Introduction to Parallel Computing

September 2018

Mike Jarsulic

Introduction

So Far...

Year	Users	Jobs	CPU Hours	Support Tickets
2012	36	*	1.2M	*
2013	71	*	1.9M	141
2014	84	739K	5.3M	391
2015	125	1.5M	5.2M	372
2016	153	1.6M	6.1M	363
2017	256	4.8M	14.7M	759
2018	296	3.0M	8.8M	631
TOTAL	*	**11.7M**	**40.2M**	**2657**

However…

The BSD research community does not understand how to utilize resource properly

Example:

- August 1st, 2018
- 1021 cores in use
- 39 jobs had requested a total of 358 cores
- All 39 jobs were single core jobs
- Thus, 319 cores were dedicated, but idle

My takeaways:

- The BSD research community does not really understand parallel computing!
- I have not done a good of training the BSD research community in this area!

Office Hours

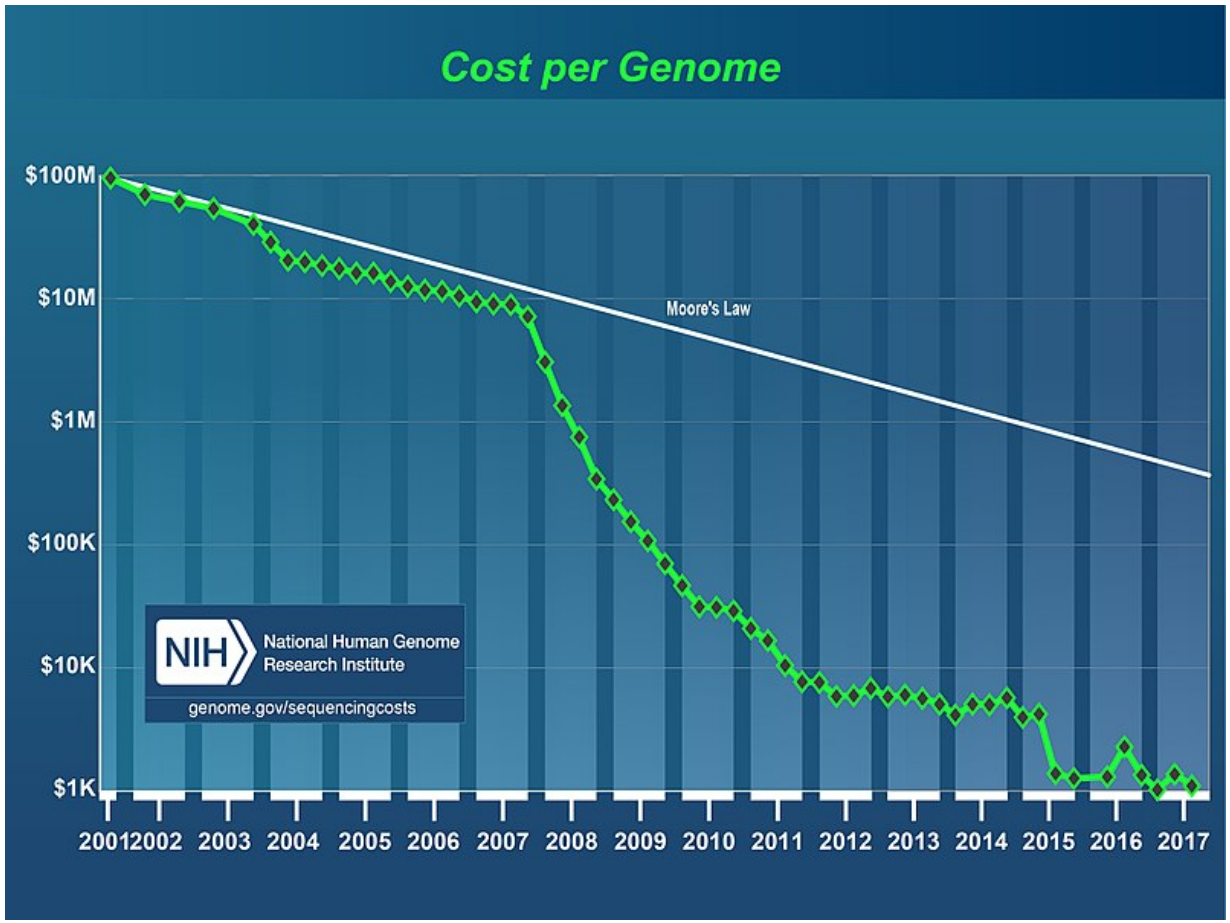
Held every Tuesday and Thursday at Peck Pavilion N161

Users come in to work one-on-one to better utilize our HPC resources

Three common questions on parallel computing:

1. Is this a worthwhile skill to learn? Aren't single processor systems fast enough and always improving?
2. Why can't processor manufacturers just make faster processors?
3. Will you write me a program that will automatically convert my serial application into a parallel application?

Obligatory Cost per Genome Slide



Motivation

Goals (of this presentation)

Understand the basic terminology found in parallel computing

Recognize the different types of parallel architectures

Learn how parallel performance is measured

Gain a high-level understanding to the different paradigms in parallel programming and what libraries are available to implement those paradigms

Please note that you will not be a parallel programming rock star after this three hour introduction

Overall Goals

Hopefully, there will be enough interest after this presentation to support a workshop series on parallel computing

Possible topics for workshops are:

- Scientific Programming
- Parameterization
- OpenMP
- MPI
- CUDA
- OpenACC

Today's Outline

Terminology

Parallel Architectures

Measuring Parallel Performance

Embarrassingly Parallel Workflows

Shared Memory

Distributed Memory

Vectorization

Hybrid Computing

A Note on Sources

Presentation given at SC17 by Stout and Jablonowski from the University o
Michigan

Introduction to Parallel Computing 2nd Edition by Grama, Gupta, et al.

Introduction to Parallel Programming by Pacheco

Using OpenMP by Chapman, Jost, and Van Der Pas

Using MPI by Gropp, Lusk, Skjellum

Advanced MPI by Gropp, Hoefler, et al.

CUDA for Engineers by Storti and Yurtoglu

CUDA by Example by Sanders and Kandrot

Parallel Programming with OpenACC by Farber

The LLNL OpenMP and MPI tutorials

A Note on Format

Today's Seminar will be recorded for a ITM/CTSA grant

- Questions
- Whiteboard
- Swearing

Terminology

Caveats

There are no standardized definitions for terms in parallel computing thus resources outside of this presentation may use the terms in a different way

Many algorithms or software applications that you will find in the rea world do not fall neatly into the categories mentioned in this presentation since they blend approaches in ways that difficult to categorize.

Basic Terminology

Parallel Computing – solving a problem in which multiple tasks cooperate closely and make simultaneous use of multiple processors

Embarrassingly Parallel – solving many similar, independent tasks; parameterization

Multiprocessor – multiple processors (cores) on a single chip

Symmetric Multiprocessing (SMP) - multiple processors sharing a single address space, OS instance, storage, etc. All processors are treated equally by the OS

UMA – Uniform Memory Access; all processors share the memory space uniformly

NUMA – Non-Uniform Memory Access; memory access time depends on the location of the memory relative to the processor

More Basic Terminology

Cluster Computing – A parallel system consisting of a combination of commodity units (processors or SMPs)

Capacity Cluster – A cluster designed to run a variety types of problems regardless of the size; the goal is to perform as much research as possible

Capability Cluster – The fastest, largest machines designed to solve large problems; the goal is to demonstrate capability of the parallel system

High Performance Computing – Solving large problems via a cluster of computers that consist of fast networks and massive storage

Even More Basic Terminology

Instruction-Level Parallelism (ILP) - Improves processor performance by having multiple processor components simultaneously executing instructions

Pipelining – Breaking a task into steps performed by different processors with inputs streaming through, much like an assembly line

Superscalar Execution - The ability for a processor to execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor

Cache Coherence - Synchronization between local caches for each processor

Crash Simulation Example

Simplified model based on a crash simulation for the Ford Motor Company

Illustrates various aspects common to many simulations and applications

This example was provided by Q. Stout and C. Jablonowski of the University of Michigan

Finite Element Representation

Car is modeled by a triangulated surface (elements)

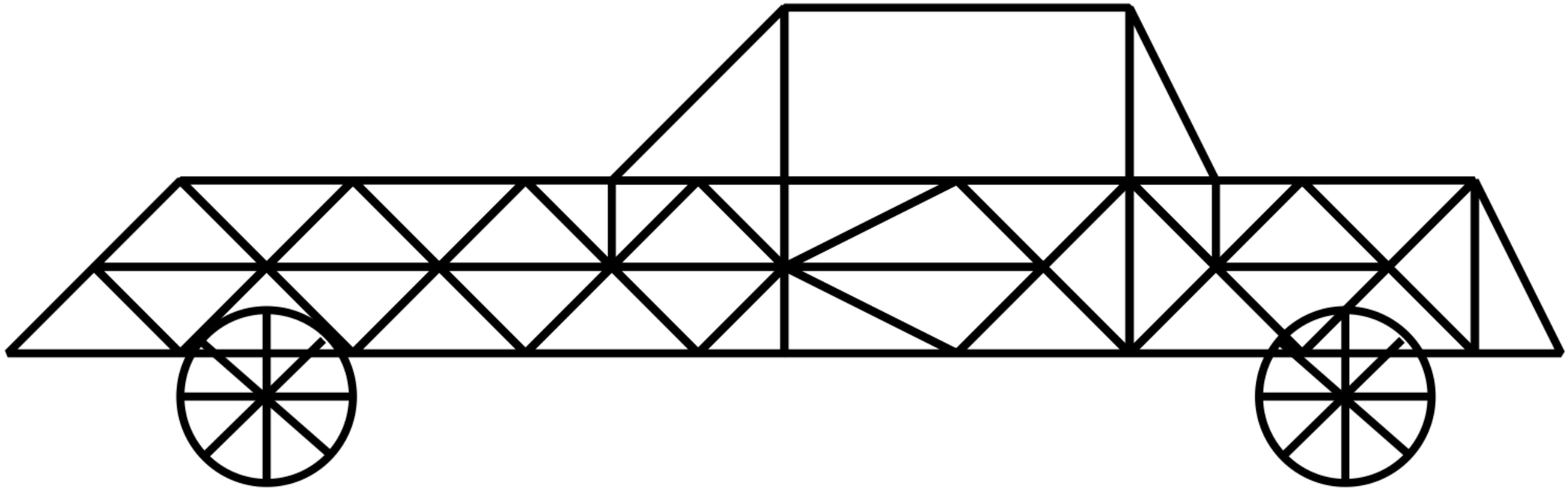
The simulation models the movement of the elements, incorporating the forces on the elements to determine their new position

In each time step, the movement of each element depends on its interaction with the other elements to which it is physically adjacent

In a crash, elements may end up touching that were not touching initially

The state of an element is its location, velocity, and information such as whether it is metal that is bending

Car



Serial Crash Simulation

1. For all elements
2. Read State(element), Properties(element), Neighbor_list(element)
3. For time=1 to end_of_simulation
4. For element=1 to num_elements
5. Compute State(element) for next time step, based on the previous state of the element and its neighbors and the properties of the element

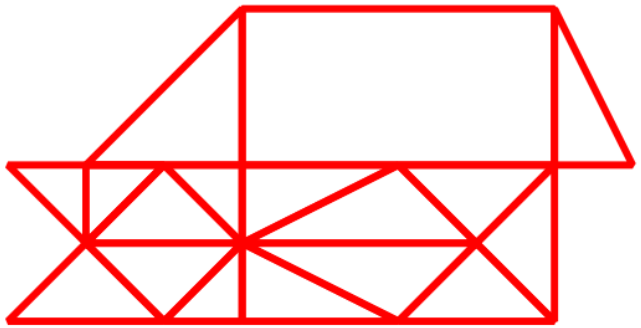
Simple Approach to Parallelization (Concepts)

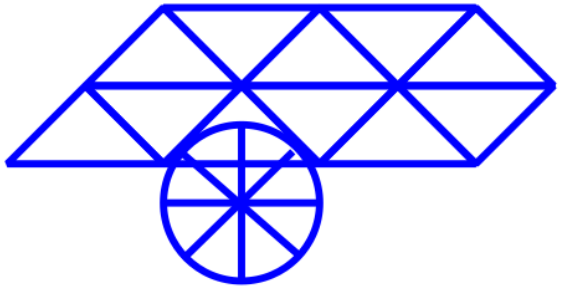
Distributed Memory – Parallel system based on processors linked with a fast network; processors communicate via messages

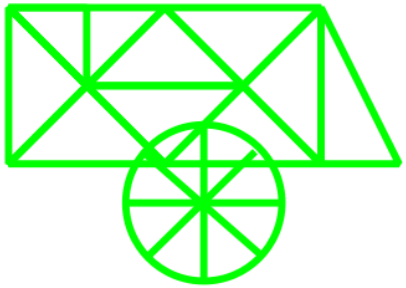
Owner Computes – Distribute elements to processors; each processor updates the elements which it contains

Single Program Multiple Data (SPMD) - All machines run the same program on independent data; dominant form of parallel computing

Distributed Car







Basic Parallel Version

Concurrently for all processors P

1. For all elements assigned to P
2. Read State(element), Properties(element), Neighbor-list(element)
3. For time=1 to end_of_simulation
4. For element=1 to num_elements_in_P
5. Compute State (element) for next time step, based on previous state of element and its neighbors, and on properties of the element

Notes

Most of the code is the same as, or similar to, serial code.

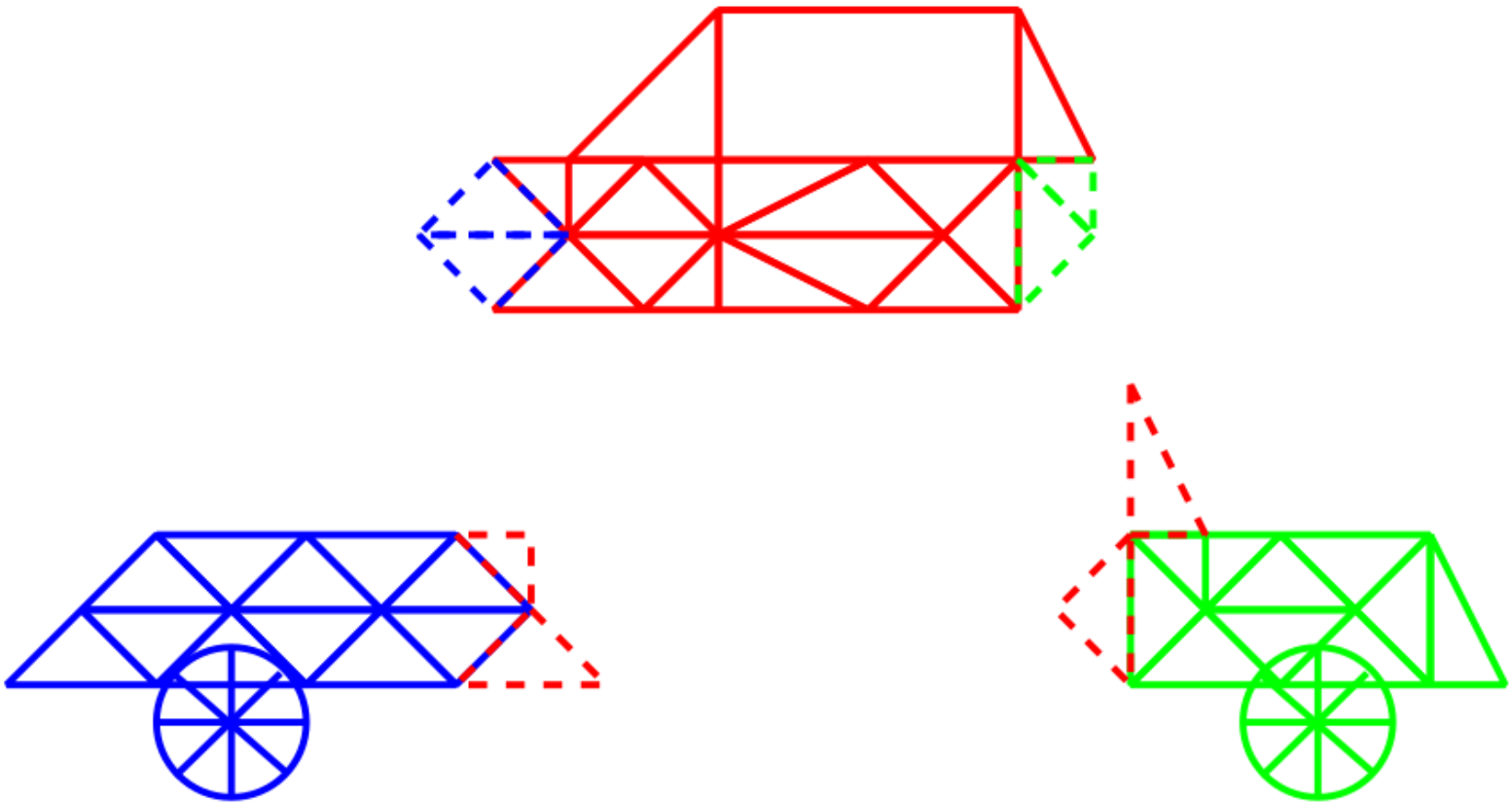
High-level structure remains the same: a sequence of steps

- The sequence is a serial construct, but
- Now the steps are performed in parallel, but
- Calculations for individual elements are serial

Question:

- How does each processor keep track of adjacency info for neighbors in other processors?

Distributed Car (ghost cells)



Parallel Architectures

Flynn's Taxonomies

Hardware Classifications

{ S, M } I { S, M } D

Single Instruction (SI) - System in which all processors execute the same instruction

Multiple Instruction (MI) - System in which different processors may execute different instructions

Single Data (SD) - System in which all processors operate on the same data

Multiple Data (MD) - System in which different processors may operate on different data

M. J. Flynn. Some computer organizations and their effectiveness. IEEE Transactions on Computers, C-21(9):948–960, 1972.

Flynn's Taxonomies

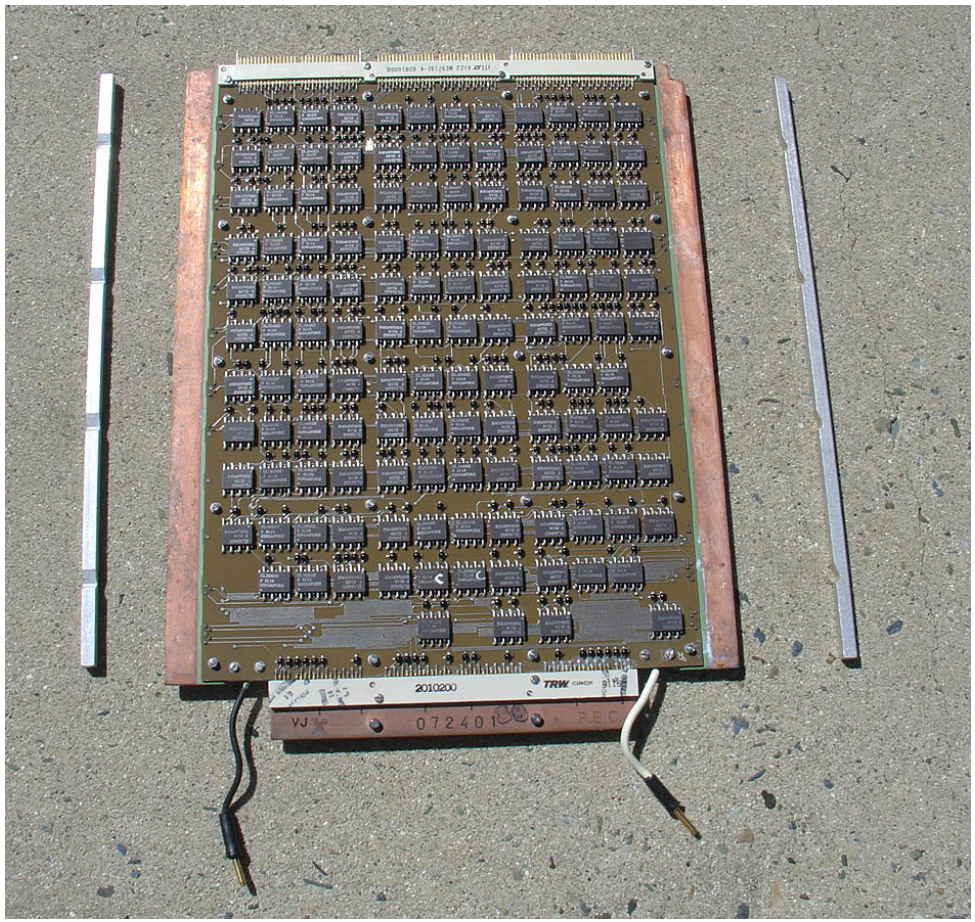
SISD – Classic von Neumann architecture; serial computer

MIMD – Collections of autonomous processors that can execute multiple independent programs; each of which can have its own data stream

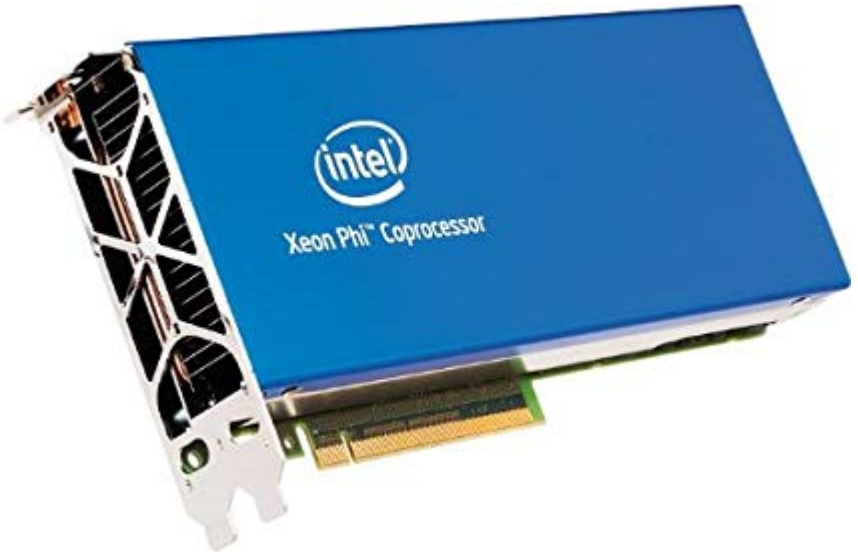
SIMD – Data is divided among the processors and each data item is subjected to the same sequence of instructions; GPUs, Advanced Vector Extensions (AVX)

MISD – Very rare; systolic arrays; smart phones carried by Chupacabras

CRAY-1 Vector Machine (1976)



Vector Machines Today



Software Taxonomies

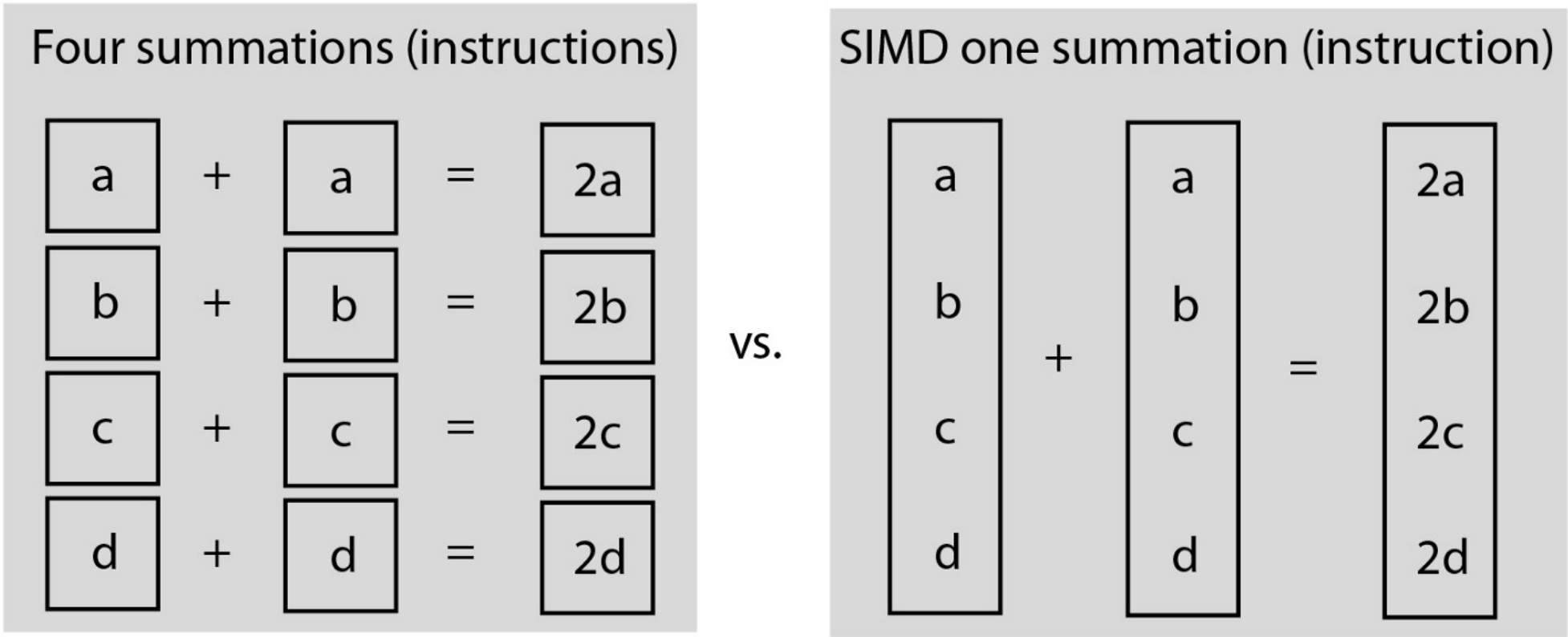
Data Parallel (SIMD)

- Parallelism that is a result of identical operations being applied concurrently on different data items; e.g., many matrix algorithms
- Difficult to apply to complex problems

Single Program, Multiple Data (SPMD)

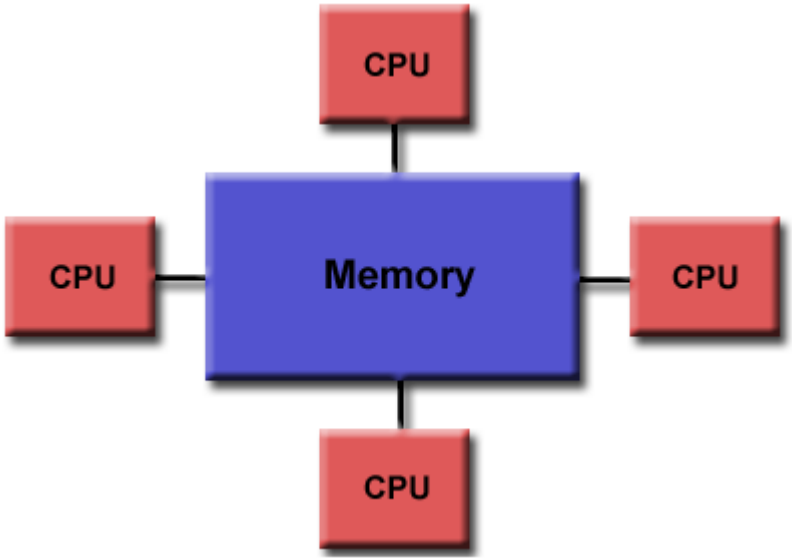
- A single application is run across multiple processes/threads on a MIMD architecture
- Most processes execute the same code but do not work in lock-step
- Dominant form of parallel programming

SISD vs. SIMD

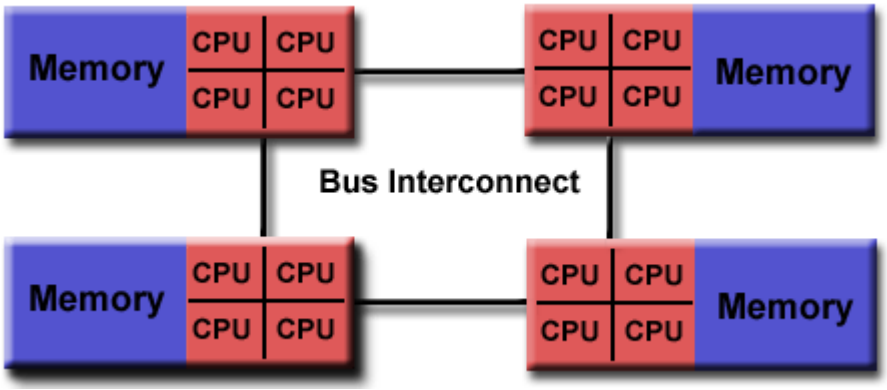

Four summations (instructions)
a + a = 2a
b + b = 2b
c + c = 2c
d + d = 2d
vs.
SIMD one summation (instruction)
a b c d + a b c d = 2a 2b 2c 2d

MIMD Architectures (Shared Memory)

Uniform Memory Access (UMA)

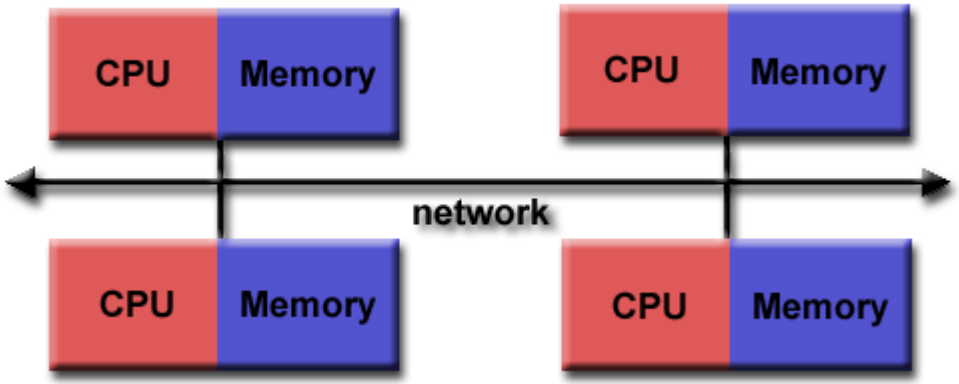


Non-Uniform Memory Access (NUMA)

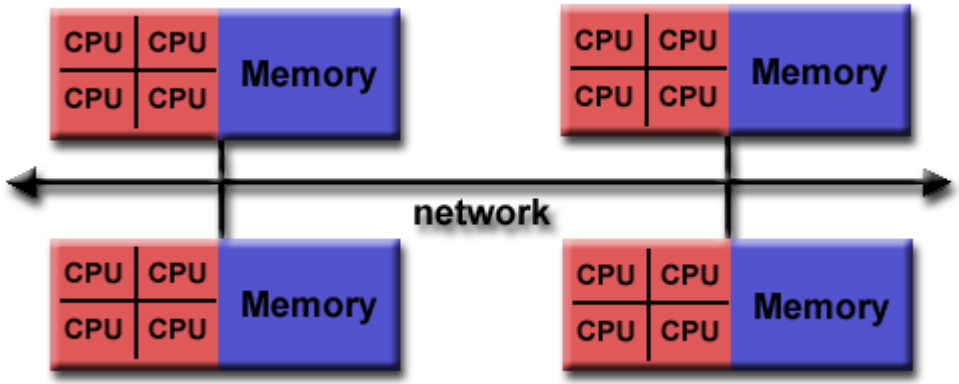


More MIMD Architectures

Distributed Memory



Hybrid Memory



Shared Memory (SM)

Attributes:

- Global memory space
- Each processor will utilize it's own cache for a portion of global memory; consistency of the cache is maintained by hardware

Advantages:

- User-friendly programming techniques (OpenMP and OpenACC)
- Low latency for data sharing between tasks

Disadvantages:

- Global memory space-to-CPU path may be a bottleneck
- Non-Uniform Memory Access
- Programmer responsible for synchronization

Distributed Memory (DM)

Attributes:

- Memory is shared amongst processors through message passing

Advantages:

- Memory scales based on the number of processors
- Access to a processor's own memory is fast
- Cost effective

Disadvantages:

- Error prone; programmers are responsible for the details of the communication
- Complex data structures may be difficult to distribute

Hardware/Software Models

Software and hardware models do not need to match

DM software on SM hardware:

- Message Passing Interface (MPI) - designed for DM Hardware but available o SM systems

SM software on DM hardware

- Remote Memory Access (RMA) included within MPI-3
- Partitioned Global Address Space (PGAS) languages
  - Unified Parallel C (extension to ISO C 99)
  - Coarray Fortran (Fortran 2008)

Difficulties

Serialization causes bottlenecks

Workload is not distributed

Debugging is hard

Serial approach doesn't parallelize

Parallel Performance

Definitions

SerialTime(n) - Time for a serial program to solve a problem with an input of size n

ParallelTime(n,p) - Time for a parallel program to solve the same problem with an input size of n using p processors

SerialTime(n) ≤ ParallelTime(n,1)

Speedup(n,p) = SerialTime(n) / ParallelTime(n,p)

Work(n,p) = p * ParallelTime(n,p)

Efficiency(n,p) = SerialTime(n) / [p * ParallelTime(n,p)]

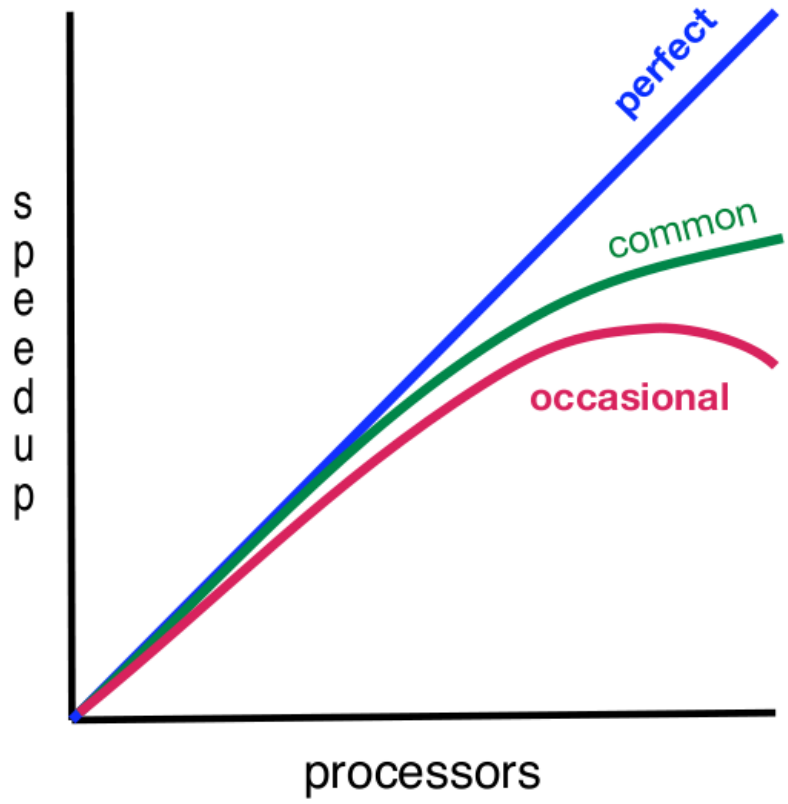
Definitions

Expectations:

- $0 < \text{Speedup} \leq p$
- $\text{SerialWork} \leq \text{ParallelWork} < \infty$
- $0 < \text{Efficiency} \leq 1$

Linear Speedup: Speedup = p, given some restriction on the relationship between n and p

In The Real World



Superlinear Speedup

Superlinear Speedup: Speedup > p, thus Efficiency > 1

Why?

- Parallel computer has p times as much RAM so a higher fraction of program memory is in RAM instead of disk.
- Parallel program used a different algorithm which was not possible with the serial program.

Amdahl's Law

Let f be the fraction of time spent on serial operations

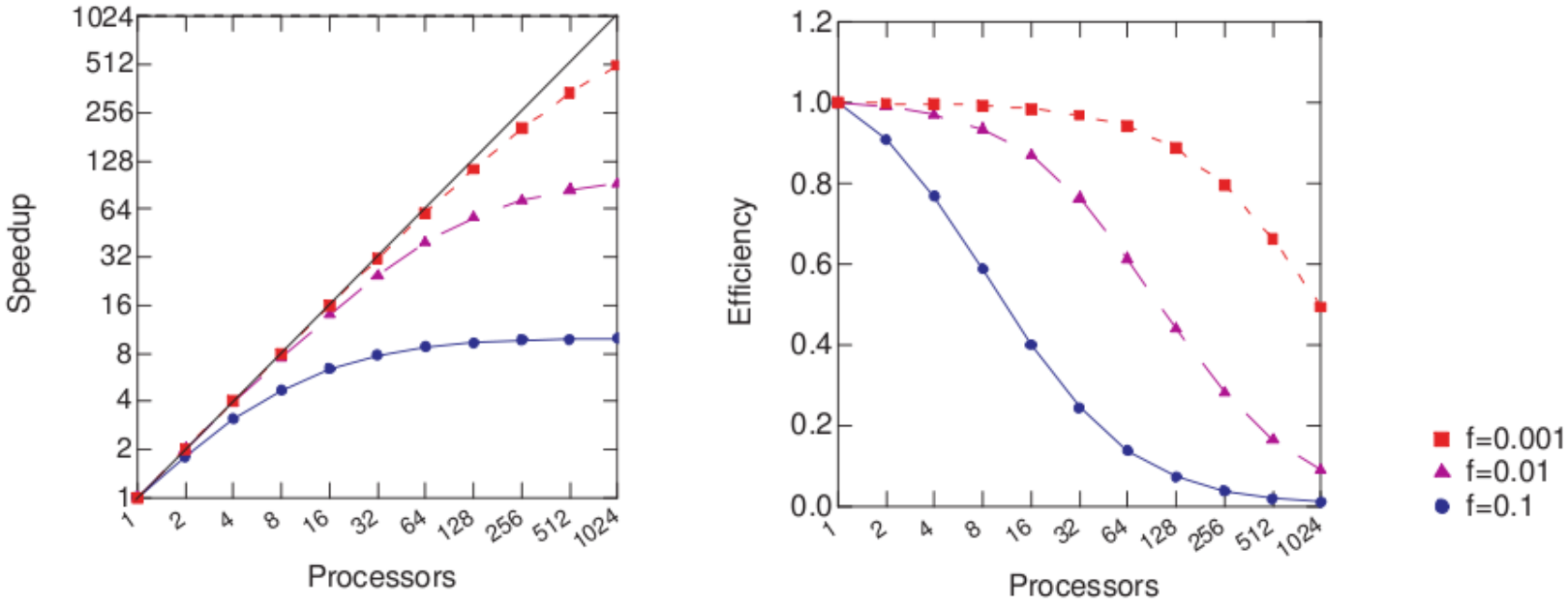
Let ParallelOps = 1 – f and assume linear speedup

For p processors:

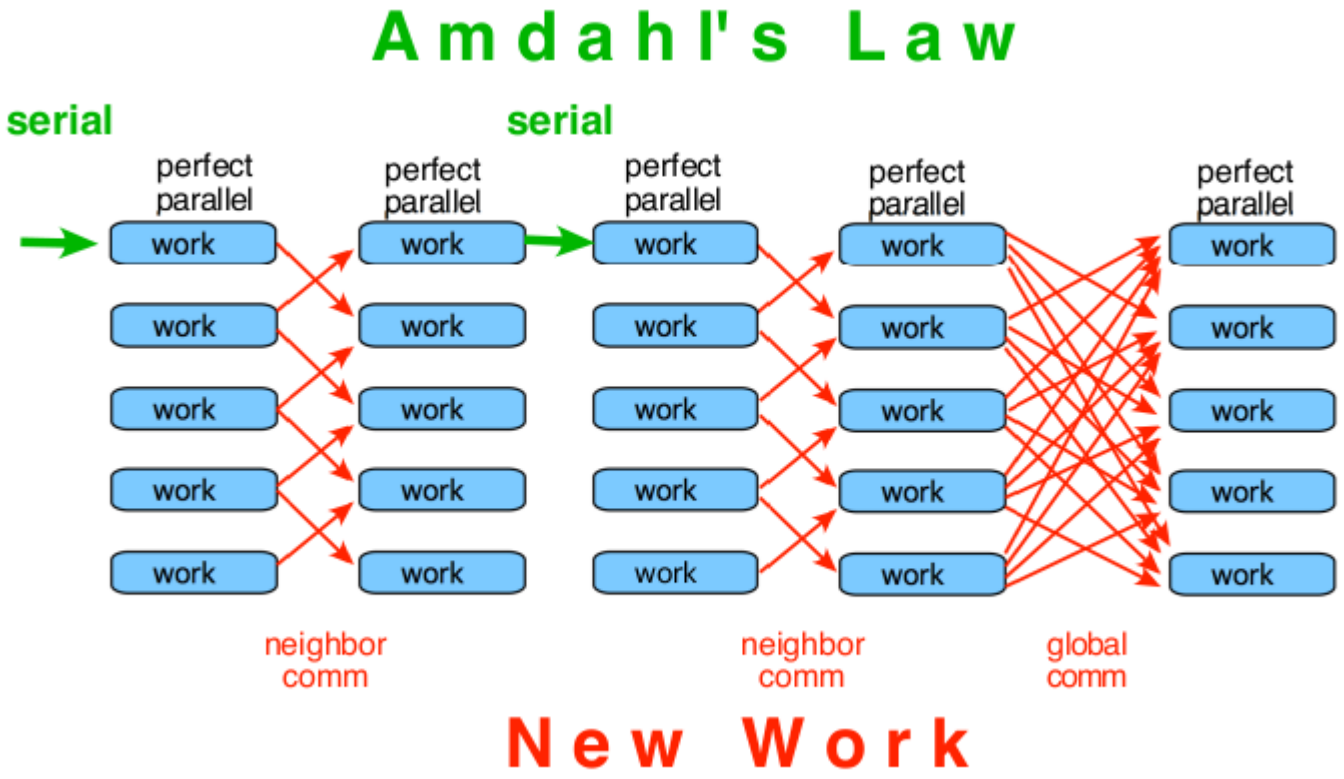
- ParallelTime(p) ≥ SerialTime(p) * [f + (ParallelOps/p)]
- Speedup(p) ≤ 1 / (f + ParallelOps/p)

Thus, Speedup(p) ≤ 1/f, no matter the number of processors used

Maximum Possible Performance



Pessimistic Interpretation

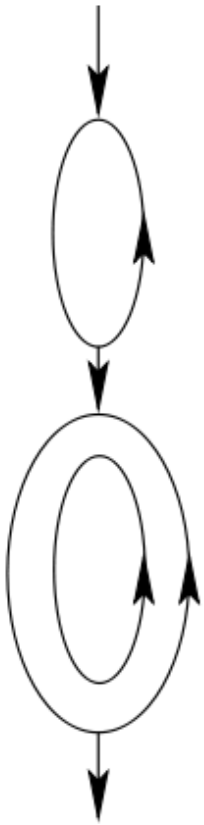


Optimistic Interpretation

We should be more optimistic than Amdahl's Law because:

- Algorithms with much smaller values of *f*
- More time spent in RAM than disk
- Time spent in *f* is a decreasing fraction of the total time as problem size increases

Common Program Structure



Serial, time grows slowly with n

Parallelizable loop, grows with n

Serial, fixed time

Parallelizable loop within loop,
time grows very rapidly with n

Serial, time grows slowly with n

Scaling Strategy

Increase *n* as *p* increases

- Amdahl considered strong scaling where *n* is fixed

Utilize weak scaling

- The amount of data per processor is fixed
- Efficiency can remain high if communication does not increase excessively

Embarrassingly Parallel Jobs

Parameterization

Requirements

- Independent tasks
- Common applications
- One job submission per task

Techniques

- Parameter sweeps (modify parameters mathematically for each task)
- List-driven analysis (modify parameters based on a list of values)

Example: List-Driven Analysis

Problem: Calculate the theoretical performance for the CRI HPC clusters (historically)

Rpeak = Nodes * CoresPerNodes* ClockSpeed * InstructPerCycle

List

Cluster	Nodes	CoresPerNode	ClockSpeed	InstructPerCycle
ibicluster	99	8	2.66e09	4
tarbell	40	64	2.20e09	8
gardner	126	28	2.00e09	16

Base

```
#PBS –N Rpeak.@Cluster
#PBS –l nodes=1:ppn=1
#PBS –l mem=1gb
#PBS –l walltime=20:00

/rpeak -n @Nodes –c @CoresPerNode \
        –s @ClockSpeed -i @InstructPerCycle
```

Shared Memory Parallelism

Shared Memory

All processors can directly access all the memory in the system (though access times may be different due to NUMA)

Software portability – Applications can be developed on serial machines and run on parallel machines without any changes

Latency Hiding – Ability to mask the access latency for memory access, I/O, and communications

Scheduling– Supports system-level dynamic mapping of tasks to processors

Ease of programming – Significantly easier to program (in my opinion using a shared memory model over a distributed memory model

Parallelization Techniques: pthreads

POSIX threads

Standard threading implementation for UNIX-like operating systems (Linux, Mac OS X)

Library that can be linked with C programs

Some compilers include a Fortran version of pthreads

Knowledge of pthreads can easily be transferred to programming other widely used threading specifications (e.g., Java threads)

Matrix-Vector Multiplication (pthreads)

```
include <stdio.h>
include <stdlib.h>
include <pthread.h>

* Global variable:  accessible to all threads */
nt thread_count;

oid* Pth_mat_vect(void* rank);     /* Thread function */

nt main(int argc, char* argv[]) {
   long thread;       /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

* Get number of threads from the command line */
hread_count = strtol(argv[1], NULL, 10);

hread_handles = malloc(thread_count*sizeof(pthread_t));
```

Matrix-Vector Multiplication (pthreads)

```
* Create a thread for each rank */
or (thread=0; thread < thread_count; thread++)
   pthread_create(&thread_handles[thread], NULL,
                          Pth_mat_vect, (void*) thread);

* Join the results from each thread */
or (thread=0; thread < thread_count; thread++)
   pthread_join(thread_handles[thread], NULL);

ree(thread_handles);
eturn 0;
    /* main */
```

Matrix-Vector Multiplication (pthreads)

```
oid* Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank * local_m;
   int my_last_row = (my_rank + 1) * local_m – 1;

   for (i = my_first_row; i <= my_last_row; i++) {
       y[i] = 0.0;
       for (j = 0; j < n; j++)
           y[i] += A[i][j]*x[j];
   }

   return NULL;
    /* Pth_mat_vect */
```

Parallelization Techniques: OpenMP

OpenMP is sort of an HPC standard for shared memory programming

OpenMP version 4.5 released in 2015 and includes accelerator support as an advanced feature

API for thread-based parallelism

Explicit programming model, compiler interprets directives

Based on a combination of compiler directives, library routines, and environment variables

Uses the fork-join model of parallel execution

Available in most Fortran and C compilers

OpenMP Goals

Standardization: standard among all shared memory architectures and hardware platforms

Lean: simple and limited set of compiler directives for shared memor programming. Often significant performance gains using just 4-6 directives in complex applications.

Ease of use: supports incremental parallelization of a serial program, not an all-or-nothing approach.

Portability: supports Fortran, C, and C++

OpenMP Building Blocks

Compiler Directives (embedded in code)

- Parallel regions (PARALLEL)
- Parallel loops (PARALLEL DO)
- Parallel workshare (PARALLEL WORKSHARE)
- Parallel sections (PARALLEL SECTIONS)
- Parallel tasks (PARALLEL TASK)
- Serial sections (SINGLE)
- Synchronization (BARRIER, CRITICAL, ATOMIC, …)
- Data structures (PRIVATE, SHARED, REDUCTION)

Run-time library routines (called in code)

- OMP_SET_NUM_THREADS
- OMP_GET_NUM_THREADS

UNIX Environment Variables (set before program execution)

- OMP_NUM_THREADS

Fork-Join Model

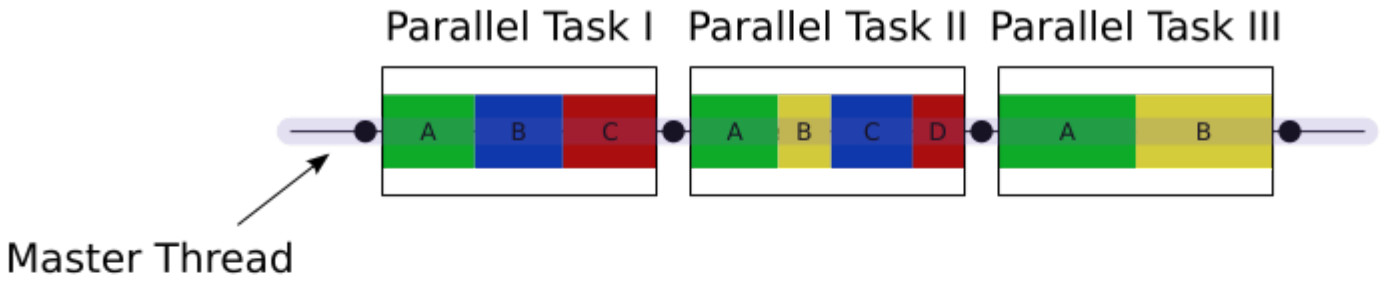
Parallel execution is achieved by generating threads which are executed in parallel

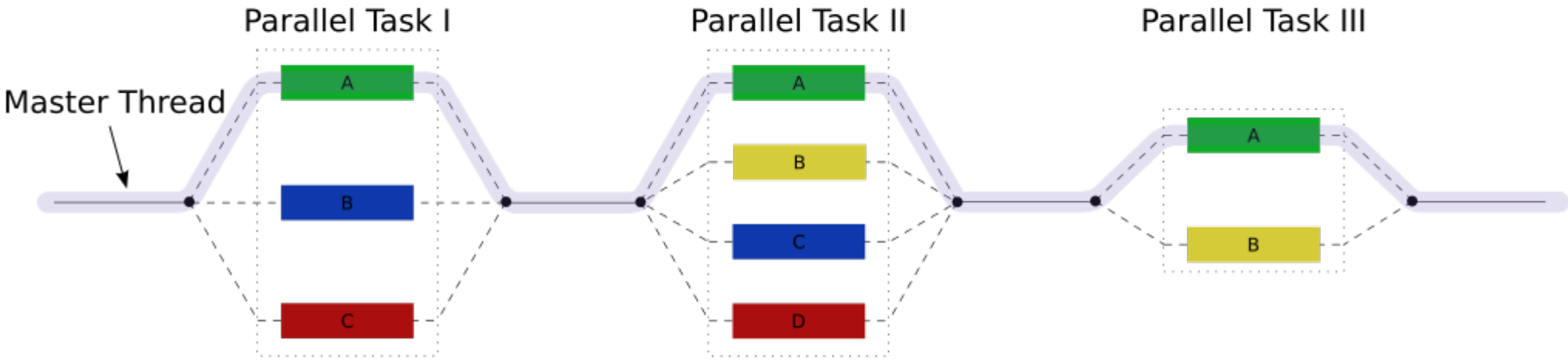
Master thread executes in serial until the first parallel region is encountered

Fork – The master thread created a team of threads which are executed in parallel

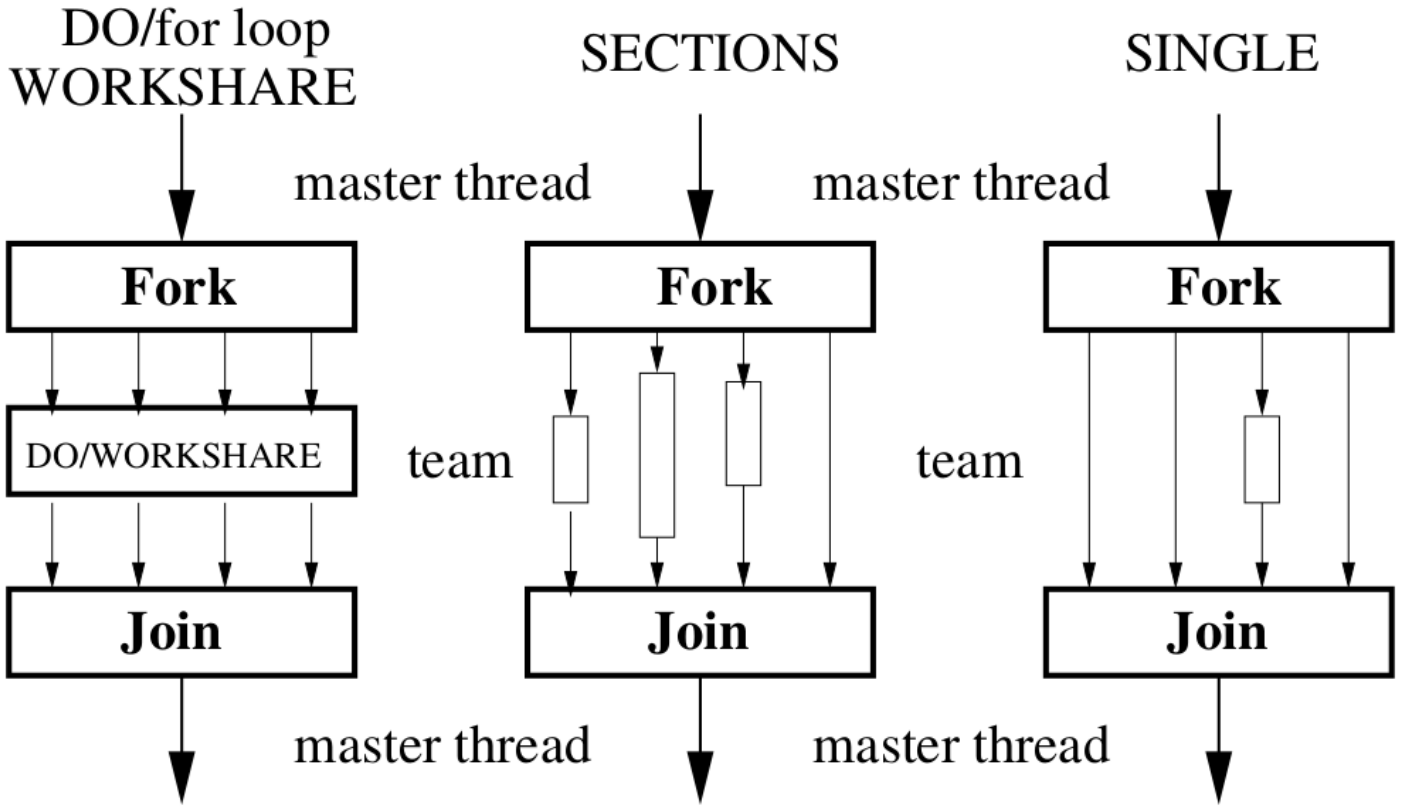
Join – When the team members complete the work, they synchronize and terminate. The master thread continues sequentially.

Fork-Join Model





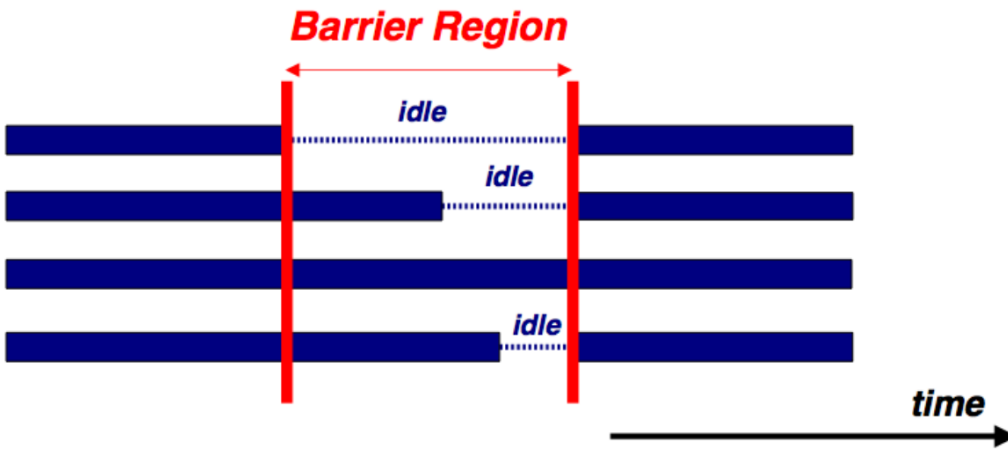
Work-Sharing Constructs



Barriers

Barriers may be needed for correctness

Synchronization degrades performance



OpenMP Notation

OpenMP recognizes the following compiler directives that start with:

- !$OMP (in Fortran)
- #pragma omp (in C/C++)

Parallel Loops

Each thread executes part of the loop

The number of iterations is statically assigned to the threads upon entry to the loop

Number of iterations cannot be changed during executions

Implicit BARRIER at the end of the loop

High efficiency

Parallel Loop Example (Fortran)

```
$OMP PARALLEL

$OMP DO

DO i = 1, n
   a(i) = b(i) + c(i)
END DO

$OMP END DO

$OMP END PARALLEL
```

Parallel Sections

Multiple independent sections can be executed concurrently by separate threads

Each parallel section is assigned to a specific thread which executes the work from start to finish

Implict BARRIER at the end of each SECTION

Nested parallel sections are possible, but impractical

Parallel Sections Example (Fortran)

```
$OMP PARALLEL SECTIONS

$OMP SECTION
DO i = 1, n
   a(i) = b(i) + c(i)
END DO

$OMP SECTION
DO i = 1, k
   d(i) = e(i) + f(i)
END DO

$OMP END PARALLEL SECTIONS
```

Parallel Tasks and Workshares

Parallel Tasks

- Unstructured parallelism
- Irregular problems like:
  - Recursive algorithms
  - Unbounded loops

Parallel Workshares

- Fortran only
- Used to parallelize assignments

Matrix-Vector Multiplication: OpenMP C

```
#include <stdio.h>
#include <stdlib.h>

void Omp_mat_vect(int m, int n, double * restrict a,
                  double * restrict b, double * restrict c) {
   int i, j;

#pragma omp parallel for default (none) \
            shared(m, n, a, b, c) private (i, j)
   for (i = 0; i < m; i++) {
      a[i] = 0.0;
      for (j = 0; j < n; j++)
         a[i] += b[i*n+j] * c[j];
   }   /* End of omp parallel for */
```

Matrix-Vector Multiplication: OpenMP Fortran

```
ubroutine Pth_mat_vect(m, n, a, b, c)

mplicit none

nteger(kind=4) :: m, n
eal(kind=8) :: a(1:m), b(1:m, 1:n), c(1:n)
nteger :: i, j

$OMP PARALLEL DO DEFAULT(NONE) &
$OMP SHARED(m, n, a, b, c) PRIVATE(i, j)
   do i = 1, m
        a(i) = 0.0
        do j = 1, n
            a(i) = a(i) + b(i, j) * c(j)
        end do
   end do
$OMP END PARALLEL DO

eturn
nd subroutine Pth_mat_vect
```

OpenMP Errors

Variable Scoping: Which are shared and which are private

Sequential I/O in a parallel region can lead to an unpredictable order

False sharing: two or more processors access different variables in the same cache line (Performance)

Race Conditions

Deadlock

Summation Example

What is with the following code wrong?

```
um = 0.0
$OMP PARALLEL DO

DO i = 1, n
    sum = sum + a(i)
END DO

$ OMP END PARALLEL DO
```

Summation Example Correction

```
um = 0.0
$OMP PARALLEL DO REDUCTION(+,sum)

DO i = 1, n
   sum = sum + a(i)
ND DO

$ OMP END PARALLEL DO
```

Distributed Memory Parallelism

Message Passing

Communication on distributed memory systems are a significant aspect of performance and correctness

Messages are relatively slow, with startup times (latency) taking thousands of cycles

Once message passing has started, the additional time per byte (bandwidth) is relatively small

Performance on Gardner

Intel Xeon E5-2683 Processor (Haswell)

Processor speed: 2,000 cycles per microsecond (µsec)

16 FLOPs/cycle: 32,000 FLOPs per µsec

MPI message latency = ~2.5 µsec or 80,000 FLOPs

MPI message bandwidth = ~7,000 bytes/µsec = 4.57 FLOPs/byte

Reducing Latency

Reduce the number of messages by mapping communicating entities onto the same processor

Combine messages having the same sender and destination

If processor A has data needed by processor B, have A send it to B, rather than waiting for B to request it. Processor A should send as soon as the data is ready, processor B should read it as late as possible to increase the probability that the data has arrived

Other Issues With Message Passing

Network Congestion

Deadlock

- Blocking: a processor cannot proceed until the message is finished
- With blocking communication, you may reach a point where no processor ca proceed
- Non-blocking communication: easiest way to prevent deadlock; processors can send and proceed before receive is finished

Message Passing Interface (MPI)

International Standard

MPI 1.0 released in 1994

Current version is 3.1 (June 2015)

MPI 4.0 standard in the works

Available on all parallel systems

Interfaces in C/C++ and Fortran

Bindings for MATLAB, Python, R, and Java

Works on both distributed memory and shared memory hardware

Hundreds of functions, but you will need ~6-10 to get started

MPI Basics

Two common MPI functions:

- MPI_SEND() - to send a message
- MPI_RECV() - to receive a message

Function like write and read statements

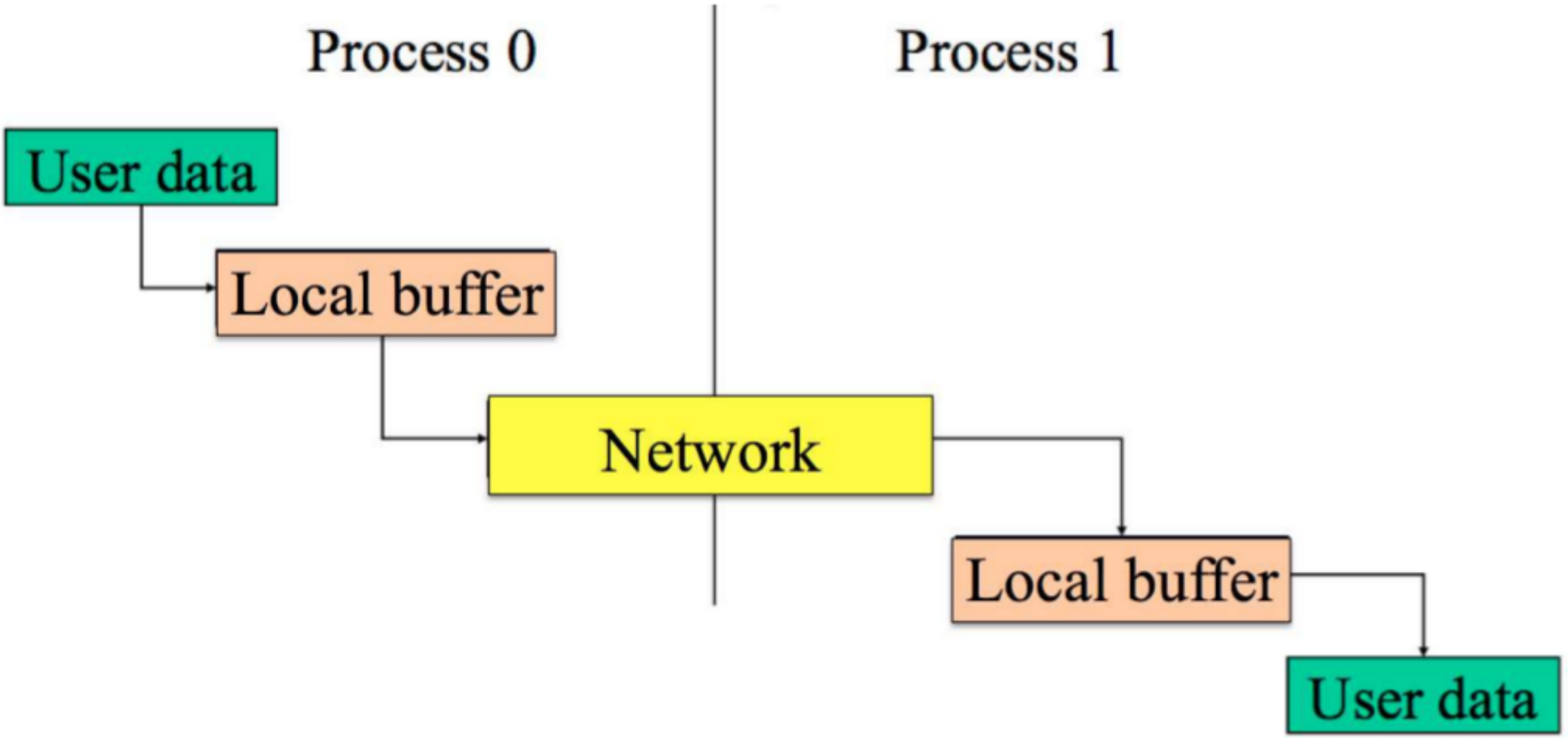
Both are blocking operations

However, a system buffer is used that allows small messages to be non-blocking, but large messages will be blocking

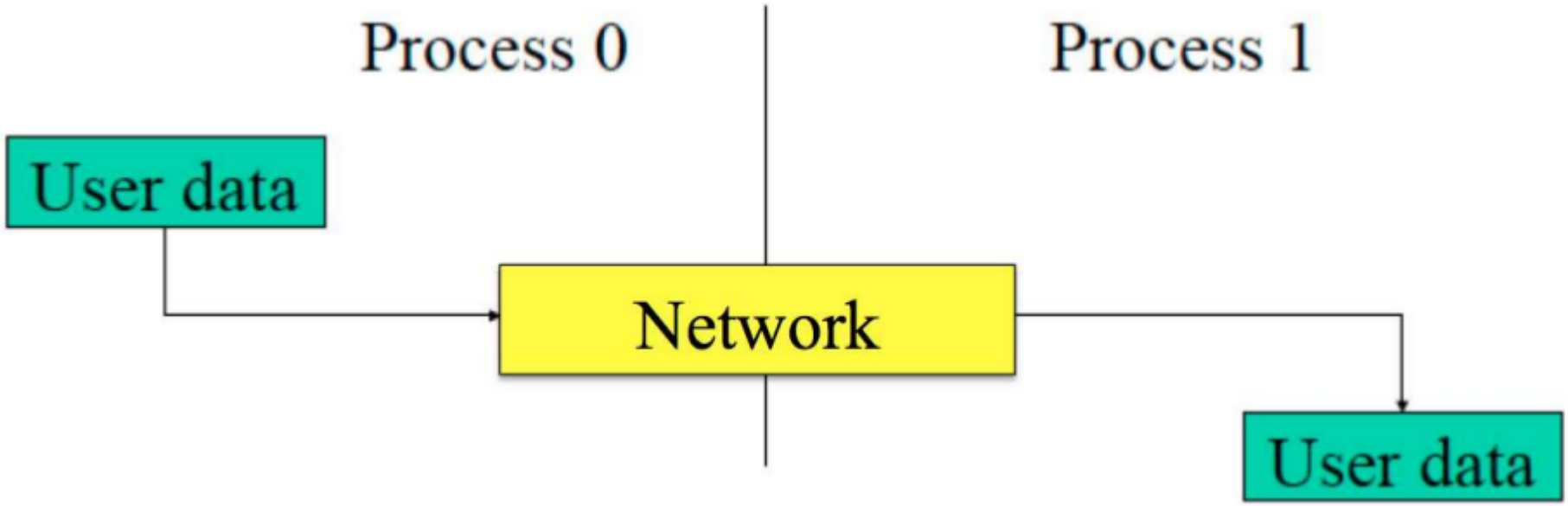
The system buffer is based on the MPI implementation, not the standard

Blocking communication may lead to deadlocks

Small Messages



Large Messages



Deadlocks

Source

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

Avoidance

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

Non-Blocking Communication

Non-Blocking operations

- MPI_ISEND()
- MPI_IRECV()
- MPI_WAIT()

The user can check for data at a later stage in the program without waiting

- MPI_TEST()

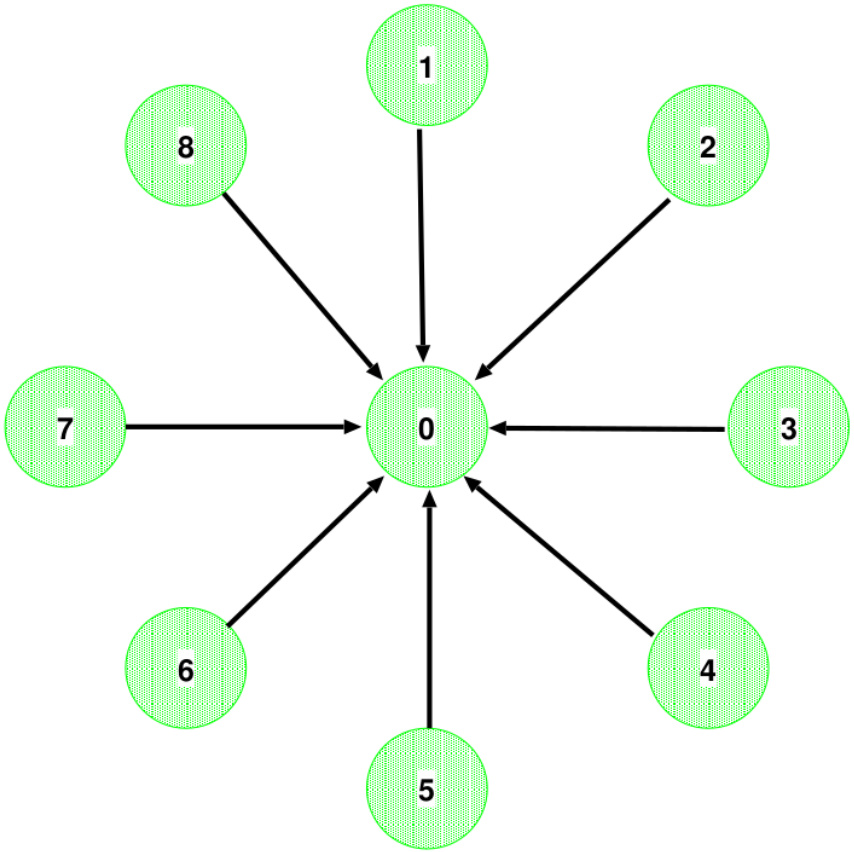
Non-blocking operations will perform better than blocking operations

Possible to overlap communication with computation

MPI Program Template

```
#include "mpi.h"

/* Initialize MPI */
MPI_Init(&argc, &argv);

/* Allow each processor to determine its role */
int num_processor, my_rank;
MPI_Comm_size(MPI_COMM_WORLD, &num_processor);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Send and receive some stuff here */


/* Finalize */
MPI_Finalize()
```

MPI Summation



MPI Summation

```
* Send and receive some stuff here */
f (my_rank == 0) {
   sum = 0.0;
   for (source=1; source < num_procs; source++) {
        MPI_RECV(&value, 1, MPI_FLOAT, source, tag,
                        MPI_COMM_WORLD, &status);
        sum += value;
   }

lse {
   MPI_SEND(&value, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
```

A More Efficient Receive

In the previous example, rank 0 received the messages in order based on ranks.

Thus, if processor 2 was delayed, then processor 0 was also delayed

Make the following modification:

```
MPI_RECV(&value, 1, MPI_FLOAT, MPI_ANY_SOURCE, tag,
                MPI_COMM_WORLD, &status);
```

Now processor 0 can process messages as soon as they arrive

MPI Reduction

Like OpenMP, MPI also has reduction operations that can be used for tasks such as summation

Reduction is a type of collective communication

```
MPI_REDUCE(&value, &sum, 1, MPI_FLOAT, MPI_SUM,
                    0, MPI_COMM_WORLD);
```

Reduction operations:

- MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD
- MPI_LAND, MPI_LOR

Collective Communication

Most frequently invoked MPI routines after send and receive

Improve clarity, run faster than send and receive, and reduce the likelihood of making an error

Examples

- Broadcast (MPI_Bcast)
- Scatter (MPI_Scatter)
- Gather (MPI_Gather)
- All Gather (MPI_Allgather)

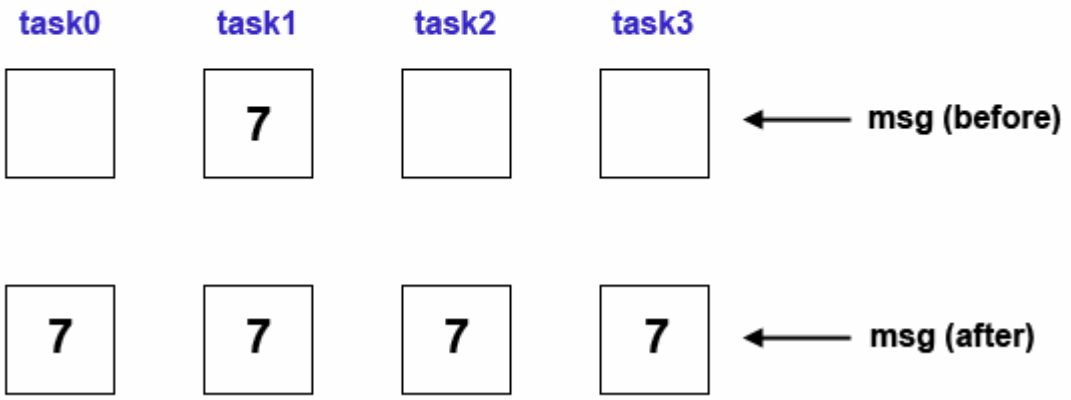
Broadcast

MPI_Bcast

Broadcasts a message from one task to all other tasks in communicator

```
count = 1;
source = 1;
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

task1 contains the message to be broadcast



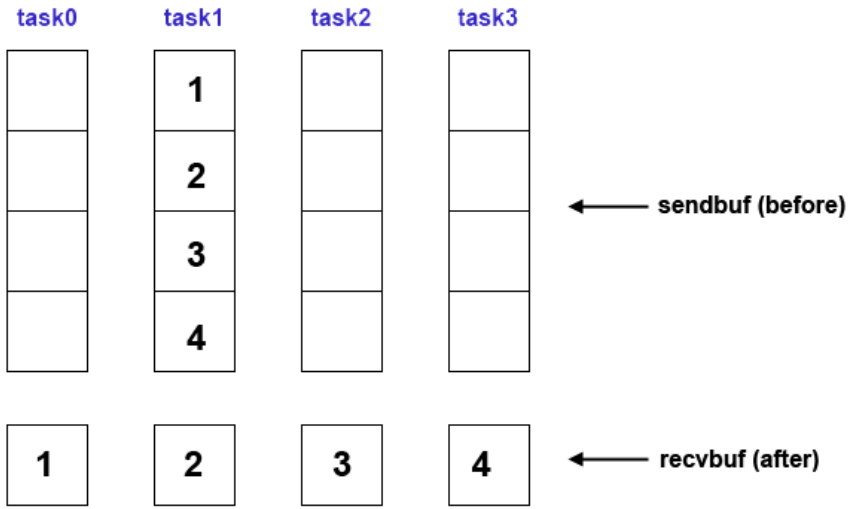
Scatter

MPI_Scatter

Sends data from one task to all other tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT
            recvbuf, recvcnt, MPI_INT
            src, MPI_COMM_WORLD);
```

task1 contains the data to be scattered



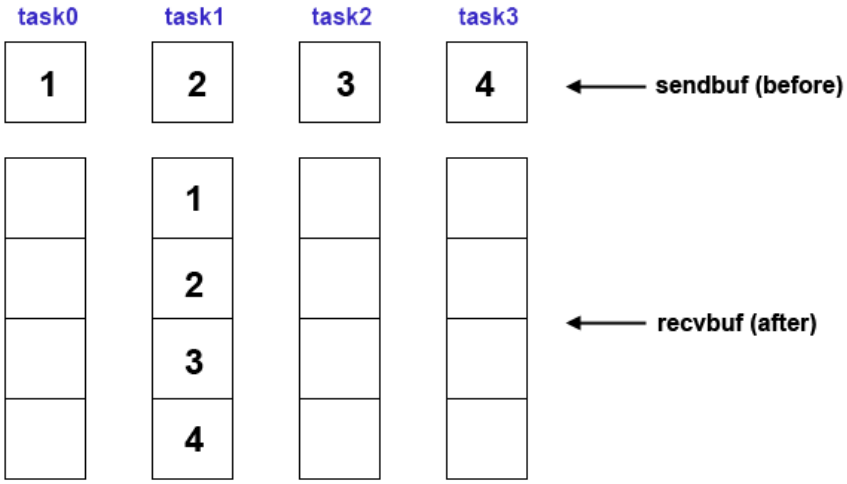
Gather

MPI_Gather

Gathers data from all tasks in communicator to a single task

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Gather(sendbuf, sendcnt, MPI_INT
           recvbuf, recvcnt, MPI_INT
           src, MPI_COMM_WORLD);
```

message will be gathered into task1

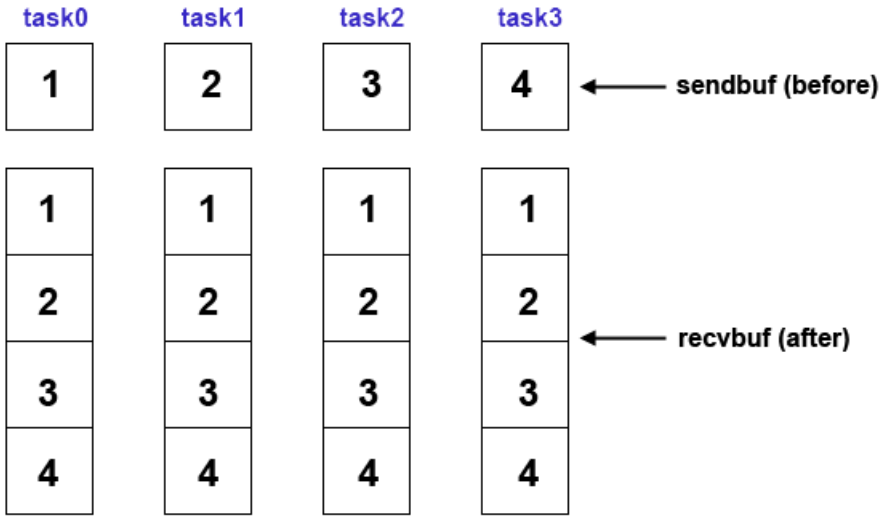


All Gather

MPI_Allgather

Gathers data from all tasks and then distributes to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT
              recvbuf, recvcnt, MPI_INT
              MPI_COMM_WORLD);
```



MPI Data Types

Primitive data types (correspond to those found in the underlying language)

- Fortran
  - MPI_CHARACTER
  - MPI_INTEGER
  - MPI_REAL, MPI_DOUBLE_PRECISION
  - MPI_LOGICAL
- C/C++ (include many more variants than Fortran)
  - MPI_CHAR
  - MPI_INT, MPI_UNSIGNED, MPI_LONG
  - MPI_FLOAT, MPI_DOUBLE
  - MPI_C_BOOL

MPI Data Types

Derived data types are also possible

- Vector – data separated by a constant stride
- Contiguous – data separated by a stride of 1
- Struct – mixed types
- Indexed – array of indices

MPI Vector

MPI_ Type_vector

count = 4; blocklength = 1; stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT, &columntype);

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

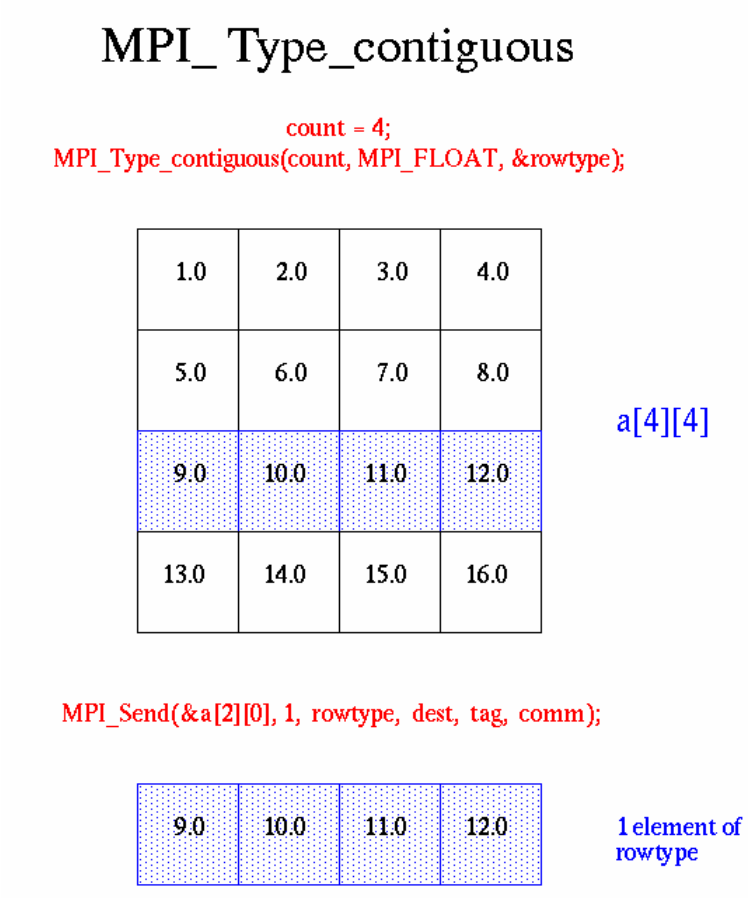
a[4][4]

MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);

2.0	6.0	10.0	14.0

1 element of columntype

MPI Contiguous



MPI Struct

MPI_ Type_struct

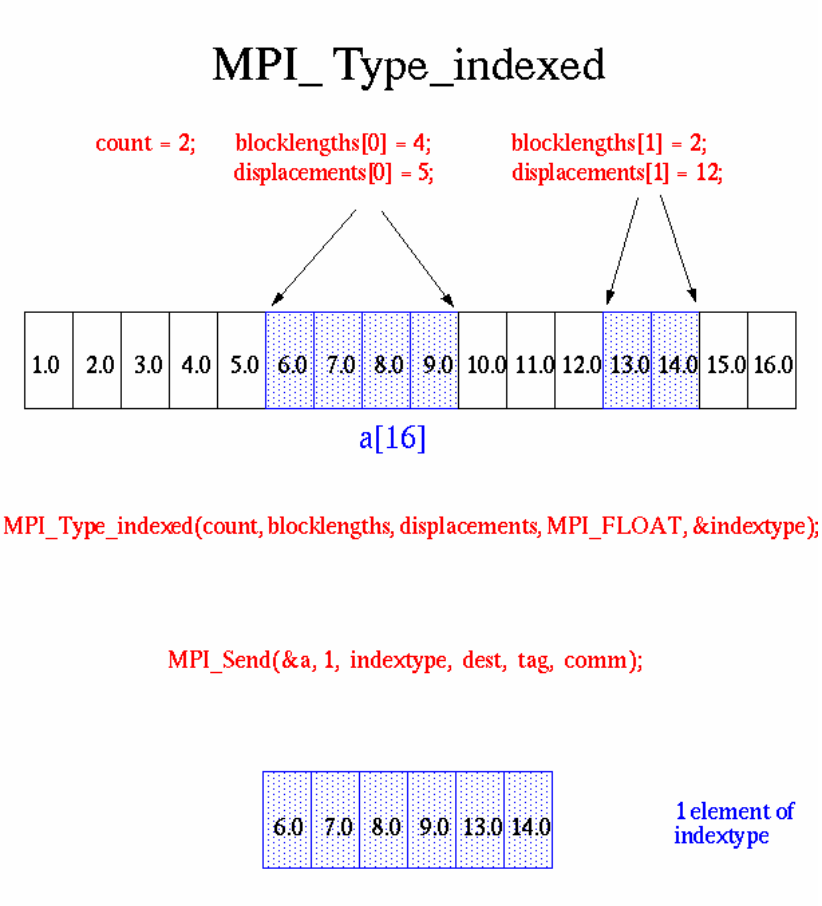
```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2; oldtypes[0] = MPI_FLOAT;   oldtypes[1] = MPI_INT
           offsets[0] = 0;            offsets[1] = 4 * extent;
           blockcounts[0] = 4;        blockcounts[1] = 2;
```

particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

MPI Indexed


MPI_ Type_indexed
count = 2;
blocklengths[0] = 4;
displacements[0] = 5;
blocklengths[1] = 2;
displacements[1] = 12;
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0
a[16]
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
MPI_Send(&a, 1, indextype, dest, tag, comm);
6.0 7.0 8.0 9.0 13.0 14.0
1 element of indextype

MPI Synchronization

Implicit synchronization

- Blocking communication
- Collective communication

Explicit synchronization

- MPI_Wait
- MPI_Waitany
- MPI_Barrier

Remember, synchronization can hinder performance

Additional MPI Features

Virtual topologies

User-created communicators

User-specified derived data types

Parallel I/O

One-sided communication (MPI_Put, MPI_Get)

Non-blocking collective communication

Remote Memory Access

Vectorization

Accelerators

Principle: Split an operation into independent parts and execute the parts concurrently in an accelerator (SIMD)

```
DO I = 1, 1000
   A(I) = B(I) + C(I)
END DO
```

Accelerators like Graphics Processing Units are specialized for SIMD operations

Popularity

In 2012, 13% of the performance share of the Top 500 Supercomputers was provided through accelerators

Today, the performance share for accelerators is ~38%

Trend: Build diverse computing platforms with multiprocessors, SMP nodes, and accelerators

Caveat: Difficult to use heterogenous hardware effectively

Modern Accelerators

NVidia Volta

7.5 TFLOPs double precision

30 TFLOPs half precision

5120 cores

16 GB RAM

900 GB/s memory bandwidth

Intel Xeon Phi Knight's Landing

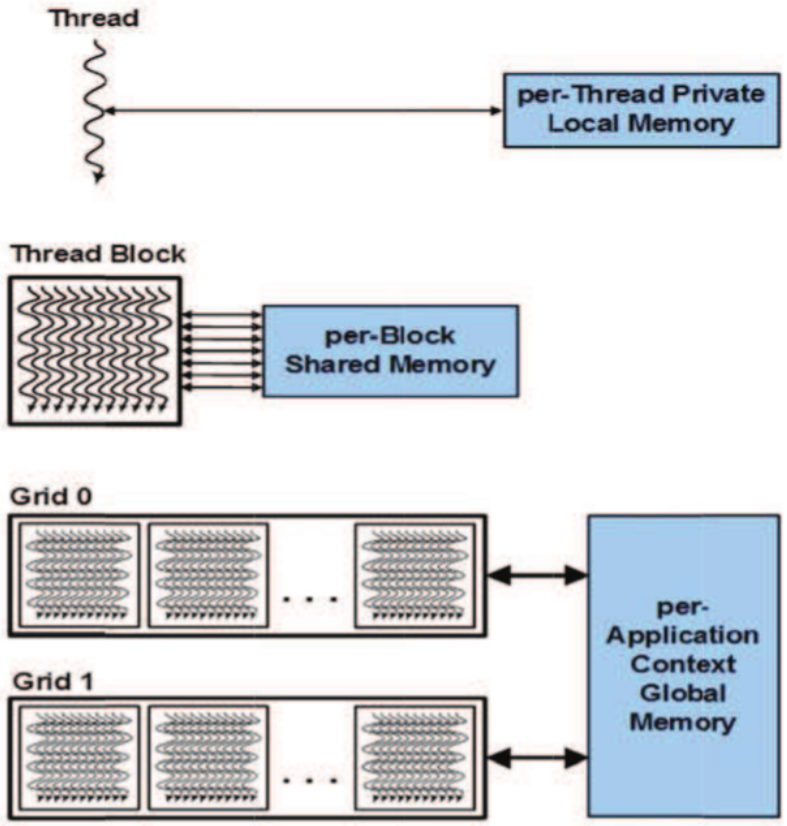
- 3 TFLOPs double precision
- 72 cores (288 threads)
- 16 GB RAM
- 384 GB/s memory bandwidth
- Includes both a scalar and a vector unit

NVidia Architecture

SIMT: Single Instruction, Multiple Thread (Similar to SIMD)

Thread blocks are executed by a warp of 32 cores

Performance is dependent on memory alignment and eliminating shared memory access



Programming GPUs

Low-Level Programming

- Proprietary – NVidia's CUDA
- Portable – OpenCL

High-Level Programming

- OpenACC
- OpenMP 4.5
- Allows for a single code base that may be compiled on both multicore and GPUs

CUDA Matrix Multiplication

```
nt main(void) {
  int *a, *b, *c;                         // CPU copies
  int *dev_a, *dev_b, *dev_c;       // GPU copies

  /* Allocate memory on GPU */
  arraysize = n*sizeof(int);
  cudaMalloc( (void**)\&dev_a, arraysize);
  cudaMalloc( (void**)\&dev_b, arraysize);
  cudaMalloc( (void**)\&dev_c, sizeof(in));

  use malloc for a, b, c
  initialize a, b

  /* Move a and b to the GPU */
 cudaMemcpy(dev_a, a, arraysize, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, arraysize, cudaMemcpyHosttoDevice);

  /* Launch GPU kernel */
  dot <<< n/threads_per_block,threads_per_block >>>(dev_a,dev_b,dev_c);

  /*  Copy results back to CPU
  cudaMemcpy(c, dev_c, sizeof(int), cudaMemcpyDeviceToHost)

  return 0;
```

CUDA Matrix Multiplication

```
global_ void dot(int *a, int *b, int *c) {
  _shared_ int temp[threads_per_block];
  int index = threadIdx.x + blockIdx.x + blockDim.x;

  /* Convert warp index to global */
  temp[threadIdx.x] = a[index] * b[index];

  /* Avoid race condition on sum within block */
  _syncthreads();

  if (threadIdx.x == 0) {
      int sum = 0;
      for (int i = 0; i < threads_per_block; i++)
          sum += temp[i];

      /* add to global sum */
      atomicAdd(c, sum)
  }
```

OpenACC and OpenMP

Share the same model of computation

Host-centric – host device offloads code regions and data to accelerators

Device – has independent memory and multiple threads

Mapping clause – Defines the relationship between memory on the host device and memory on the accelerator

OpenACC Matrix Multiplication

```
nt main(int argc, char** argv) {
  int n = atoi(argv[1]);
  float *a = (float *)malloc(sizeof(float) * n * n);
  float *b = (float *)malloc(sizeof(float) * n * n);
  float *c = (float *)malloc(sizeof(float) * n * n);

  #pragma acc data copyin(a[0:n*n], b[0:n*n]), copy(c[0:n*n]) {
      int i, j, k;

      #pragma acc kernels
      for (i = 0; i < n; i++) {
          for (j = 0; j < n; j++) {
              a[i*n+j] = (float) i + j;
              b[i*n+j] = (float) i – j;
              c[i*n+j] = 0.0;
          }
      }
```

OpenACC Matrix Multiplication

```
    #pragma acc kernels
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            for (k = 0; k < n; k++) {
                c[i * n + j] += a[i * n + k] * b[k * n + j];
            }
        }
    }
}
```

Speedup?

Some problems can achieve a speedup of 10x-100x by offloading some of the work to a GPU (e.g. linpack, matrix multiplication)

In reality:

- Only a small fraction of applications can utilize a GPU effectively
- The average speedup of those applications is ~2x-4x
- Optimizing your code using multicore technologies is probably a better use of your time

Problem Areas

Branching can cut your performance by 50% since instructions need to be issued for both branches

Memory bandwidth on GPUs is poor

- Need to make sure your operands are used more than once
- For the NVidia Volta, all operands need to be used 67x to achieve peak FLOPs

Accelerator Trends

Intel has canceled Knight's Landing and Knight's Hill; future Xeon Phi status is unknown

GPUs are becoming less rigidly SIMD and including better memory bandwidth

More programmers moving from low-level programming like CUDA to high-level directives

PGI Compiler was purchased by NVidia in 2013; cut support for OpenCL

OpenACC and OpenMP still have not merged

Efficiency of GPUs is still problematic and the programming model is a moving target

New Motivation: Deep Learning

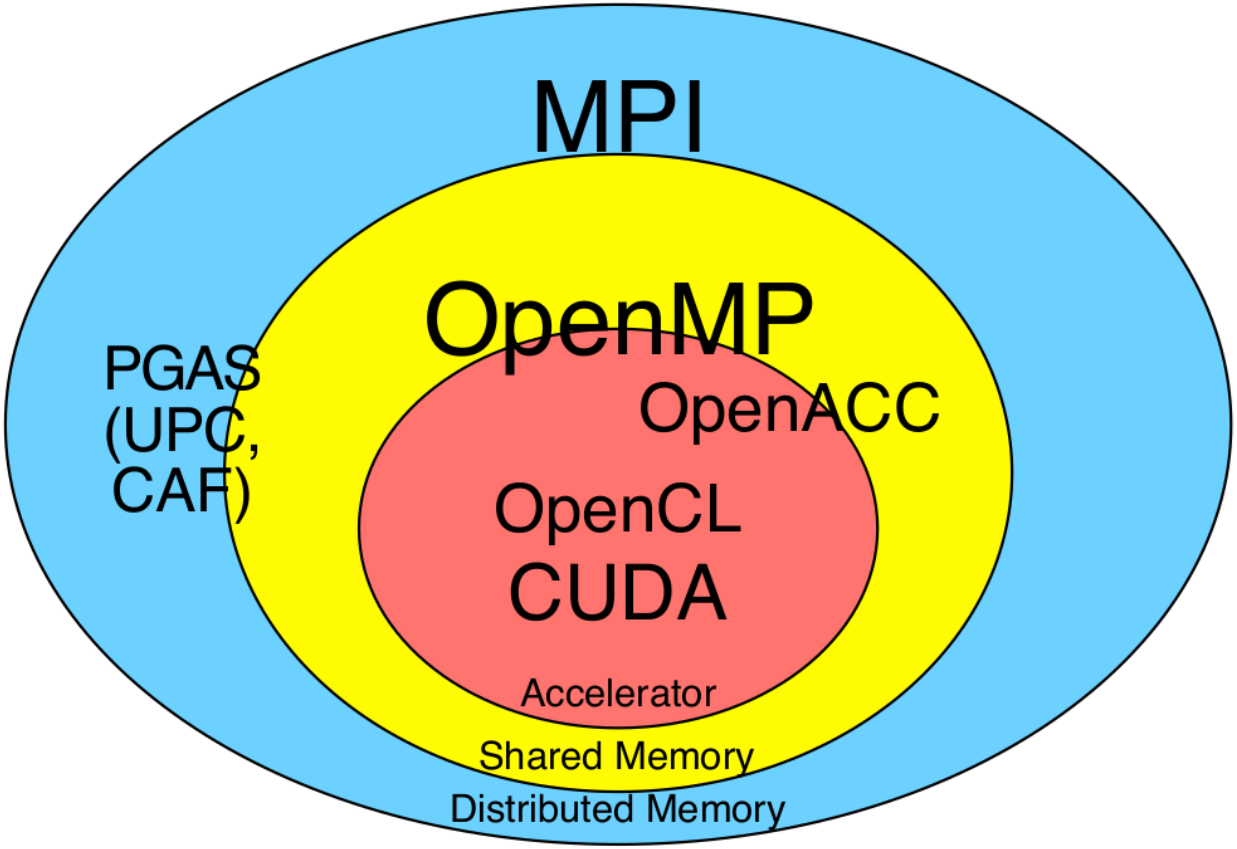
Hybrid Computing

Current Trend

Accelerator-based machines are grabbing the high rankings on the Top 500, but clusters with standard cores are still most important economically

Economics dictates distributed memory system of shared memory boards with multicore commodity chips, perhaps with some form of accelerator distributed sparingly

MPI+X


MPI
OpenMP
OpenACC
OpenCL
CUDA
PGAS
(UPC,
CAF)
Accelerator
Shared Memory
Distributed Memory

Conclusion

Topics for Possible Future Workshops

Scientific Computing (C++ or Fortran or Both)

Parallel Computing Theory (e.g., Load balancing, measuring performance)

Embarrassingly Parallel

pthreads

OpenMP

MPI

OpenACC

CUDA

Parallel Debugging/Profiling