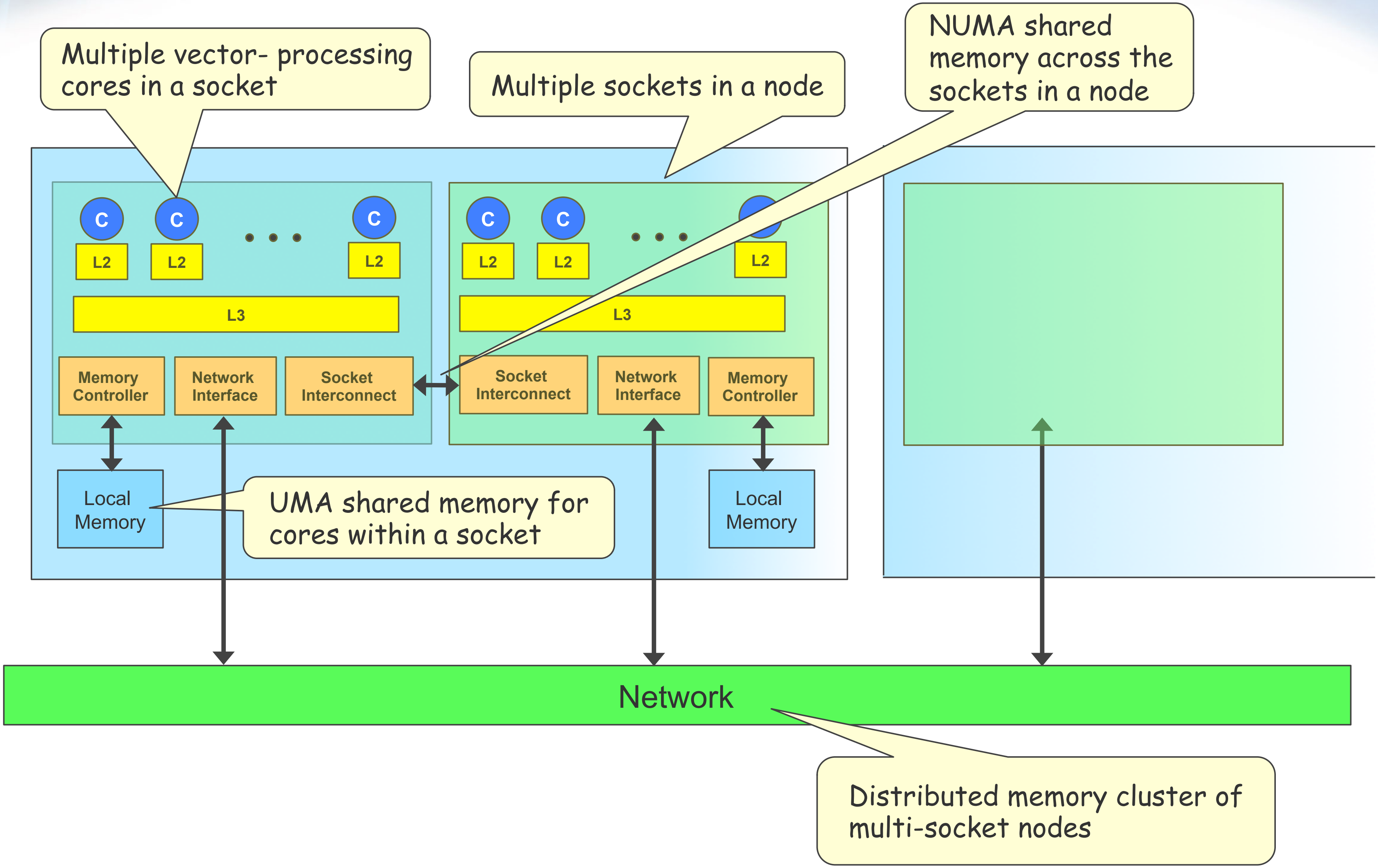
Hypercube



Fully Connected

- Network characteristics:
  - Latency (*l*): time it takes for a link to transmit a unit of data (sec)
  - Bandwidth (*b*): rate at which data is transmitted (bytes/sec)
  - Message transmission time for *n* bytes = $l + n/b$
  - Bisection (band)width: a measure of network quality – number of links connecting two halves of a network

Typical supercomputer: hybrid



Multiple vector- processing cores in a socket

Multiple sockets in a node

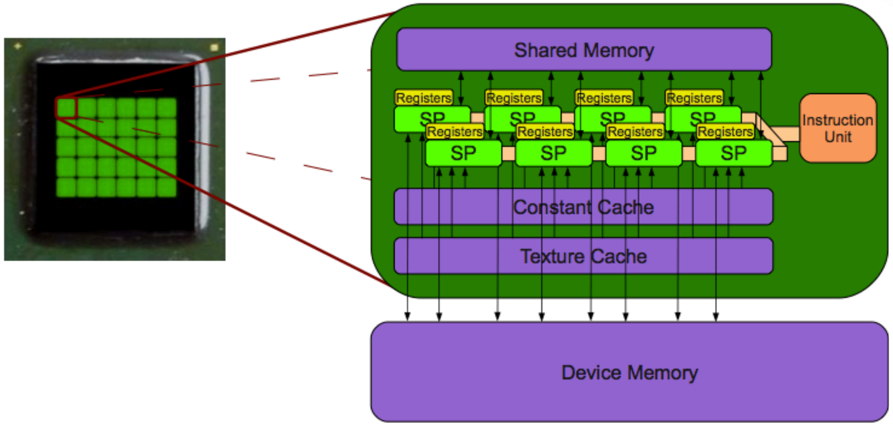
NUMA shared memory across the sockets in a node

UMA shared memory for cores within a socket

Distributed memory cluster of multi-socket nodes

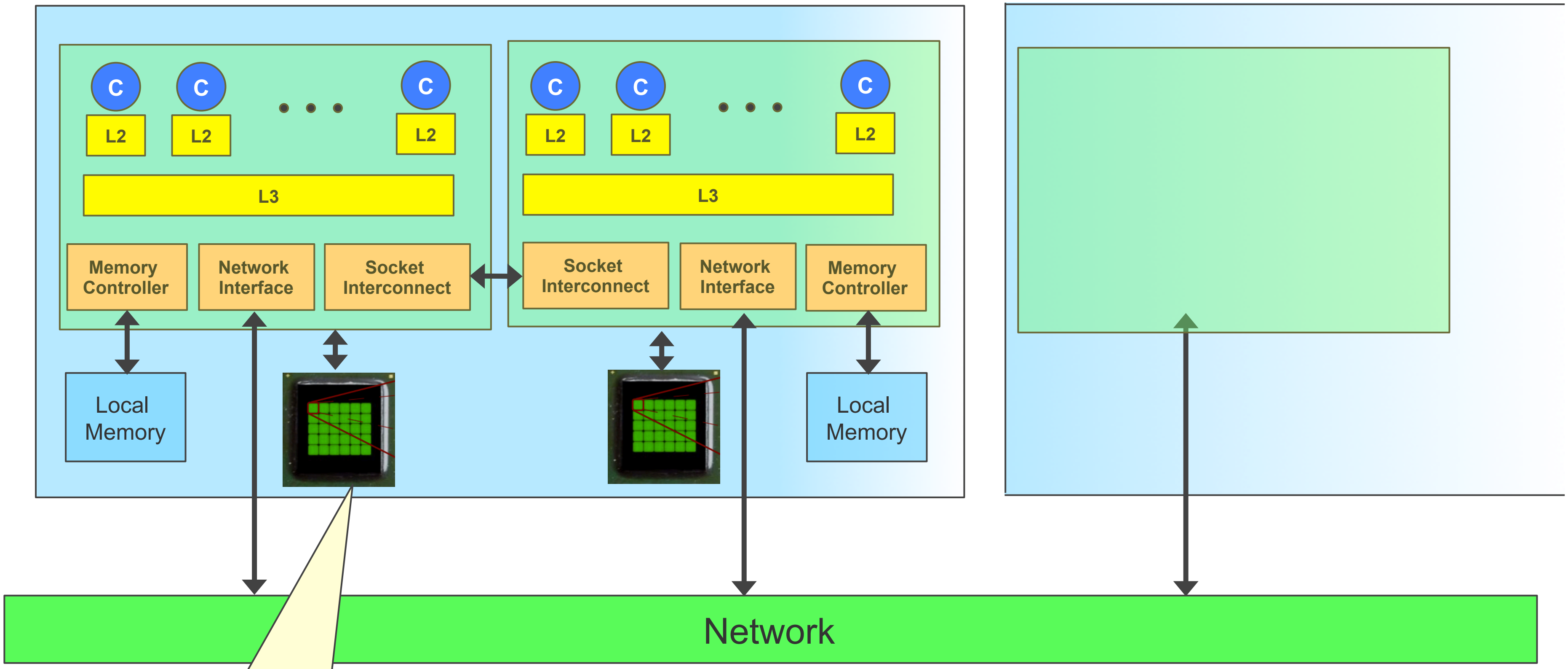
Accelerators

- Co-processor hardware to accelerate computation
  - NVIDIA GPGPUs (General Purpose Graphical Processing Units)
  - Intel MIC (Many-Integrated Cores)
- Generally comprised of many smaller cores with local memory executing in SIMD or MIMD mode



- Advantage: Capable of providing significant computational capability at a lower power draw
- Disadvantage: Programmability
  - Partition the code between the CPU and the accelerator
  - Optimize the code for execution on the accelerator
  - Manage the data movement between the CPU and accelerator memories

Typical supercomputer: with accelerators



Nodes augmented with accelerator chips

Pleiades

- NASA’s premier supercomputer
  - Peak performance: 1.75 PetaFlops
  - #11 in the TOP500 list
- SGI Altix ICE: distributed memory cluster
  - Four generations of Intel Xeon processor dual-socket nodes:
    - ➢ Harpertown (4 cores/socket): 4096 nodes
    - ➢ Nehalem (4 cores/socket): 1280 nodes
    - ➢ Westmere (6 cores/socket): 4608 nodes
    - ➢ Sandy Bridge (8 cores/socket): 1728 nodes
    - Total cores: 126,720
    - Total memory: 233 TB
  - 64 Westmere-based nodes enhanced with NVIDIA M2090 GPUs
- InfiniBand-based 12d hypercube interconnect
- Storage: 10 PB disk; 50 PB archival tape

http://www.nas.nasa.gov/hecc/resources/environment.html

Columbia

- Cluster of SGI Altix shared memory “nodes”
- SGI ALTIX 4700:
  - Intel Itanium dual-core processors
  - SGI NUMAlink technology for shared memory
  - Four nodes:
    - 1 x 2048 core – 4TB shared memory
    - 2 x 1024 core – 2TB shared memory
    - 1 x 512 core – 1TB shared memory
  - InfiniBand node interconnect

Agenda

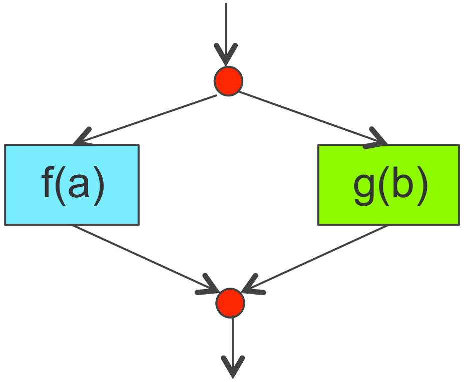
- Parallel Computing
- Parallel Hardware
- Parallel Programming
  - Forms of parallelism
  - Shared Memory Programming: OpenMP
  - Distributed Memory Programming
  - Other approaches
- Performance Measures

Parallel Programming

- Parallel programming requires specifying:
  - Units of work to be done in parallel
  - Data to be shared and/or communicated
  - Synchronization between the tasks
- Two kinds of parallelism
  - Task / Functional Parallelism
  - Data Parallelism

Task Parallelism

- Independent computations performed in parallel



- Low level: functional units in a CPU
- High level:
  - Multiple programs
    - Coarse pipelines
  - Parameter space sweeps

Data parallelism

- Same or similar computations performed on independent parts of the data
- SIMD or vector systems:
  - Mostly handled automatically by compilers via loop parallelization

```
        for (i  =  1: n) x(i) = y(i) + z(i)
```

- MIMD systems: the most popular approach is called *SPMD (Single Program Multiple Data)*
  - Single program executed by independent processes on multiple data sets synchronizing only when necessary

Approaches to Parallel Programming

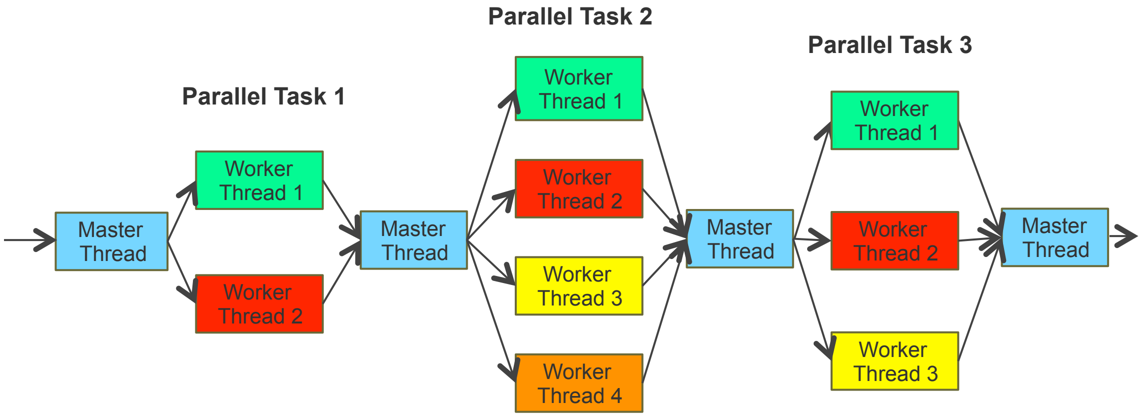
- Shared memory programming: assumes a global address space – global data visible to all processes
  - Issue: synchronizing updates of shared data
  - OpenMP

$$sum = sum + a(i) , i = 1, n$$

- Distributed memory programming: assumes distributed address spaces – each process sees only its local data
  - Issue: communication of data to other processes
  - MPI (Message Passing Interface)

OpenMP

- *OpenMP*: a standard API to support shared memory parallel rogramming
  - Managed by the OpenMP Architecture Review Board, OpenMP v1.0 was released in 1997, latest v3.1 released July 2011
- A directive-based approach to control:
  - *Parallel threads*: Master thread creates parallel worker threads and the work is divided amongst the workers
  - *Data sharing*: assumed a global address space
- Major components:
  - Parallel control structure
  - Work sharing
  - Data sharing and control
  - Synchronization
  - Other runtime functions



OpenMP (contd.)

- Parallel control structure: to specify invocation of the worker threads
  - Number of threads specified external to the program

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

- Work sharing
  - *for* (or *do)*: used to split up loop iterations among the threads
  - *sections*: assigning consecutive but independent code blocks to different threads (can be used to specify task parallelism)
  - *single*: specifying a code block executed by only one thread
  - *master*: code block executed by the master thread only

OpenMP (contd.)

- *Data sharing*: by default variables are visible to all threads. Constructs to control visibility include:
  - *shared*: variable is shared by all threads simultaneously
  - *private*: variable is private to each thread – each thread has a private copy

```
#pragma omp parallel for private(W)
```

- Synchronization
  - *critical*: code block executed by only one thread at a time e.g., allows multiple threads to update shared data
  - *barrier*: wait until all of threads of a team have reached this point before continuing. Work sharing constructs have implicit barriers at the end.

OpenMP code: sum of squares

```
...
long sum = 0, loc_sum;
int thread_id;
#pragma omp parallel private(thread_id, loc_sum)
{
loc_sum = 0;
thread_id = omp_get_thread_num();
#pragma omp for
 for(i = 0; i < N; i++) loc_sum = loc_sum +  i * i;

printf("\n Thread %i: %li\n", thread_id, loc_sum);

#pragma omp critical
sum = sum + loc_sum;
}
printf("\n Sum of Squares = %li", sum);
```

Forks off the threads and starts the work-sharing construct; declares *thread_id* and *loc_sum* private

Each thread retrieves its own id`

Parallel for splits loop range across the threads.

Each thread computes and prints its id and local sum

Threads cooperate to update global variable one by one

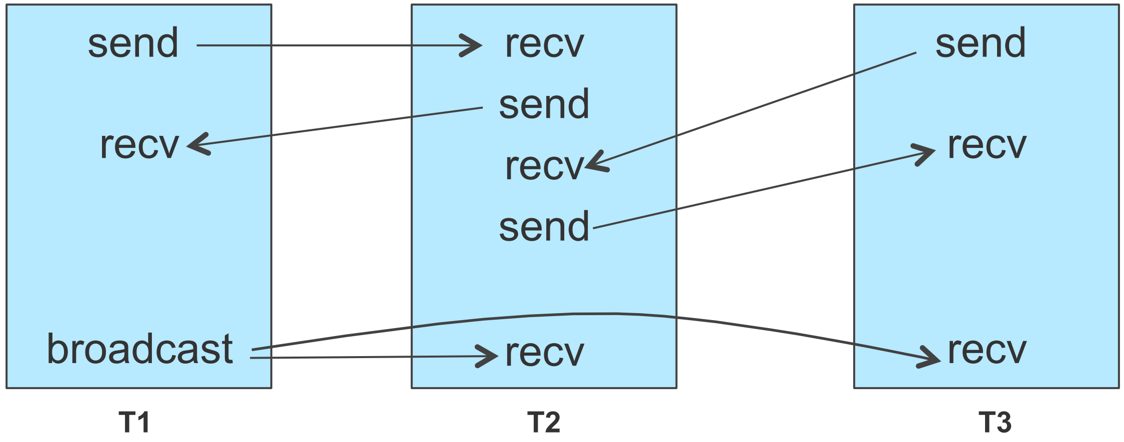
Master thread prints result

Other Shared Memory Approaches

- Thread libraries
  - Posix Threads
  - Intel Thread Building Blocks
- Global Arrays
- Cilk
- …

MPI

- MPI (Message Passing Interface): a standard *message passing* library specification to support process communication on a variety of systems
  - MPI v1.0 (June 1994), latest MPI v2.2 (Sept 2009)
- MPI assumes a distributed address space, i.e., each process (rank) sees only local variables with explicit constructs to communicate data to other processes



MPI Features

- MPI-1
  - General: Init/finalize, Communication group size/rank
  - Point to Point communication:
    - send, recv with multiple modes (blocking/non blocking, …)
  - Collective communication:
    - Barrier for synchronization
    - Broadcast
    - Gather/scatter
    - Reduction operations (built-in and user defined)
- MPI-2
  - One-sided communication: Put, Get, Accumulate
  - Extensions to collectives
  - Dynamic process management

MPI code: sum of squares

```
…
int num_tasks, my_rank, rc;
int  sum, loc_sum, N = …;

MPI_Init();
MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    for(i = 0; i < N; i += numtasks) loc_sum = loc_sum +  i * i;
    printf("\n Thread %i: %li\n", my_rank, loc_sum);

    if (my_rank != 0)
            rc = MPI_Send(loc_sum, 1, MPI_INTEGER, 0, my_rank, … );
    else {
        sum = loc_sum
        for (i = 1; i < num_tasks; i++) {
            rc = MPI_Recv(&loc_sum, 1, MPI_INTEGER, i, i, … );
            sum = sum + loc_sum
        }
        printf("\n Sum %i: %li\n", my_rank, loc_sum);
    }

MPI_Finalize();
```

Each process retrieves its rank`

Each process computes and prints its rank and local sum

Each process sends local sum to rank 0

Rank 0 receives data from all other ranks, computes and prints result

OpenMP vs MPI

- OpenMP – directive based
  - needs compiler (and runtime) support
  - directives can be ignored for serial compilation
- MPI – library based, requires only runtime support
- Both can be used to program both task and data parallelism
- OpenMP allows incremental parallelization while the whole code needs to be parallelized when using MPI
- OpenMP can only be used on systems with a global address space (through hardware or software)
- MPI can be used on both shared and distributed memory systems
- Current supercomputers are hybrid: distributed memory cluster of shared memory nodes. Approaches to programming such systems:
  - MPI: processes run on all cores within the node
  - Hybrid:
    - MPI at the outer level to specify processes across nodes
    - OpenMP within each MPI rank to exploit shared memory in a node

Other approaches

- Libraries
- Programming Environments, e.g., Matlab
- GPU programming: CUDA, OpenCL
- OpenACC: directives for accelerators
- Domain specific languages and frameworks
- PGAS: Partitioned Global Address Space Languages
- Automatic parallelization tools

Research in parallel programming languages is focused on

- High level abstractions so that users can program as close to their domain as possible, along with
- Software (compilers, libraries, runtime systems) to automatically and effectively exploit a variety of underlying architectures

Agenda

- Parallel Computing
- Parallel Hardware
- Parallel Programming
- Performance Measures
  - Speedup & Efficiency
  - Amdahl’s Law
  - Scalability

Performance Measures

- So how well does our parallel program perform?
- Some sources of inefficiencies
  - Overhead due to introducing parallelism:
    - setting up processes, synchronizing, communicating, load imbalance
  - Serial sections

- Speedup (S) is one measure:

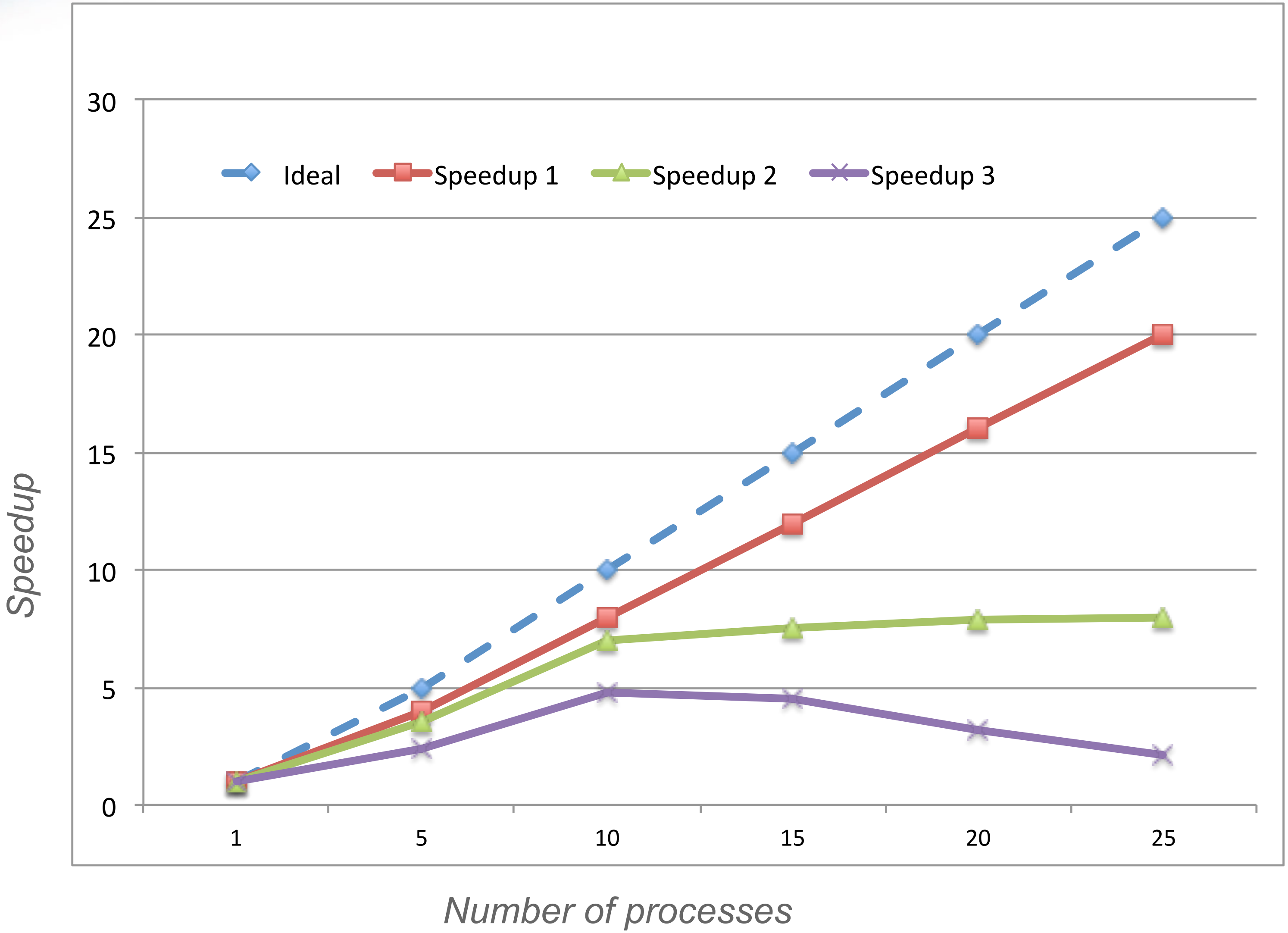
$$S = T(serial) / T(parallel)$$

- On p processors, perfect speedup is

$$S = p$$

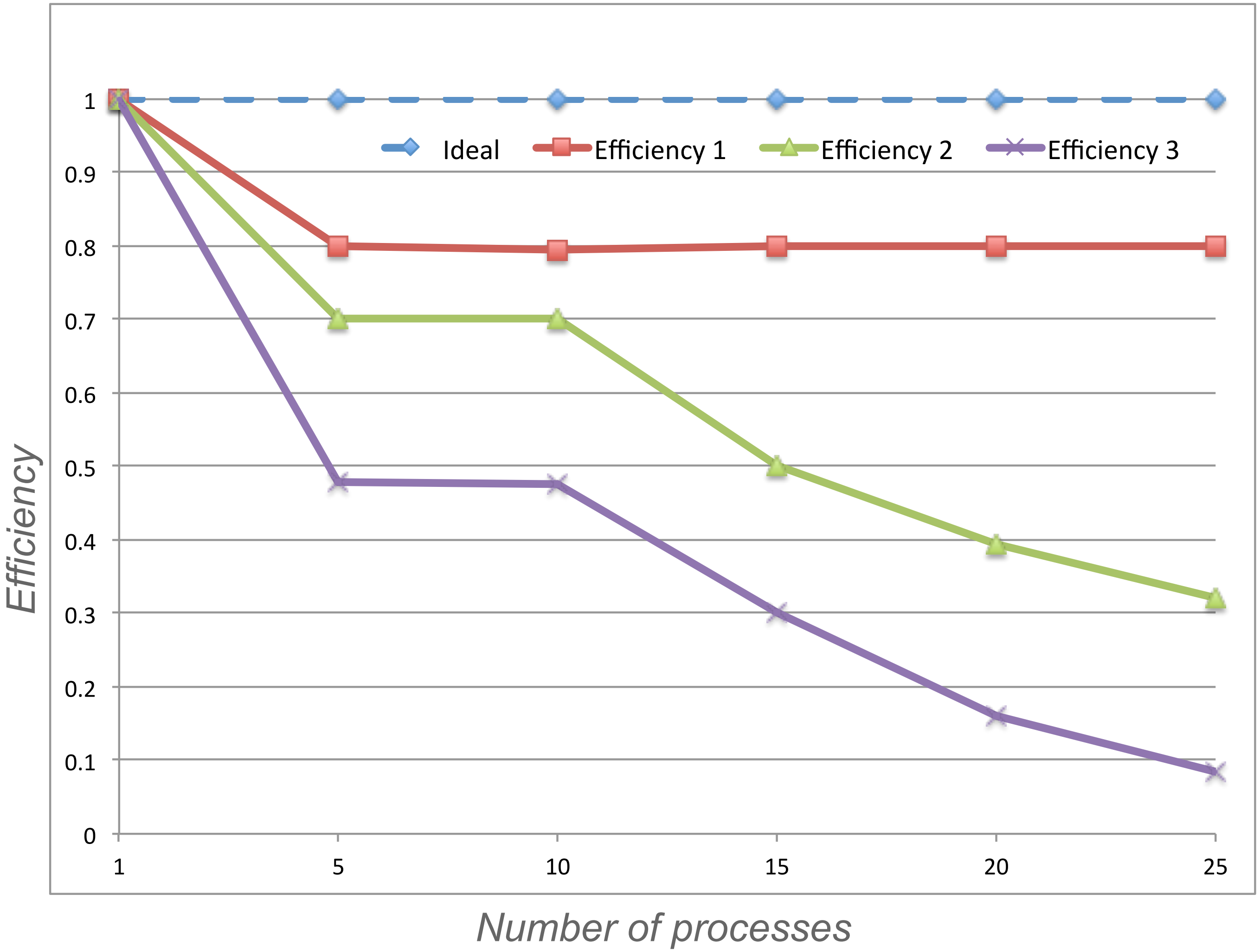
Generally $S < p$ due to overheads etc.

Sample Speedup Curves



Efficiency

- Efficiency: $E = S / p = T(serial) / p * T(parallel)$



Amdahl's Law

- Amdahl's Law: speedup obtainable by a parallel program is limited by the serial portions of the code
- Let r be the parallelizable fraction of the program
- If we get perfect parallel speedup on p processors:

$$S = T(serial) / (r^*T(serial)/p + (1-r)^* T(serial))$$

$$= 1 / (r/p + (1-r))$$

$$\leq 1 / (1-r)$$ for large values of p

- So if $r = 0.9, S \leq 10$ and so forth
- Caveat: does not take into account problem size
  - Increasing the problem size generally increases r

Scalability

- How well does a parallel program handle increasing problem size?
- A program is considered *weakly scalable* if as the problem size increases, we can achieve constant efficiency by increasing the number of processes at the same rate
- *Strong scaling*: efficiency remains constant as we increase number of processes with a *fixed* problem size

Other Issues

- Parallel I/O
- Debugging
- Fault tolerance/Check pointing
- Performance analysis and optimization

References

- Web – a rich resource
  - Wikipedia
- Books
  - *An Introduction to Parallel Programming,* by Peter Pacheco
  - *Introduction to Parallel Computing,* by Ananth Grama, George Karypis, Vipin Kumar and Anshul Gupta
  - *Scientific Computing* by Michael T. Heath