COMP 322: Fundamentals of Parallel Programming

Lecture 34: MPI (contd), Introduction to Cloud Computing

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Acknowledgments for Today's Lecture

- Slides accompanying Chapter 6 of "Introduction to Parallel Computing", 2nd Edition, Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, Addison-Wesley, 2003
  - http://www-users.cs.umn.edu/~karypis/parbook/Lectures/AG/chap6_slides.pdf
- mpiJava home page: http://www.hpjava.org/mpiJava.html
  - Includes specification for mpiJava
    - http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec.pdf
- MPI lectures given at Rice HPC Summer Institute 2009, Tim Warburton, May 2009
- Slides from Lectures 1 and 2 in UC Berkeley CS61C course, "Great Ideas in Computer Architecture (Machine Structures), Spring 2012, Instructor: David Patterson
  - http://inst.eecs.berkeley.edu/~cs61c/sp12/

Outline

- **<u>MPI (contd)</u>**

- **Warehouse Scale Computers and Cloud Computing**

Worksheet #33: MPI Gather

```
1.    MPI.Init(args) ;
2.    int myrank = MPI.COMM_WORLD.Rank() ;
3.    int numProcs = MPI.COMM_WORLD.Size() ;
4.    int size = ...;
5.    int[] sendbuf = new int[size];
6.    int[] recvbuf = new int[???];
7.    . . . // Each process initializes sendbuf
8.    MPI.COMM_WORLD.Gather(sendbuf, 0, size, MPI.INT,
9.                          recvbuf, 0, size, MPI.INT,
10.                         0/*root*/);
11.   . . .
12.   MPI.Finalize();
```

Question: In the space below, indicate what values should be provided instead of ??? in line 6, and why.

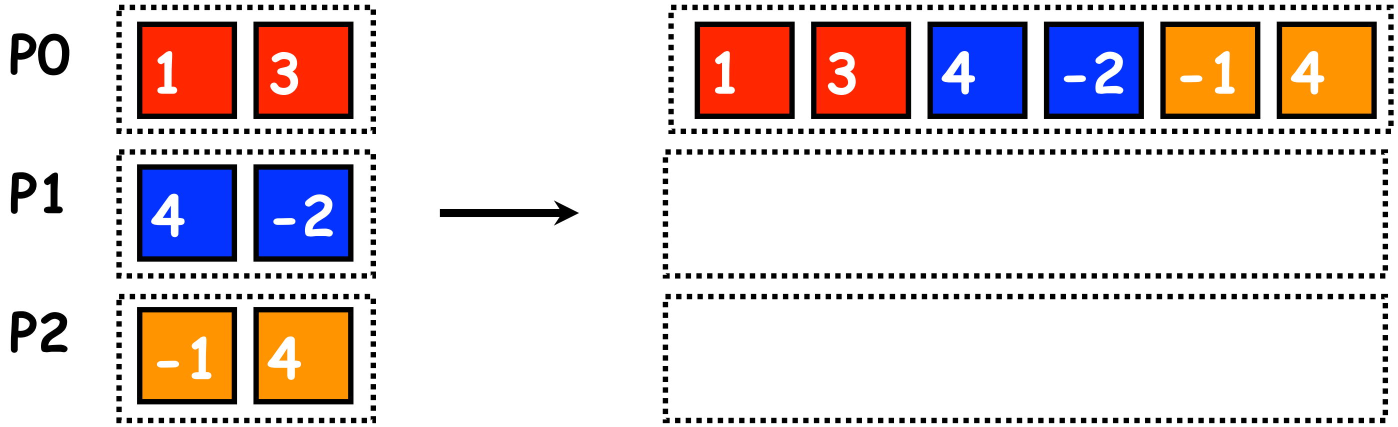
Answer:

recvbuf should be allocated with numProcs*size elements for Gather. Since recvbuf also needs to be allocated in the root, line 6 can be replaced by:

```
6.int[] recvbuf = (myrank==0) ? new int[numProcs*size] : null;
```

MPI_Gather (Recap)

- **Use to copy an array of data from each process into a single array on a single process.**
- **Graphically:**



- **Note: only process 0 (P0) needs to supply storage for the output**

```
void Gather(Object sendbuf, int sendoffset, int sendcount,
            Datatype sendtype, Object recvbuf, int recvoffset,
            int recvcount, Datatype recvtype, int root)
```

– **Each process sends the contents of its send buffer to the root process.**

Collective Communications

- Each collective operation is defined over a communicator (most often, MPI.COMM_WORLD)
  - Each collective operation contains an *implicit barrier*. The operation completes and execution continues when all processes in the communicator perform the *same* collective operation.
  - A mismatch in operations results in *deadlock* e.g.,

    Process 0: .... MPI.Bcast(...) ....

    Process 1: .... MPI.Bcast(...) ....

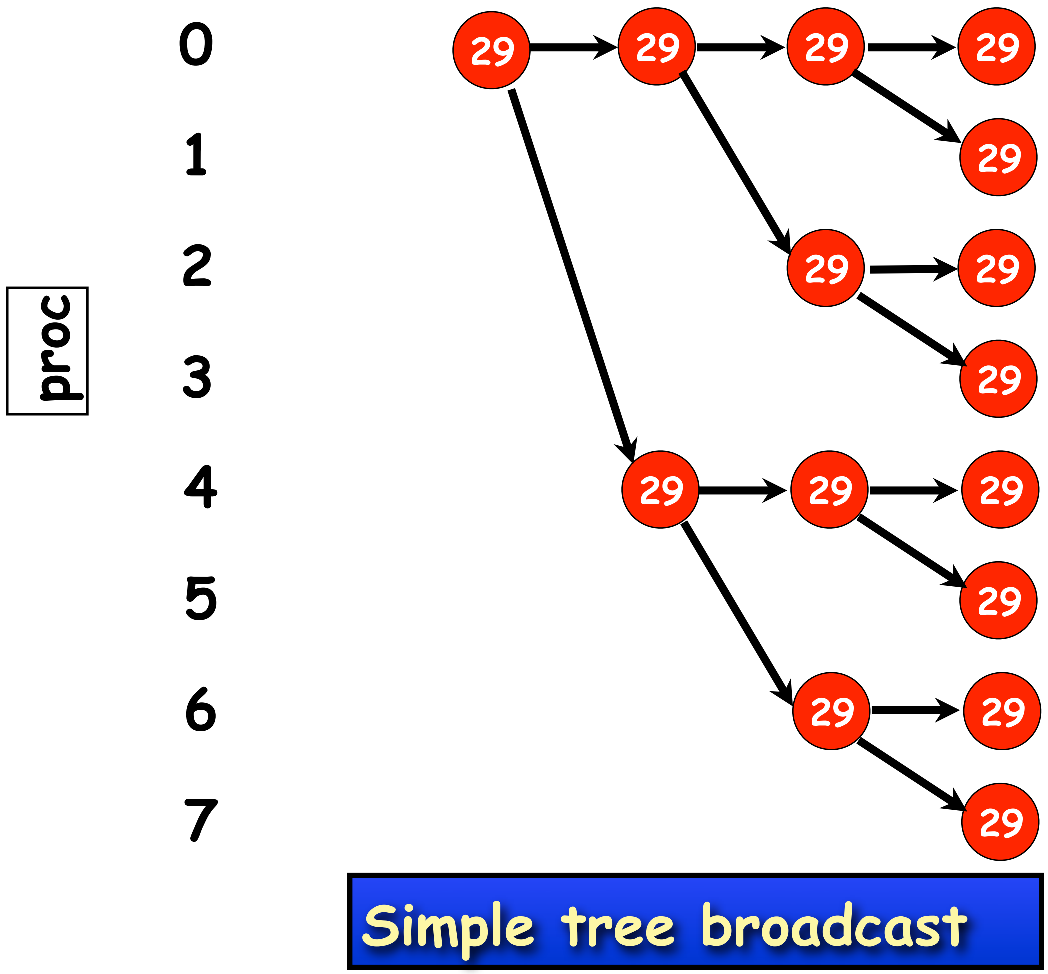
    Process 2: .... MPI.Gather(...) ....
- We can model the synchronization performed by MPI operations as phasers to understand their semantics
  - Assume that all processes are registered on multiple phasers, one for each kind of collective operation e.g., ph1 for Bcast, ph2 for Gather
  - The above example can be rewritten as follows, where doNext() performs a "next" operation on one phaser only

    Process 0: .... ph1.doNext(); ....

    Process 1: .... ph1.doNext(); ....

    Process 2: .... ph2.doNext(); ....

MPI_Bcast



A root process sends same message to all

29 represents an array of values

Examples of Collective Operations

```
void Barrier()
```

- **Blocks the caller until all processes in the group have called it.**

```
void Gather(Object sendbuf, int sendoffset, int sendcount,
  Datatype sendtype, Object recvbuf, int recvoffset,
  int recvcount, Datatype recvtype, int root)
```

- **Each process sends the contents of its send buffer to the root process.**

```
void Scatter(Object sendbuf, int sendoffset, int sendcount,
  Datatype sendtype, Object recvbuf, int recvoffset,
  int recvcount, Datatype recvtype, int root)
```

- **Inverse of the operation Gather.**

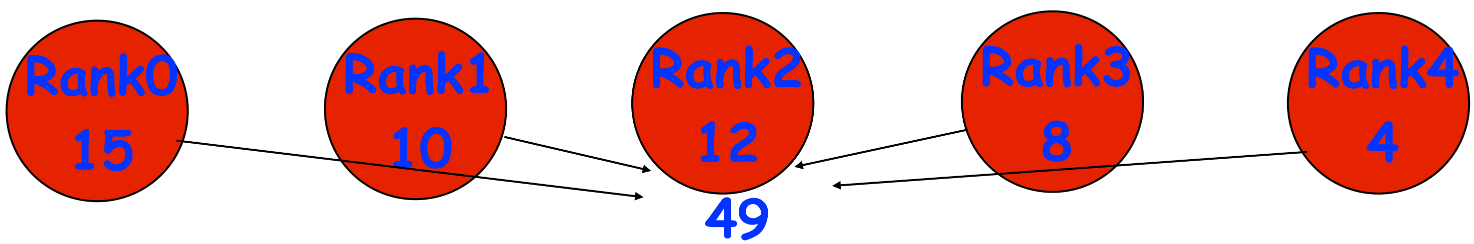
Let's try out a Scatter in Worksheet #34!

```
void Reduce(Object sendbuf, int sendoffset, Object recvbuf,
  int recvoffset, int count, Datatype datatype, Op op,
  int root)
```

- **Combine elements in send buffer of each process using the reduce operation, and return the combined value in the receive buffer of the root process.**

MPI Reduce

```
void MPI.COMM_WORLD.Reduce(
                    Object          sendbuf     /* in */,
                    int             sendoffset  /* in */,
                    Object          recvbuf     /* out */,
                    int             recvoffset  /* in */,
                    int             count       /* in */,
                    MPI.Datatype    datatype    /* in */,
                    MPI.Op          operator    /* in */,
                    int             root        /* in */ )
```




```
MPI.COMM_WORLD.Reduce( msg, 0, result, 0, 1, MPI.INT, MPI.SUM, 2);
```

Predefined Reduction Operations

Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	int, long, float, double
`MPI_MIN`	Minimum	int, long, float, double
`MPI_SUM`	Sum	int, long, float, double
`MPI_PROD`	Product	int, long, float, double
`MPI_LAND`	Logical AND	int, long
`MPI_BAND`	Bit-wise AND	byte, int, long
`MPI_LOR`	Logical OR	int, long
`MPI_BOR`	Bit-wise OR	byte, int, long
`MPI_LXOR`	Logical XOR	int, long
`MPI_BXOR`	Bit-wise XOR	byte, int, long
`MPI_MAXLOC`	max-min value-location	Data-pairs (see next slide)
`MPI_MINLOC`	min-min value-location	Data-pairs

MPI_MAXLOC and MPI_MINLOC

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i 's).
- `MPI_MINLOC` does the same, except for minimum value of v_i.

Value	15	17	11	12	17	11
Process	0	1	2	3	4	5

```
MinLoc(Value, Process) = (11, 2)
MaxLoc(Value, Process) = (17, 1)
```

An example use of the `MPI_MINLOC` and `MPI_MAXLOC` operators.

Datatypes for MPI_MAXLOC and MPI_MINLOC

MPI datatypes for data-pairs used with the `MPI_MAXLOC` and `MPI_MINLOC` reduction operations.

MPI Datatype	C Datatype
`MPI_2INT`	**pair of** ints
`MPI_SHORT_INT`	short **and** int
`MPI_LONG_INT`	long **and** int
`MPI_LONG_DOUBLE_INT`	long double **and** int
`MPI_FLOAT_INT`	float **and** int
`MPI_DOUBLE_INT`	double **and** int

More Collective Communication Operations

- **If the result of the reduction operation is needed by all processes, MPI provides:**

```
void AllReduce(Object sendbuf, int sendoffset,
Object recvbuf, int recvoffset, int count,
Datatype datatype, Op op)
```

- **MPI also provides the MPI_Allgather function in which the data are gathered at all the processes.**

```
void AllGather(Object sendbuf, int sendoffset,
int sendcount, Datatype sendtype, Object recvbuf,
int recvoffset, int recvcount, Datatype recvtype)
```

- **To compute prefix-sums, MPI provides:**

```
void Scan(Object sendbuf, int sendoffset,
Object recvbuf, int recvoffset, int count,
Datatype datatype, Op op)
```

MPI_Allreduce

```
void MPI.COMM_WORLD.Allreduce(
        Object[]        sendbuf         /* in */,
        int             sendoffset      /* in */,
        Object[]        recvbuf         /* out */,
                        int             recvoffset      /* in */,
                        int             count   /* in */,
                        MPI.Datatype datatype          /* in */,
                        MPI.Op                  operator        /* in */)
```

Equivalent to Reduce followed by Bcast

Integration example in mpiJava (1 of 2)

```java
static public void main(String[] args) throws MPIException {
    MPI.Init(args) ;
    int myrank = MPI.COMM_WORLD.Rank();
    int nprocs = MPI.COMM_WORLD.Size();
    int[] size = new int[1];
    if(myrank == 0) size[0] = args[1]; //# points for integration
    // Broadcast size to all processes
    MPI.COMM_WORLD.Bcast(size, 0, 1, MPI.INT, 0);
    int npts = size[0];
    // Identify "chunk" of npts according to myrank
    int nlocal = (npts-1)/nprocs + 1;
    int nbeg = myrank*nlocal +1;
    int nend = Math.min(nbeg+nlocal-1,npts);
```

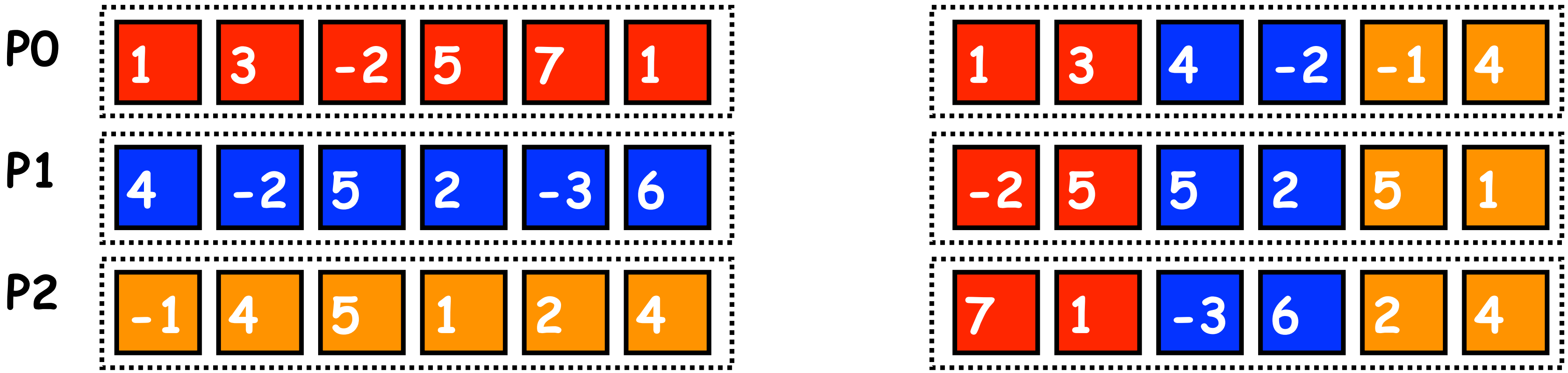
Source: http://users.cs.cf.ac.uk/David.W.Walker/CM0323/code.html

Integration example in mpiJava (2 of 2)

```
    // compute local sum for my chunk
    double delta = Math.PI/npts;
    double psum = 0.0;
    for(int i=nbeg;i<=nend;i++){
        psum += (Math.sin((i-0.5)*delta))*delta;
    }
    double [] localSum = new double[1]; localSum[0] = psum;
    double [] globalSum = double[1]; // buffer for global sum
    MPI.COMM_WORLD.AllReduce(localSum, 0, globalSum, 0,
                         1, MPI.DOUBLE, MPI.SUM);
    if (myrank==0){
        System.out.println("The integral = " + globalSum[0]);
    }
    MPI.Finalize();
}
```

MPI_Alltoall

```
void Alltoall(Object sendbuf, int sendoffset, int sendcount,
              Object recvbuf, int recvoffset, int count,
              Datatype datatype)
```



- **Each process submits an array to MPI_Alltoall.**
- **The array on each process is split into *nprocs* sub-arrays**
- **Sub-array *n* from process *m* is sent to process *n* placed in the *m*'th block in the result array.**

Outline

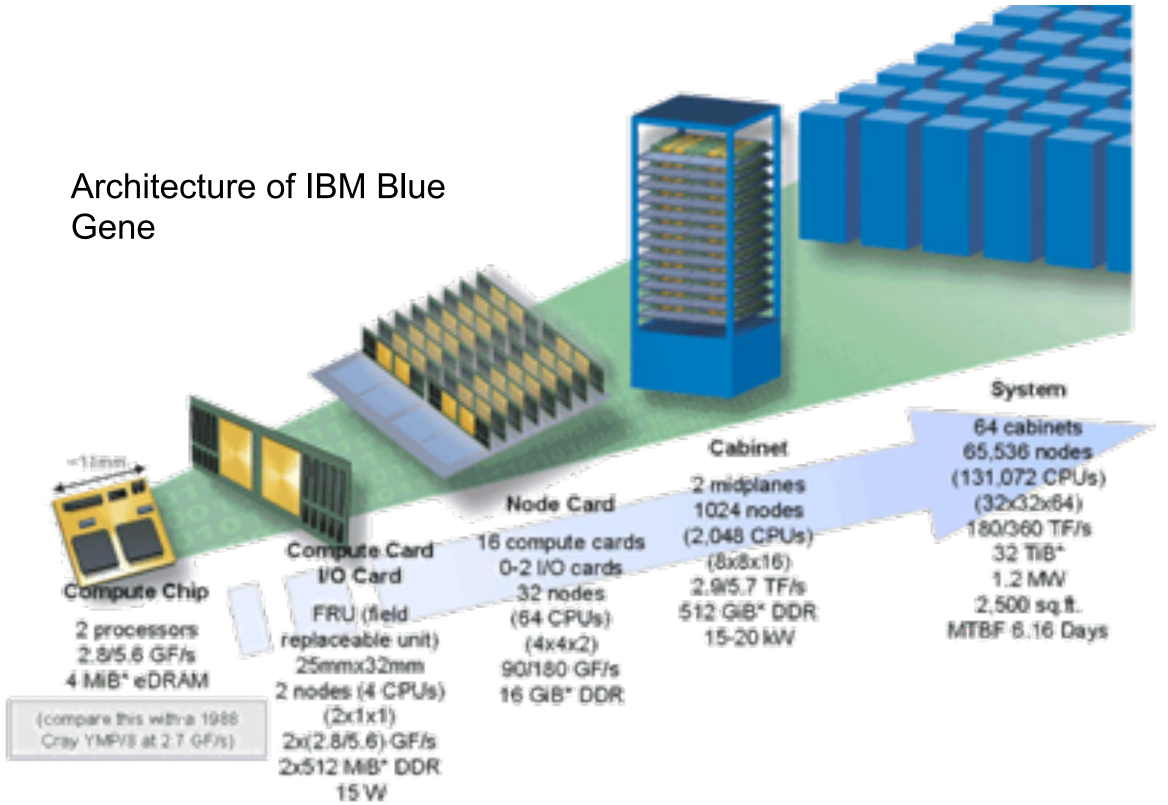
- MPI (contd)
- <u>Warehouse Scale Computers and Cloud Computing</u>

Supercomputing niche

Cray
(1976)

IBM Blue
Gene
(2005)

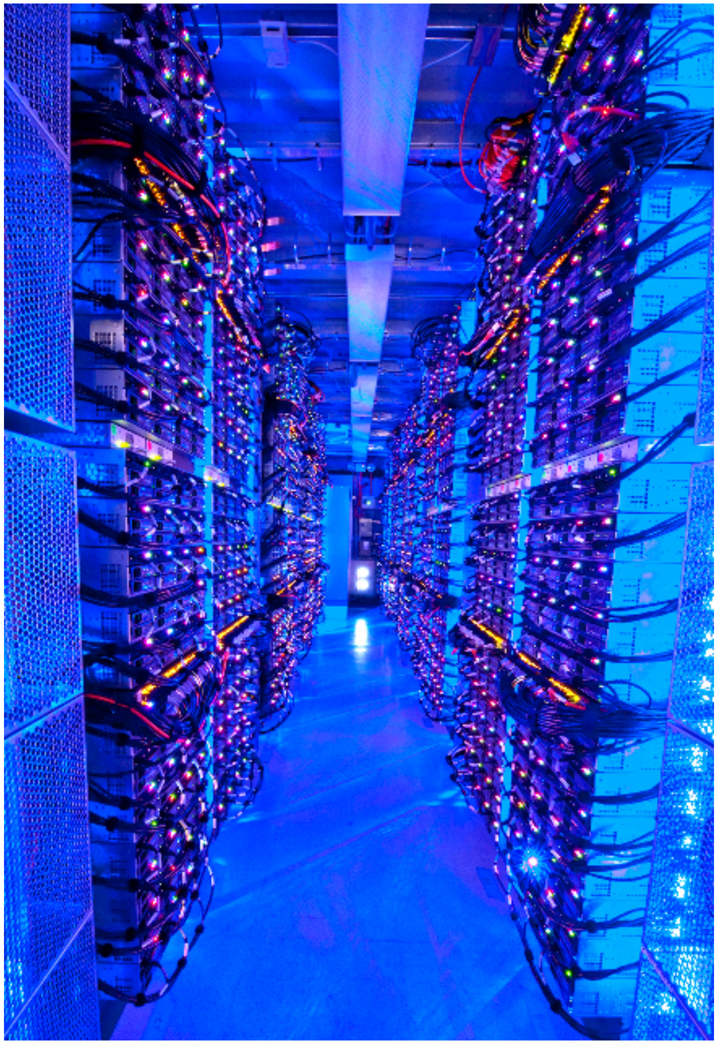
Architecture of IBM Blue
Gene



Source: www.csm.ornl.gov/~hqi/CrashCourse06/3-**MPI**.ppt

Mainstream trends

Personal Mobile Devices (PMD): Relying on wireless networking, Apple, Nokia, … build $500 smartphone and tablet computers for individuals
=> Objective C, Android OS



Cloud Computing: Using Local Area Networks, Amazon, Google, … build $200M Warehouse Scale Computers with 100,000 servers for Internet Services for PMDs
=> MapReduce, Ruby on Rails

Parallelism is the dominant technology trend in Cloud Computing

Software

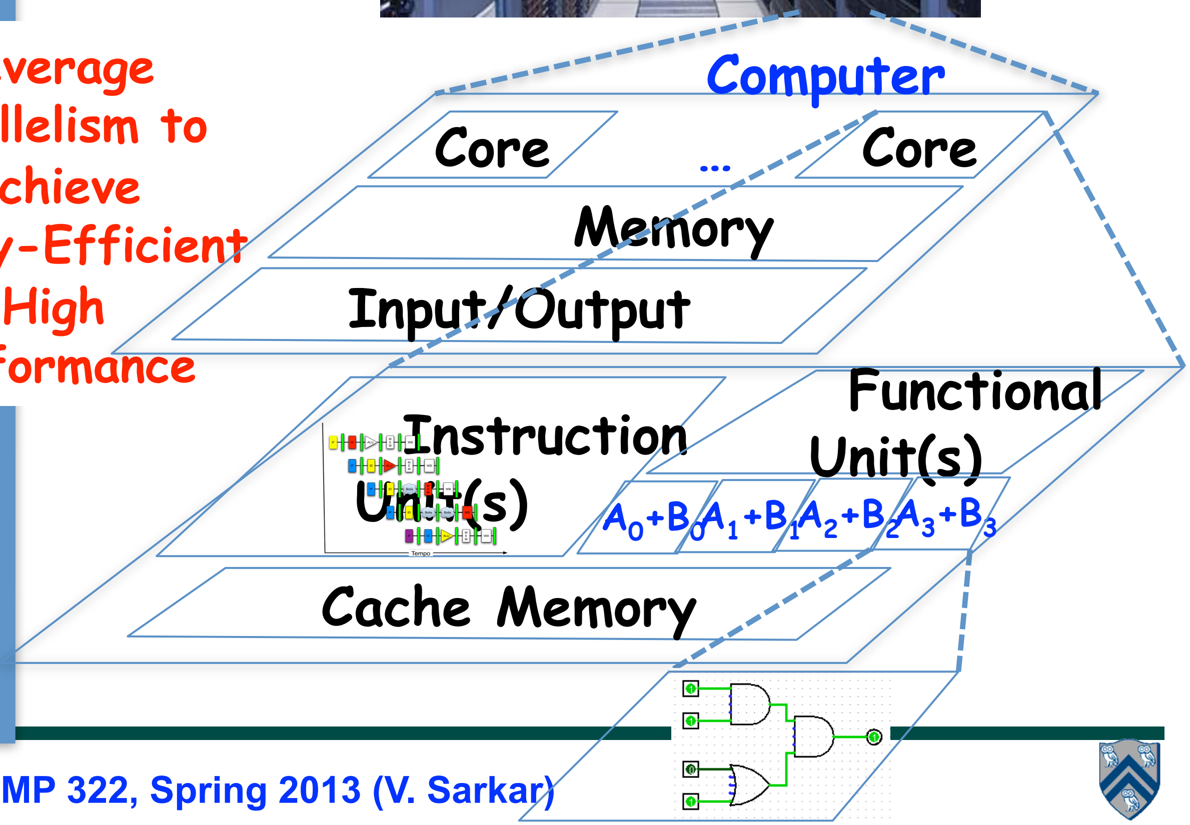
- **Parallel Requests**
  Assigned to computer
  e.g., Search "Rice Marching Owl Band"
- **Parallel Threads**
  Assigned to core
  e.g., Lookup, Ads
- **Parallel Instrs**
  >1 instruction/cycle
  e.g., 5 pipelined instructions
- **Parallel Data**
  >1 data access/cycle
  e.g., Load of 4 consecutive words

Hardware

Warehouse Scale Computer

Smart Phone

Leverage Parallelism to Achieve Energy-Efficient High Performance



Parallelism enables "Cloud Computing" as a Utility

- Offers computing, storage, communication at pennies per hour
- No premium to scale:

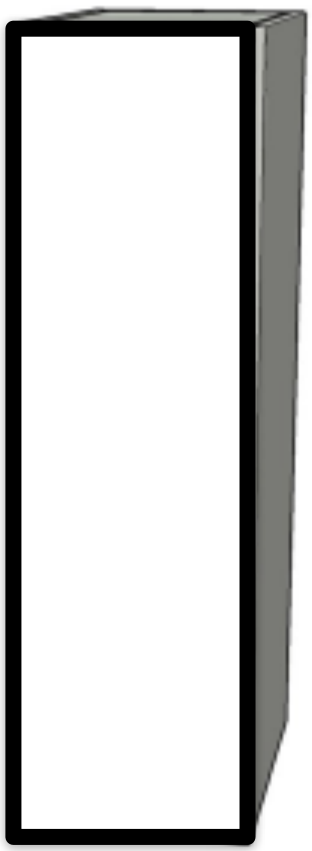
  1000 computers @ 1 hour
  = 1 computer @ 1000 hours
- Illusion of infinite scalability to cloud user
  - As many computers as you can afford
- Leading examples: Amazon Web Services (AWS), Google App Engine, Microsoft Azure
  - Economies of scale pushed down cost of largest datacenter by factors 3X to 8X
  - Traditional datacenters utilized 10% - 20%
  - Make profit offering pay-as-you-go use service at less than your costs for as many computers as you need
  - Strategic capability for company's needs

2012 AWS Instances & Prices

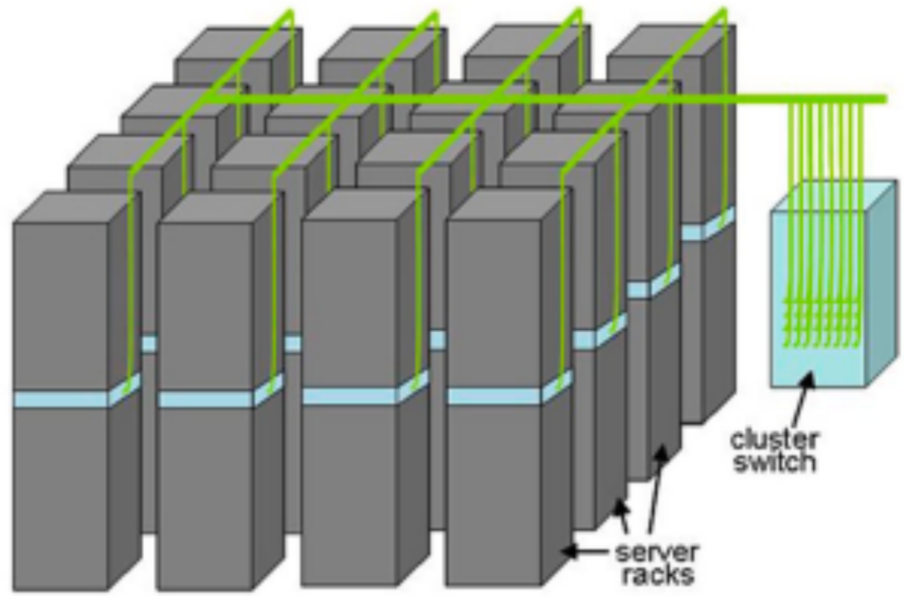
Instance	Per Hour	Ratio to Small	Compute Units	Virtual Cores	Compute Unit/ Core	Memory (GB)	Disk (GB)	Address
Standard Small	$0.08	1.0	1.0	1	1.00	1.7	160	32 bit
Standard Large	$0.34	4.0	4.0	2	2.00	7.5	850	64 bit
Standard Extra Large	$0.68	8.0	8.0	4	2.00	15.0	1690	64 bit
High-Memory Extra Large	$0.50	5.9	6.5	2	3.25	17.1	420	64 bit
High-Memory Double Extra Large	$1.20	14.1	13.0	4	3.25	34.2	850	64 bit
High-Memory Quadruple Extra	$2.40	28.2	26.0	8	3.25	68.4	1690	64 bit
High-CPU Medium	$0.17	2.0	5.0	2	2.50	1.7	350	32 bit
High-CPU Extra Large	$0.68	8.0	20.0	8	2.50	7.0	1690	64 bit
Cluster Quadruple Extra Large	$1.30	15.3	33.5	16	2.09	23.0	1690	64 bit
Eight Extra Large	$2.40	28.2	88.0	32	2.75	60.5	1690	64 bit

Equipment Inside a WSC

Server (in rack format): 1 ¾ inches high "1U", x 19 inches x 16-20 inches: 8 cores, 16 GB DRAM, 4x1 TB disk



7 foot Rack: 40-80 servers + Ethernet local area network (1-10 Gbps) switch in middle ("rack switch")

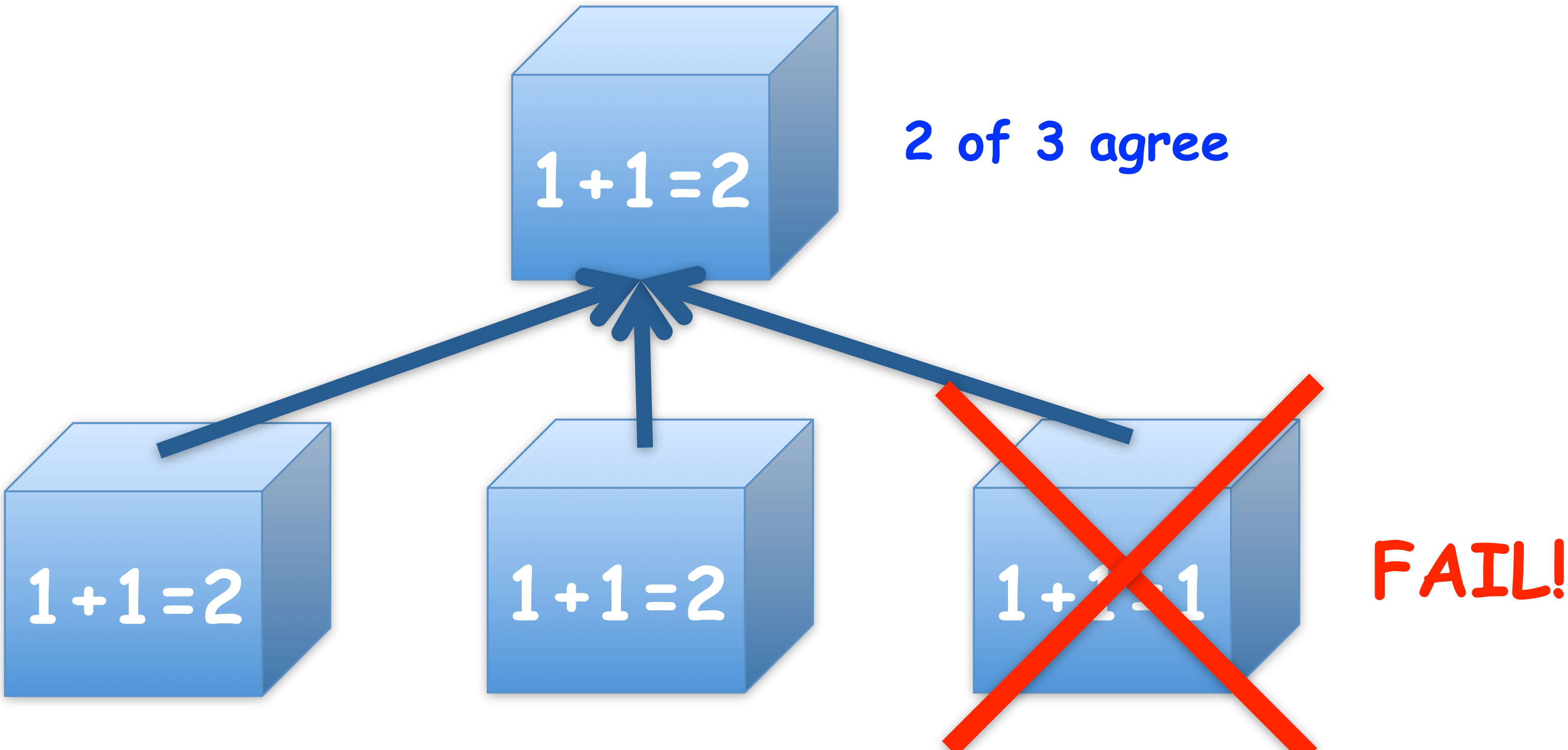


Array (aka cluster): 16-32 server racks + larger local area network switch ("array switch") 10X faster => cost 100X: cost $f(N^2)$

Server, Rack, Array

Parallelism enables Redundancy

- **Redundancy so that a failing piece doesn't make the whole system fail**



Increasing transistor density reduces the cost of redundancy

Redundancy enables Fault Tolerance and Resilience

- **Applies to everything from datacenters to storage to memory**
  - Redundant datacenters so that can lose 1 datacenter but Internet service stays online
  - Redundant disks so that can lose 1 disk but not lose data (Redundant Arrays of Independent Disks/ RAID)
  - Redundant memory bits of so that can lose 1 bit but no data (Error Correcting Code/ECC Memory)

Request-Level Parallelism (RLP)

- Hundreds or thousands of requests per second
  - —Not from your laptop or cell-phone, but from popular Internet services like Google search
  - —Such requests are largely independent
    - Mostly involve read-only databases
    - Little read-write (aka "producer-consumer") sharing
    - Rarely involve read–write data sharing or synchronization across requests
- Computation easily partitioned within a request and across different requests

Worksheet #34: MPI Scatter

Name 1: ________________________ Name 2: ________________________

```
MPI.Init(args) ;
int myrank = MPI.COMM_WORLD.Rank() ;
int numProcs = MPI.COMM_WORLD.Size() ;
int size = ...;
int[] sendbuf = new int[numProcs*size];
int[] recvbuf = new int[size];
if (myrank == 0) {
   . . . // Root initializes sendbuf
}
 MPI.COMM_WORLD.Scatter(___________________________
                       ___________________________
                       ____________);
. . .
MPI.Finalize();
```

Fill in the ____ blanks to implement a Scatter operation from the root to all processes (the inverse of the Gather shown in slides 4 and 5)