Message Passing Architecture
Message Passing Multiprocessors

Collection of N nodes

- Node = CPU with cache and private memory space
- Node i has address space 0, ..., A_i − 1
- No shared memory locations

Processes communicate by exchange of structured messages

Switching fabric — network
Message Passing Systems

Interprocessor communication
Send / receive transactions
Message Passing Interface (MPI)
  API for message passing parallel programming
  Transaction primitives between processor and interprocessor network
  Implementation depends on OS and hardware

Interprocessor network performance
General computer network
Communication protocol and message types
Routing, network access, segmentation/reassembly (SAR)
Data requests and transfers

Scalable to large systems
Depends on network capacity
No geographical restriction
Grows naturally into cluster computing
Message Passing Issues

**Basic message types**

- Send data
- Send request (for data)
- Respond to request (data or status)
- Send system command

**Structured messages**

- Header
  - Source ID + destination ID + timestamp
- Content
  - Data / request / response
- Message management $\Rightarrow$ overhead

**Sequential consistency**

- No shared memory $\Rightarrow$ no snooping $\Rightarrow$ no snooping overhead
- Code expects specific content from specific source
- Message header + content $\Rightarrow$ consistent write order
Message Passing Example — Scalar Product

Compute $\sum_{i=0}^{3} a[i] * b[i]$ on data pre-distributed to nodes 0 - 3

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $R_a$, a</td>
<td>load $R_a$, a</td>
<td>load $R_a$, a</td>
<td>load $R_a$, a</td>
</tr>
<tr>
<td>load $R_b$, b</td>
<td>load $R_b$, b</td>
<td>load $R_b$, b</td>
<td>load $R_b$, b</td>
</tr>
<tr>
<td>$R_a \leftarrow R_a * R_b$</td>
<td>$R_a \leftarrow R_a * R_b$</td>
<td>$R_a \leftarrow R_a * R_b$</td>
<td>$R_a \leftarrow R_a * R_b$</td>
</tr>
<tr>
<td>send P1, $R_a$</td>
<td>recv P0, $R_b$</td>
<td>send P3, $R_b$</td>
<td>recv P2, $R_b$</td>
</tr>
<tr>
<td></td>
<td>$R_a \leftarrow R_a + R_b$</td>
<td>$R_a \leftarrow R_a + R_b$</td>
<td>$R_a \leftarrow R_a + R_b$</td>
</tr>
<tr>
<td></td>
<td>send P3, $R_a$</td>
<td></td>
<td>store p, $R_a$</td>
</tr>
</tbody>
</table>

**Message overhead**

Source or destination

Time of creation

**Sequential consistency guaranteed by message overhead**

P3 distinguishes P1 data from P2 data by source ID

No data hazard
Blocking versus Non-Blocking Models

Program must pair
Send instruction in one thread
Receive instruction in another thread

Synchronous (blocking) model
Threads synchronize on send/receive pair
Send and receive instructions block in code
Receive waits for RTS
Send waits for CTS
Creates synchronization barrier at transfer

Asynchronous (non-blocking) model
Sender execute send instruction and continues
Message buffered at sender until sent through network
Message buffered at receiver until needed
Receiver accepts message when executing receive instruction
### Some MPI Environment Messages

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI_Init (&amp;argc,&amp;argv)</strong></td>
<td>Initialize MPI execution environment</td>
</tr>
<tr>
<td></td>
<td>Broadcast command line arguments to all processes</td>
</tr>
<tr>
<td><strong>MPI_COMM_WORLD</strong></td>
<td>Environment variable</td>
</tr>
<tr>
<td></td>
<td>Lists MPI-aware processes</td>
</tr>
<tr>
<td><strong>MPI_Comm_size (comm,&amp;size)</strong></td>
<td>Returns number of processes in group</td>
</tr>
<tr>
<td><strong>MPI_Comm_rank (comm,&amp;rank)</strong></td>
<td>Returns process number of calling process</td>
</tr>
<tr>
<td><strong>MPI_Finalize ()</strong></td>
<td>Terminates MPI execution environment</td>
</tr>
<tr>
<td></td>
<td>Last MPI routine in every MPI program</td>
</tr>
</tbody>
</table>

Ref: [https://computing.llnl.gov/tutorials/mpi/](https://computing.llnl.gov/tutorials/mpi/)
## Some MPI Point-to-Point Messages

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI_Send</strong>(<em>buffer</em>, <em>count</em>, <em>data_type</em>, <em>destination_task</em>, <em>id_tag</em>, <em>comm</em>)</td>
<td>Blocking send</td>
</tr>
<tr>
<td></td>
<td><em>id_tag</em> — defined by user to distinguish specific message</td>
</tr>
<tr>
<td></td>
<td><em>comm</em> — identifies group of related tasks (usually <strong>MPI_COMM_WORLD</strong>)</td>
</tr>
<tr>
<td><strong>MPI_Recv</strong>(<em>buffer</em>, <em>count</em>, <em>data_type</em>, <em>source</em>, <em>id_tag</em>, <em>comm</em>, &amp;<em>status</em>)</td>
<td>Blocking receive</td>
</tr>
<tr>
<td></td>
<td><em>status</em> — collection of error flags</td>
</tr>
<tr>
<td><strong>MPI_Isend</strong>(<em>buffer</em>, <em>count</em>, <em>data_type</em>, <em>destination_task</em>, <em>id_tag</em>, <em>comm</em>, &amp;<em>request</em>)</td>
<td>Non-blocking send</td>
</tr>
<tr>
<td></td>
<td><em>request</em> — system returns request number for subsequent synchronization</td>
</tr>
<tr>
<td><strong>MPI_Irecv</strong>(<em>buffer</em>, <em>count</em>, <em>data_type</em>, <em>source_task</em>, <em>id_tag</em>, <em>comm</em>, &amp;<em>request</em>)</td>
<td>Non-blocking receive</td>
</tr>
<tr>
<td><strong>MPI_Wait</strong>(&amp;<em>request</em>, &amp;<em>status</em>)</td>
<td>Blocks until a specified non-blocking send or receive operation has completed</td>
</tr>
</tbody>
</table>

Ref: https://computing.llnl.gov/tutorials/mpi/
Some Collective Communication Messages

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Barrier(comm)</td>
<td>Each task reaching MPI_Barrier call blocks until all tasks in group reach same MPI_Barrier</td>
</tr>
<tr>
<td>MPI_Bcast(buffer,count,data_type,root,comm)</td>
<td>Sends broadcast message from process root to all other processes in group</td>
</tr>
<tr>
<td>MPI_Scatter(sendbuf,sendcnt,sendtype,recvbuf,recvcnt,recvtype,root,comm)</td>
<td>Distributes distinct messages from a single root task to each task in group</td>
</tr>
<tr>
<td>MPI_Gather(sendbuf,sendcnt,sendtype,recvbuf,recvcount,recvtype,root,comm)</td>
<td>Gathers distinct messages from each task in group to a single destination task</td>
</tr>
<tr>
<td>MPI_Reduce(sendbuf,recvbuf,count,data_type,operation,root,comm)</td>
<td>Applies a reduction operation on all tasks in group and places result in one root task</td>
</tr>
</tbody>
</table>

Ref:  https://computing.llnl.gov/tutorials/mpi/
**Scatter and Gather**

---

**Scatter**

- **Send buffer**
- **Destination buffers**

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Gather**

- **Destination buffer**
- **Send buffers**

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Destination buffers**

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

---

**Send buffers**

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>
Reduce

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Send buffers

Reduce: ADD

<table>
<thead>
<tr>
<th>Task 0</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Destination buffer
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
  char message[20];
  int myrank; /* myrank = this process number */
  MPI_Status status; /* MPI_Status = error flags */
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  /* MPI_COMM_WORLD = list of active MPI processes */
  if (myrank == 0) /* code for process zero */
  {
    strcpy(message,"Hello, there");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else /* code for process one */
  {
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    printf("received :%s:\n", message);
  }
  MPI_Finalize();
}
## Send/Receive Instructions

### Send instruction

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>Instruction name</td>
</tr>
<tr>
<td>message</td>
<td>Pointer to message buffer</td>
</tr>
<tr>
<td>strlen(message)+1</td>
<td>Length of message</td>
</tr>
<tr>
<td>MPI_CHAR</td>
<td>Data type in message</td>
</tr>
<tr>
<td>1</td>
<td>Destination process</td>
</tr>
<tr>
<td>99</td>
<td>Message identifier (tags specific message from source)</td>
</tr>
<tr>
<td>MPI_COMM_WORLD</td>
<td>List of running MPI-aware processes</td>
</tr>
</tbody>
</table>

### Receive instruction

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Recv</td>
<td>Instruction name</td>
</tr>
<tr>
<td>message</td>
<td>Pointer to message buffer</td>
</tr>
<tr>
<td>20</td>
<td>Maximum length of message</td>
</tr>
<tr>
<td>MPI_CHAR</td>
<td>Data type in message</td>
</tr>
<tr>
<td>0</td>
<td>Source process</td>
</tr>
<tr>
<td>99</td>
<td>Message identifier (filters on tag)</td>
</tr>
<tr>
<td>MPI_COMM_WORLD</td>
<td>List of running MPI-aware processes</td>
</tr>
<tr>
<td>&amp;status</td>
<td>Error flags</td>
</tr>
</tbody>
</table>
Scalar Product with MPI Constructs

Compute $\sum_{i=0}^{3} a[i] * b[i]$ on 4 nodes

/* scatter data from root node 0 */
/* each node receives 1 component of a and one of b */
MPI_Scatter(a,1,MPI_INT,a,1,MPI_INT,0,MPI_COMM_WORLD)
MPI_Scatter(b,1,MPI_INT,b,1,MPI_INT,0,MPI_COMM_WORLD)

/* each node calculates */
p = a * b

/* OS for root node adds 1 integer from each node and places sum in root 0 */
MPI_REDUCE(p,p,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)
Implementation Issues for Processor

**Processor directly supporting message passing**

Synchronous model
- Message send/receive port
- Hardware blocking on port access

Asynchronous model
- Message send/receive port or equivalent
- Internal send/receive buffer for messages

**Interconnect support for message passing**

No direct support in processor
- Interconnect buffers messages for processor

Synchronous model
- Processor blocks on send/receive instruction
- Interconnect sends/receives messages

Asynchronous model
- Interconnect buffers and sends messages for processor
- Interconnect buffers received messages and interrupts processor
Message Passing Support in Alpha 21364

Ref: Kevin Krewell, "Alpha EV7 Processor", Microprocessor Report, 2002
Message Passing Multiprocessor Configurations

Four-Processor Configuration

Sixteen-Processor 2D Torus Configuration
Cluster Computing

Large message passing distributed memory system

Exploits MPI scalability
Up to millions of nodes

Typical node

Standard workstation

Node-to-node scale

Physical bus
Crossbar switch
LAN
WAN

Interconnection network

LAN / WAN

MPI_COMM_WORLD includes network addresses
Benefits of Clusters

**Scalability in parallel computing**

- Message passing parallel algorithms scale to large number of nodes
- Typical communication / computation ratio low to medium
- Less sensitive to latency than shared memory
- Very high scalability for thread-replicating servers
  - Web servers
  - Database access servers
  - On-line transaction processing (OLTP)

**Availability**

- Nodes provide backup to each other
- On device failure load transferred to another node within cluster
- Transparent to user

**Manageability**

- Single system image $\Rightarrow$ simple system restoration
- Automatic load balancing
Socket API maps endpoints to socket numbers

Socket number identifies connection to remote endpoint
   Endpoint = IP address:port
Client — socket identifies connection to server endpoint
Server — socket identifies connection from client endpoint

Systems with MPI and sockets

MPI cluster implemented as client/server system
   Primary node (client) opens connection to each remote server
   Secondary nodes run MPI server listening on service port number
MPI maps socket numbers to MPI rank construct
   Socket number provides ID number for node in MPI_COMM_WORLD

Node performs MPI send to remote node
   Writes to socket for remote node

Node performs MPI receive
   Reads from socket for remote node
Network Latency in Cluster Computing

Amdahl's law with communication overhead

\[
S = \frac{1}{(1 - F_P) + F_P \left( \frac{1}{N} + F_{\text{overhead}} \right)}
\]

\[
F_P = \frac{\text{Parallel instructions}}{\text{All instructions}}
\]

\[
F_{\text{overhead}} = \frac{CPI^{\text{comm}}}{CPI} = \frac{\text{Communication activity CC per instruction}}{\text{Computation activity CC per instruction}}
\]

Cluster computing works well for \( F_P \) and \( N \) large, and \( F_{\text{overhead}} \) small

Communications overhead

Networking software
Network hardware
CPU CC \( \approx 10^{-9} \) sec
Message latency \( \approx 150 \times 10^{-6} \) sec
\( \approx 1.5 \times 10^5 \) CC
Comparable to page fault

Ref: Cisco Systems, "Cluster Computing"
Hardware Support for Cluster Processing

**TCP offload engine (TOE)**

- Dedicated communication processor
- Processes TCP software for CPU(s) in node
  - Segmentation/reassembly (SAR)
  - Error checking
- Feature of mainframes since 1960s
- Returning as asymmetric multicore processors

**Remote Direct Memory Access (RDMA)**

- Performs DMA functions between nodes

Ref: Cisco Systems, "Cluster Computing"
Typical Cluster Configuration

Node

Shared memory SMP running tightly coupled code over OpenMP

High speed interconnect

TCP/IP over Gigabit Ethernet (1 Gbps serial transmission)

Cluster of SMPs (CluMP)

Nodes exchange data for loosely coupled code over MPI

https://computing.llnl.gov/tutorials/linux_clusters/
Thunder Cluster

**CluMP type cluster computer**

1024 Linux nodes

Each node is 4-CPU Itanium 2 "Madison Tiger4" SMP

**Built at Lawrence Livermore National Laboratory (LLNL)**

US Department of Energy National Nuclear Security Administration

Second fastest supercomputer in June 2004

(http://www.top500.org)

Peak performance: 23 TFLOPS

https://computing.llnl.gov/tutorials/linux_clusters/thunder.html
Node Technical Details

1024 nodes

4-CPU 1.4 GHz Itanium 2 "Madison Tiger4"
VLIW explicitly parallel RISC language machine
8 GB DDR per node (8 TB total)
16KB L1 Data Cache
16KB L1 Instruction Cache
256 KB L2 Cache
4 MB L3 Cache

Intel E8870 chipset
6.4 GB/sec switch
Interconnect Details

Quadrics QsNetII

PCI-to-PCI switch

900 MB/s of user space to user space bandwidth

Optical interconnect extends maximum link length to 100m

Fat tree topology

Up to 4096 nodes in 5 switching stages

Switch built from Elan4 ASIC

- 8 port custom switch
- Integrated DMA controller

Supports collective communication

- Barrier synch
- Broadcast
- Multicast

http://www.quadrics.com
Blue Gene/L

Massively parallel supercomputer

- 65,536 dual-processor nodes
- 32 TB (32,768 GB) main memory
- Based on IBM system-on-a-chip (SOC) technology
- Peak performance of 596 teraflops

Built at Lawrence Livermore National Laboratory (LLNL)

- US Department of Energy National Nuclear Security Administration
- 2nd fastest supercomputer in June 2008 (1st in 2007)

Target applications

- Large compute-intensive problems
- Simulation of physical phenomena
- Offline data analysis

Architectural goals

- High performance on target applications
- Cost/performance of typical server

Gara, et. al., "Overview of the Blue Gene/L system architecture", IBM Technical Journal
Target Applications

**Simulations of physical phenomena**

- Represent physical state variables on lattice
- Time evolution through sequence of iterations
  - Exchange state variables among lattice points
  - Update local state variables according to physical laws
- Short-range physical interactions
  - Exchange state variables within small volume of lattice points
- Spectral methods
  - FFT and others
  - Require long-range communication between lattice point
  - Preferable for good convergence behavior

**Mapping lattice sites onto multiple processors straightforward**

- Each node allocated subset of problem
- Spatial decomposition $\Rightarrow$ nodes communicate with small group of nodes

**Global reduction**

- Normalization over full data set
- Decision step in algorithm
Scaling Issues in Very Large Systems

Cost / performance ratio scales badly at massive scale

- Equipment overhead
  - Equipment racks
  - Floor space
  - Electrical power
  - Cooling

- Communication complexity
  - Fully connected network grows as (number of processors)
  - Raises communication overhead FP

Computing power versus electrical power

- Performance / rack = (Performance / Watt) × (Watts / rack)
- Watts / rack ~ constant ~ 20 kW / rack
- Rack performance limited by Performance / Watt
- Performance / Watt scales badly with performance
Architectural Decisions

High integration and hardware density

- Fewer racks ⇒ lower rack overhead
- Higher performance per rack
- Shorter and simpler communication paths

Requirements

High performance / Watt

- Medium performance processors
- High performance processor ⇒ high ILP at high power
- TLP more significant than ILP in target applications

Highly integrated system on a chip (SOC)

- Multiple cores
- Processor + memory + I/O on single integrated circuit
- High performance interconnect
- Application specific integrated circuit (ASIC)
Implementation

65,536 nodes

- 130-nm copper IBM CMOS 8SFG technology
- Single ASIC
  - two processors
  - nine DDR SDRAM chips
- Nodes physically small
  - 2 to 10 times higher density greater than high-frequency uniprocessors

1,024 dual-processor nodes per rack

- 27.5 kW of total power
- 85% of inter-node connectivity contained within racks

Five interconnection networks

- High connectivity static switching fabric for bulk communications
- Collective network supporting scatter/gather/broadcast/reduction
- Barrier network supporting barrier synchronization
- Control system network
- Gigabit Ethernet network supporting I/O
Node ASIC
**Static interconnection network**

- $64 \times 32 \times 32$ three-dimensional torus for bulk communication
- Nodes communicate with neighboring nodes
- Same bandwidth and nearly same latency to all nodes
  - No edges in torus configuration
- Simplifies programming model
- Nearest-neighbor link rate is 1.4 Gb/s

**Collective network**

- Global broadcast/scatter/gather of data
- Improvement over 3D torus network
  - 1% to 10% of typical supercomputer latency on collective operations
- Built-in ALU hardware on network
  - Supports reduction operations
    - Min, max, sum, OR, AND, XOR

**Barrier network**

- Propagates OR of nodes to trigger interrupts
- Propagates AND of nodes for synchronization

2 $\times$ 2 $\times$ 2 torus
IBM Sequoia — Blue Gene/Q

Massively parallel supercomputer
96 K (98,304) 16-core nodes
1.6 PB (1.6 \times 1024 \times 1024 GB) main memory
Based on IBM POWER system-on-a-chip (SOC)
Peak performance of 16 petaflops

Built at Lawrence Livermore National Laboratory (LLNL)
US Department of Energy National Nuclear Security Administration
Fastest supercomputer in June 2012

Operating systems
Red Hat Enterprise Linux on I/O nodes
Connect to file system
Compute Node Linux (CNL) on application processors
Runtime environment based on Linux kernel

Target applications
Advanced Simulation and Computing Program
Simulated testing of US nuclear arsenal
Nuclear detonations banned since 1992
Example of Cluster Application

Parallel multiplication of $N \times N$ matrix by vector

Medium coupled computation

$$c[i] = \sum_{j=0}^{N-1} A[i][j] \times b[j] = A[i] \text{ dot } b$$

Works well on shared memory SMP

Dominated by communication overhead on cluster

Manager code

setup
initialize matrix $a[i][j]$
for (i=0 ; i<N ; i++)
    MPI_Send row $A[i]$ to rank_i
for (i=0 ; i<N ; i++)
    MPI_Recv $c[i]$ as row * vector

Worker code

setup
initialize vector $b[i]$
for (i=0 ; i<N ; i++)
    MPI_Recv row $A[i]$
dotp = 0.0
for (i=0 ; i<N ; i++)
dotp += $c[i] \times b[i]$
MPI_Send dotp
Speedup?

Computations

Each row of vector result
- N multiplications
- N – 1 additions
- 2N – 1 flops

Communications

Send N × N matrix ⇒ N² floats
Return N vector components ⇒ N floats
Total = N² + N floats sent
Send (N² + N) / (2N – 1) floats per flop

Overhead Factor

\[ S = \frac{1}{(1 - F_p) + F_p \left( \frac{1}{N} + N \right)} \]

On fast interconnect
\[ FP_{overhead} = \frac{CPI^{comm}}{CPI} = \frac{N^2 + N}{2N - 1} \times \frac{2CC^{communication}}{CC_{processor}} \sim N \]

\[ S = \frac{1}{(1 - F_p) + F_p \left( \frac{1}{N} + N \right)} \sim \frac{1}{N} \quad \text{for } F_p \approx 1, N \text{ large} \]
Example of Reasonable Cluster Application

Calculate \( \pi = \int_{0}^{1} \frac{4}{1+x^2} \, dx \) numerically

**Sequential Version**

N steps
step = 1 / N
for (i=0; i<N; i++)
{
x = (i+0.5)*step;
sum = sum + 4.0/(1.0 + x*x);
}
pi = step * sum;

**Loop Computations**

N * (sum, product, product, sum, division, sum) + product
~ 6N + 1 flops

**Cluster Version**

N processors
processor i computes
x = (i+0.5)*step;
sum = 4.0/(1.0 + x*x);
MPI_REDUCE (p, N, 1, MPI_INT,
MPI_SUM,0,MPI_COMM_WORLD)
pi = step * p;

**Computations**

sum, product, product, sum, division + reduce_add
~ 6 flops

**Communications**

Send / receive 1 float per 6 flops
\( F_{overhead} \approx \frac{1}{6} \times 2 \Rightarrow S \sim 3 \)
Examples of Good Cluster Applications

**Thread-replicating servers**
- Web servers
- Database access servers
- On-line transaction processing (OLTP)

**Number crunching on separable data**
- Numerical integration
- Ray tracing
- VLSI layout planning

**Numerical solutions for 2nd order partial differential equations**
- Electrodynamics and optics
- Quantum mechanics and quantum chemistry
- Thermodynamics, hydrodynamics, aerodynamics
- Plasma magnetohydrodynamics
Learning Curve in Parallel Programming

**Small scale projects**
- Single programmer
- "Embarrassingly" parallel
- Obvious task/data decomposition
- Follow design rules
- Vector constructs
- OpenMP constructs
- Structure optimization
- Sequential consistency

**Medium scale projects**
- Coordination — folklore or science
- Debugging consistency across team of programmers

**Large scale projects**
- High performance computing (HPC) for difficult mathematical problems
- Requires expertise in programming and heuristics of mathematical problem
- Programmers become specialists in specific HPC field
### Empirical Results

#### Study of student programmers

Graduate-level courses in parallel programming
Students implement "game of life" program in C on PC cluster
Compare serial with parallel solutions (OpenMP and MPI)

<table>
<thead>
<tr>
<th>Programming model</th>
<th>Speedup on 8 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parallel version compared to sequential</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td>MPI</td>
<td>4.7</td>
</tr>
<tr>
<td></td>
<td>2.8</td>
</tr>
<tr>
<td>OpenMP</td>
<td>6.7</td>
</tr>
</tbody>
</table>

**Speedup on parallel version with N = 8 compared to parallel version with N = 1**

<table>
<thead>
<tr>
<th>Programming model</th>
<th>Speedup on 8 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.0</td>
</tr>
<tr>
<td></td>
<td>4.8</td>
</tr>
<tr>
<td></td>
<td>5.6</td>
</tr>
<tr>
<td>OpenMP</td>
<td>5.7</td>
</tr>
</tbody>
</table>