Parallel Computing I

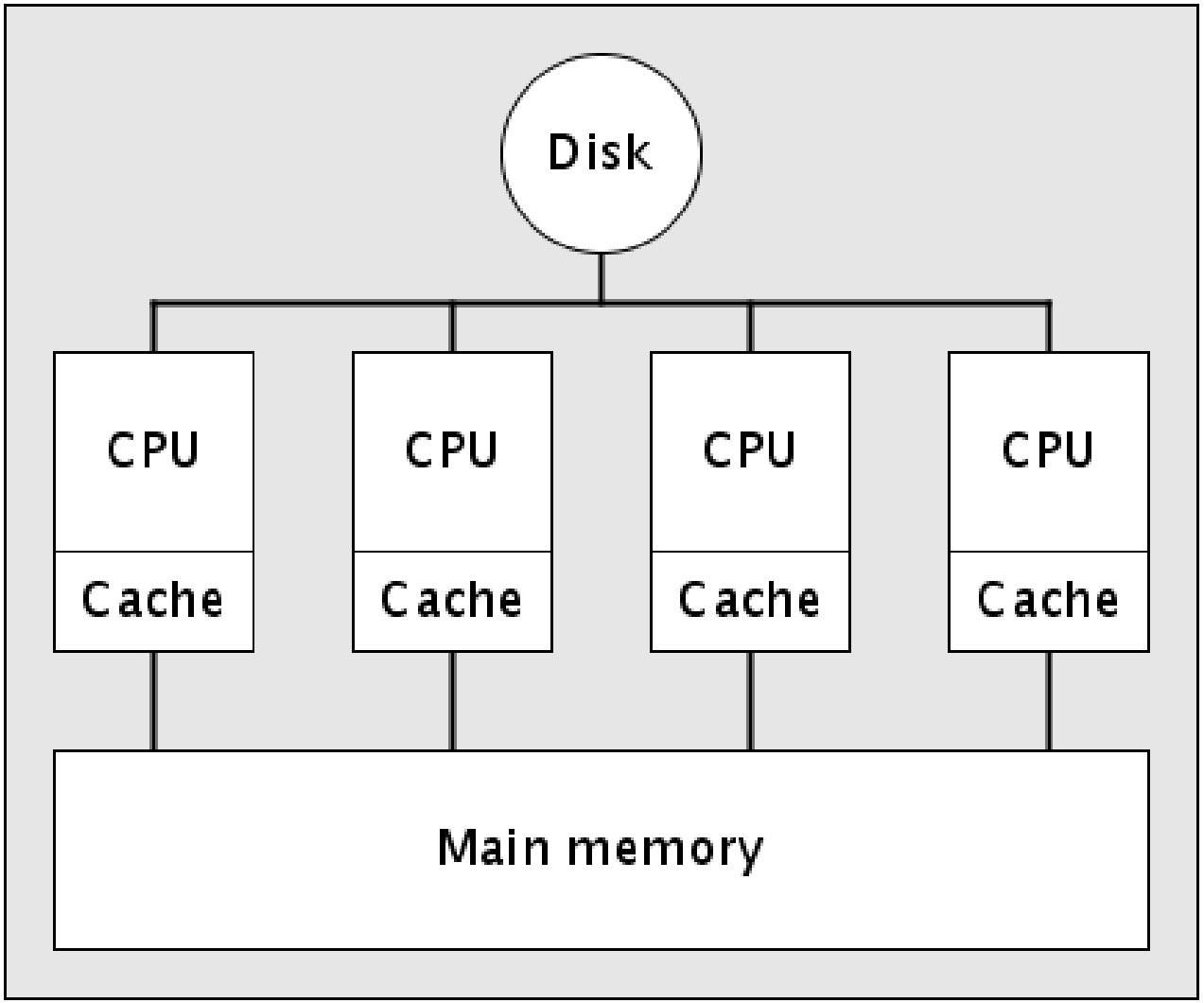
Cluster Parallel Programs
Cluster: Massively Parallel Problems

From SMP to Cluster

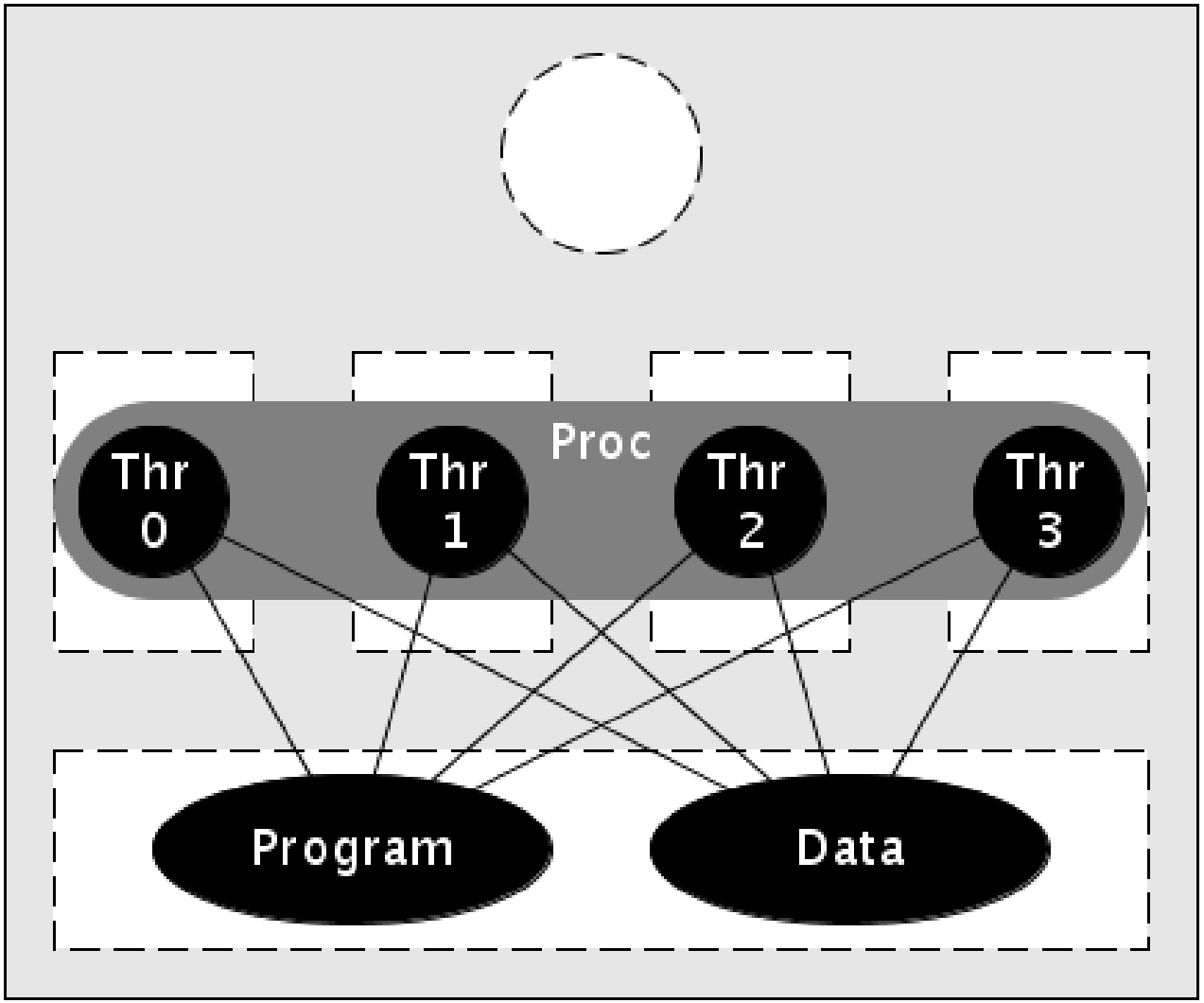
Finished our study of SMP Parallel Programs.

Beginning our study of Cluster Parallel Programs.

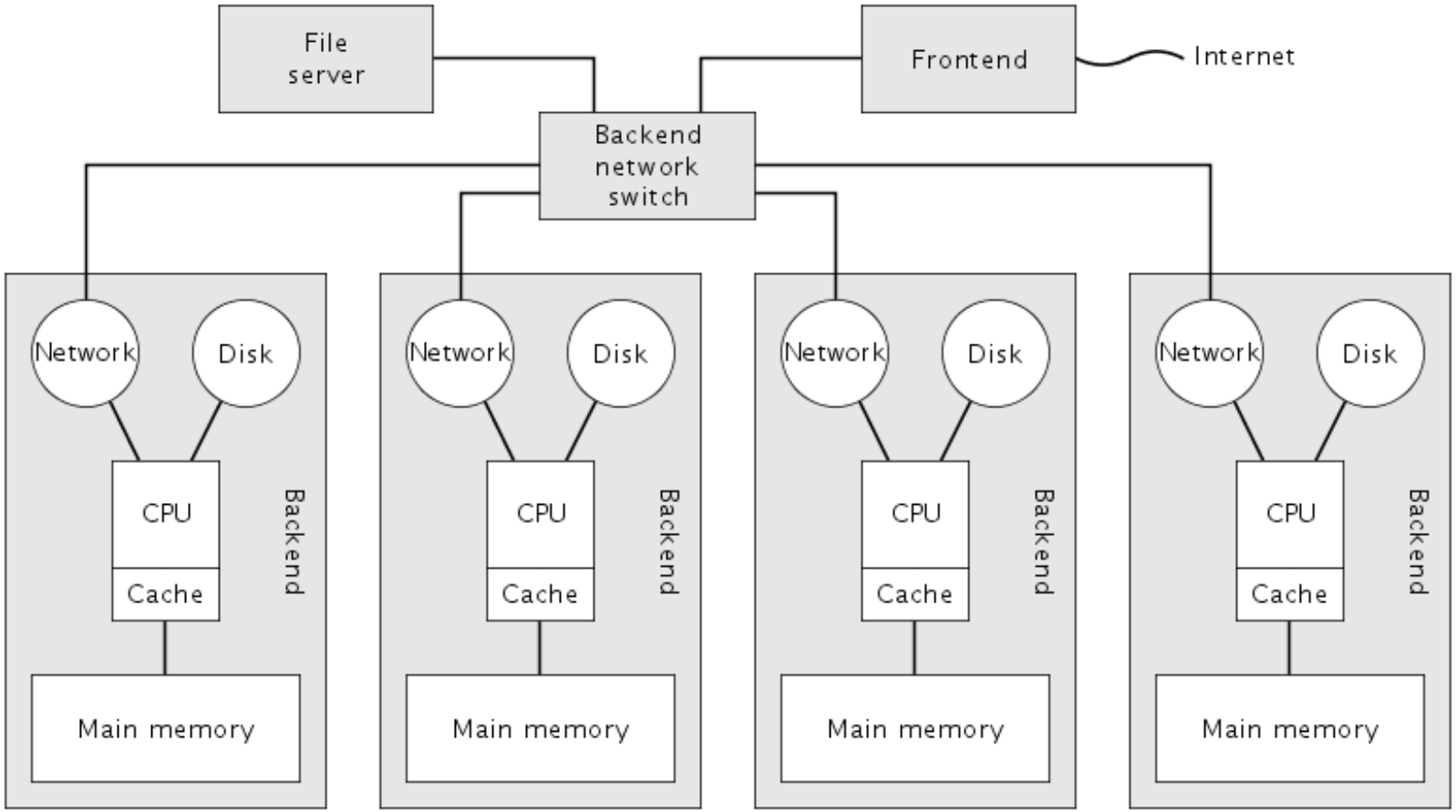
SMP Parallel Computers: Architecture



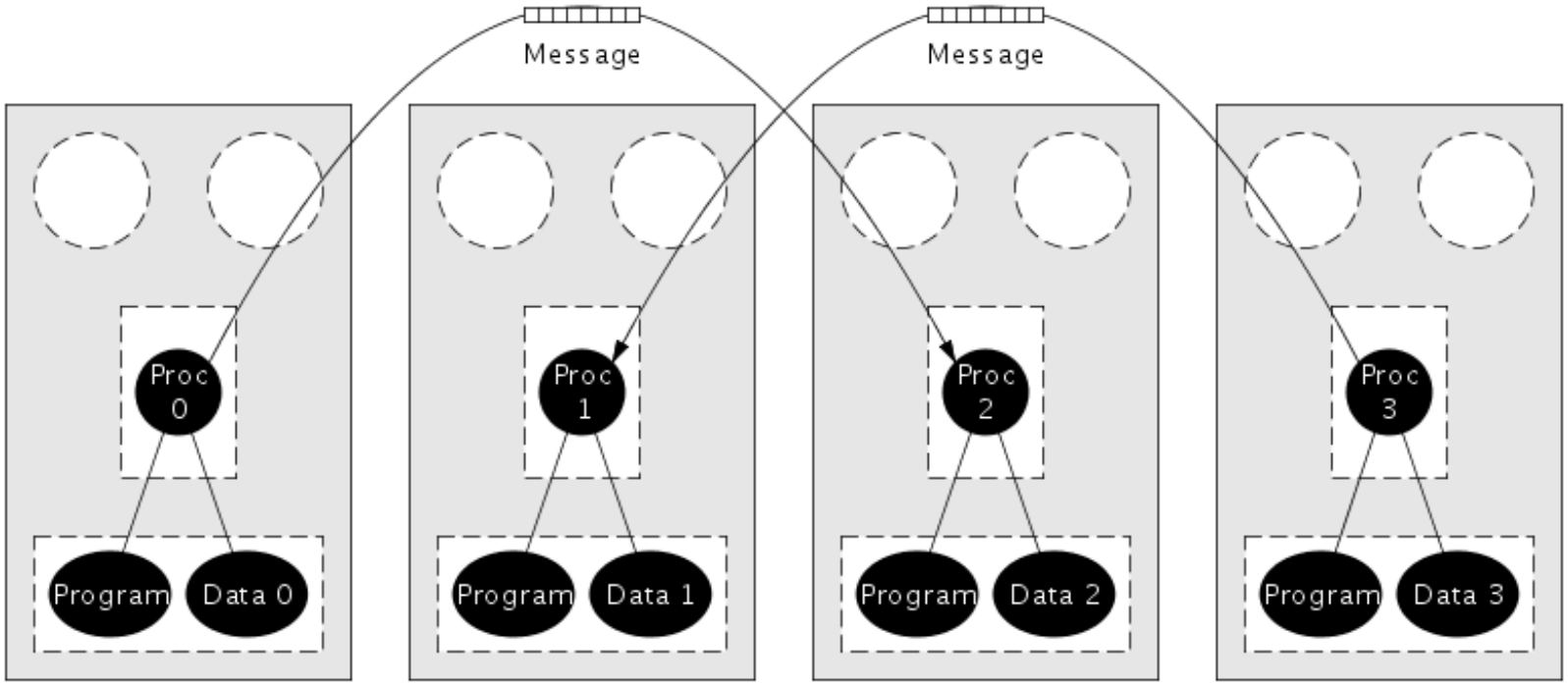
SMP Parallel Computers: Communication



Cluster Parallel Computers: Architecture



Cluster Parallel Computers: Communication



SMP vs Cluster Parallel Computers

- SMP Parallel Computers
  - Pro: Better suited for problems with data dependencies between processors
  - Con: Limited memory and CPU scalability
  - Con: Worse price/performance ratio
- Cluster Parallel Computers
  - Pro: No limits on scalability of memory and CPU
  - Pro: Better price/performance ratio
  - Con: Messaging overhead

Cluster Parallel Computers: Architecture

A cluster's performance is not only dicated by backend processors, but also by backend network characteristics:

- Latency (seconds)
  - amount of time needed to start up a message, regardless of the message's size
  - depends upon the hardware and software protocols of the network
- Bandwith (bits per second)
  - rate at which data is transmitted once a message has started.
- Bisection bandwith (bits per second)
  - rate at which data is transmitted
    if half the nodes are sending messages to the other half
- Topology
  - pattern of interconnections of the various elements
    (nodes, switches, links, etc.)

A First Cluster Parallel Program

Primality tests via trial division

- `code/PrimeTesterSeq.java`
- `code/PrimeTesterClu.java`

A First Cluster Parallel Program

Initialize PJ's communication layer:

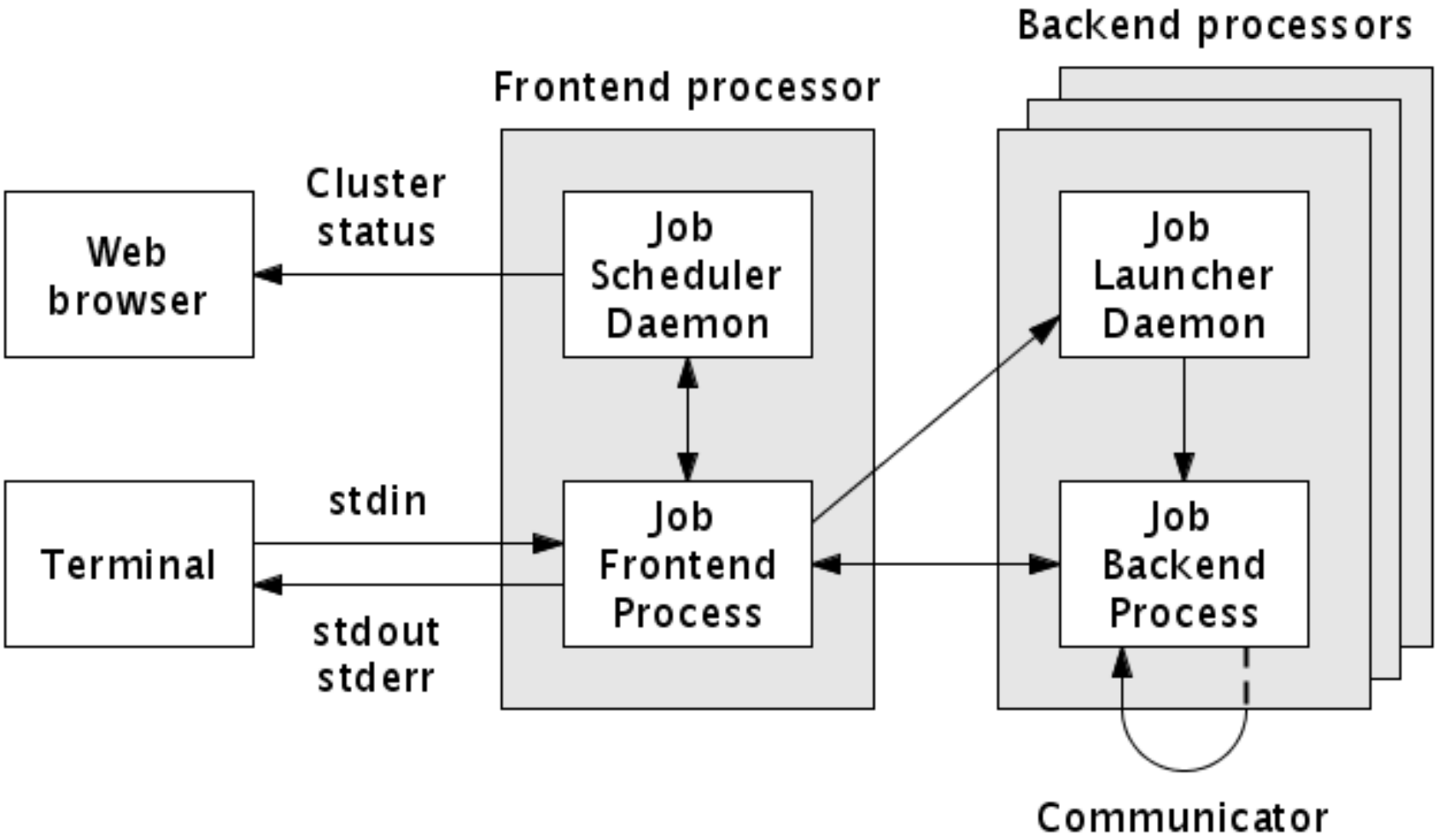
```
static Comm world;
int size, rank;

public static void main (String[] args) {
  Comm.init (args);
  Comm world = Comm.world();
  int size = world.size();
  int rank = world.rank();
  ...
}
```

A First Cluster Parallel Program

- `Comm.init(args)` creates a process on each backend processor that is executing the same `main()` method with the same command-line arguments.
- Each process also creates an object called the *world communicator* that sends and receives messages between backend processes.
- The world communicator has two attributes:
  - size: the number of processes in the group
  - rank: the number (between **0** and ***size*** $-$ **1**) that uniquely identifies a process

Note: There is not one world communicator (shared by all processes); there is one world communicator *per process*.

Cluster Parallel Program Architecture



Cluster Parallel Program Architecture

Some details:

- Launch the parallel program on the frontend processor
- This creates the *job frontend process*.
- This process calls `Comm.init(args)`:
  - Contacts the Job Scheduler Daemon:
    - Asks it to create a new parallel processing job.
    - Job Scheduler Daemon assigns particular backend processors.
  - Contacts each backend processor's Job Launcher Daemon
    - Asks it to create a new *job backend process*, executing the same `main(args)` method.
- Job backend processes establish communication with each other and with the job frontend process.
- Standard I/O of job backend processes are routed through job frontend process.
- Job frontend process terminates when each job backend process has terminated.

Cluster Parallel Program Architecture

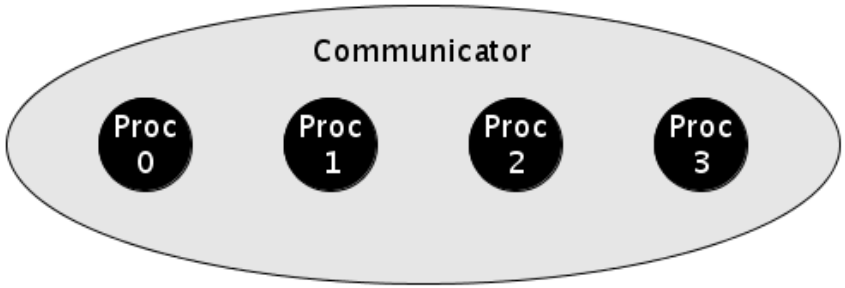
Cluster middleware

- responsible for creating each process on a different processor
- consists of the Job Scheduler Daemon, the Job Launcher Daemon, and the communication layers

Communicators

Communicator:

- an abstraction of the *communication medium* that encompasses a group of processes



- size: the number of processes in the group
- rank: the number (between **0** and ***size*** $-$ **1**) that uniquely identifies a process

Communicators in Parallel Java

`edu.rit.pj.Comm`

- **`static void`** `Comm.init(String[] args);`
  - initializes the PJ communication layer
  - `args` is an array of the program's command line arguments
- **`static`** `Comm Comm.world();`
  - returns a reference to the world communicator
  - the `pj.np` property specifies the world communicator's size
- **`int`** `Comm.size()`
  - returns the communicator's size
- **`int`** `Comm.rank()`
  - returns the calling process's rank within the communicator

Communicators in Parallel Java

```
public static void main (String[] args) throws Exception {
  Comm.init (args);
  Comm world = Comm.world();
  int size = world.size();
  int rank = world.rank();
  ...
}
```

Communication Operations

Point-to-Point Communication

- Transfers data from one process to one other proccess

Collective Communication

- Transfers data among *all* processes
- (Implemented as point-to-point communications.)

Communication Operations

Point-to-Point Communication

- send and receive
- wildcard receive
- nonblocking send and receive
- send-receive

Collective Communication

- broadcast
- flood
- scatter
- gather
- all-gather
- reduce
- all-reduce
- all-to-all
- scan
- barrier

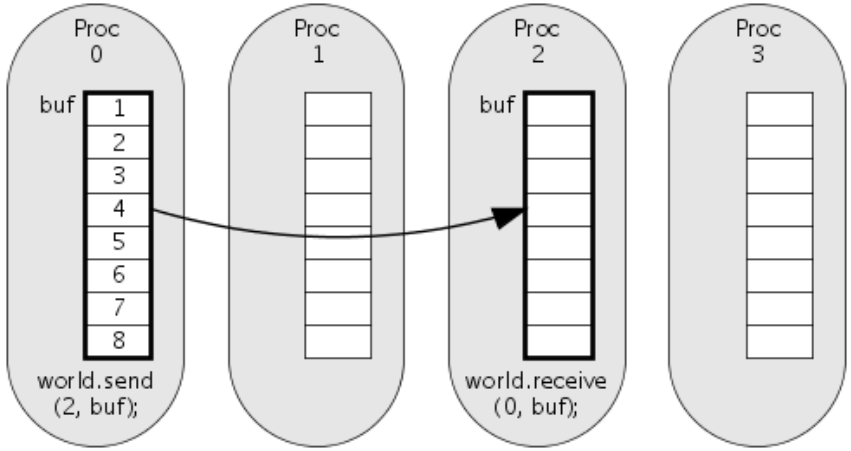
Today, just a brief introduction to communication operations; will study them in detail with applications over next few weeks.

Send and Receive

```
world.send (toRank, buf);

CommStatus status = world.receive(fromRank, buf);
```

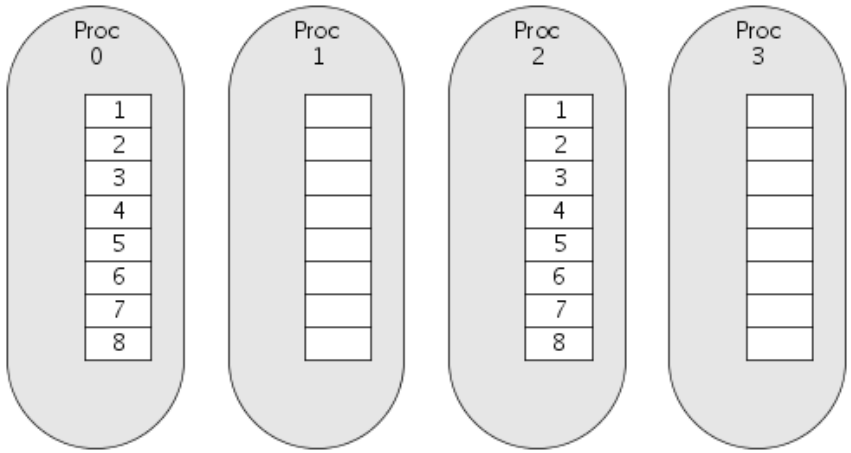
Send and Receive

```
world.send (toRank, buf);

CommStatus status = world.receive(fromRank, buf);
```



Send and Receive

```
world.send (toRank, buf);

CommStatus status = world.receive(fromRank, buf);
```



Wildcard Receive

```
CommStatus status = world.receive(null, buf);
```

Nonblocking Send and Receive

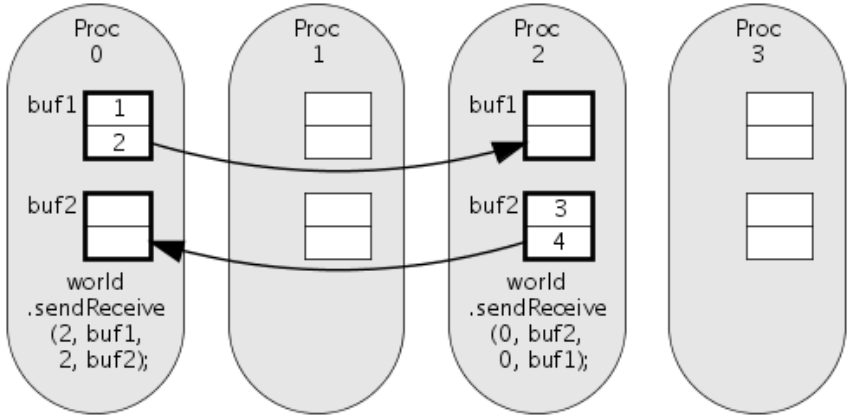
```
CommRequest request = new CommRequest();
world.send (toRank, buf, request);
// Other processing
request.waitForFinish();

CommRequest request = new CommRequest();
CommStatus status = world.receive(fromRank, buf, request);
// Other processing
request.waitForFinish();
```

Send-Receive

```
CommStatus status =
  world.sendReceive (toRank, srcBuf,
                     fromRank, dstBuf);
```

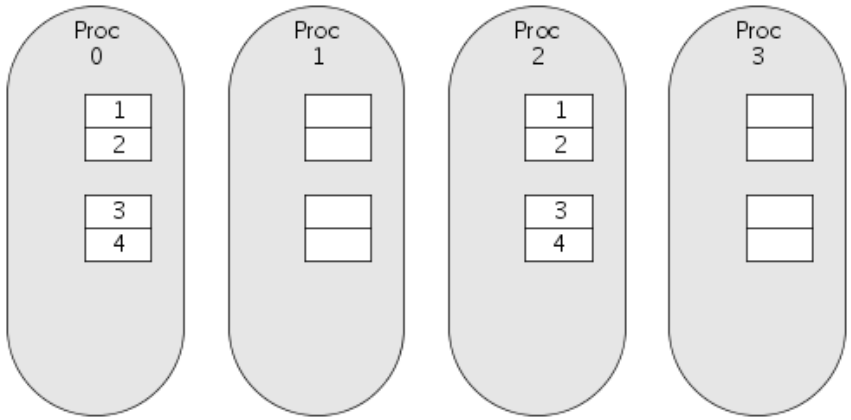
Send-Receive

```
CommStatus status =
  world.sendReceive (toRank, srcBuf,
                     fromRank, dstBuf);
```



Send-Receive

```
CommStatus status =
  world.sendReceive (toRank, srcBuf,
                     fromRank, dstBuf);
```

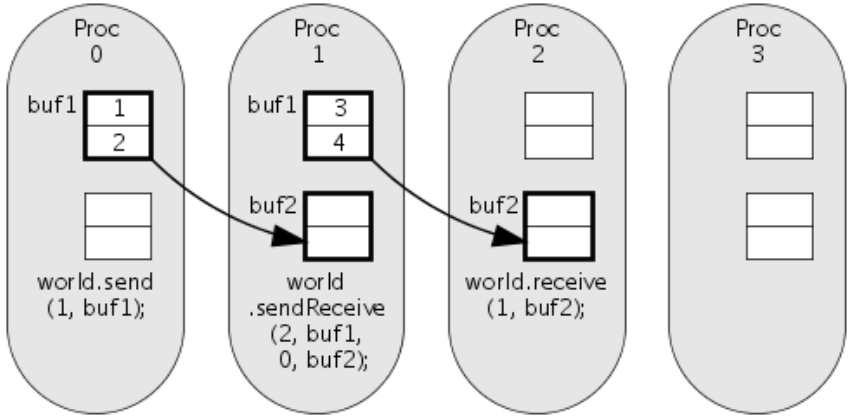


Send-Receive

```
CommStatus status =
  world.sendReceive (toRank, srcBuf,
                     fromRank, dstBuf);
```

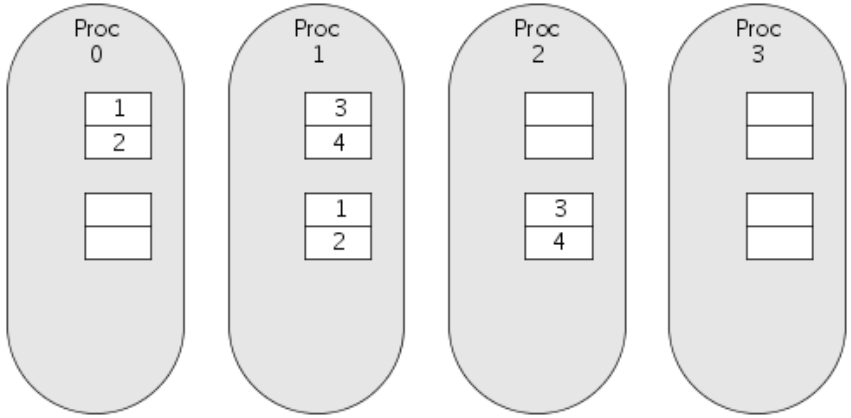
Send-Receive

```
CommStatus status =
  world.sendReceive (toRank, srcBuf,
                     fromRank, dstBuf);
```



Send-Receive

```
CommStatus status =
  world.sendReceive (toRank, srcBuf,
                     fromRank, dstBuf);
```



Nonblocking Send-Receive

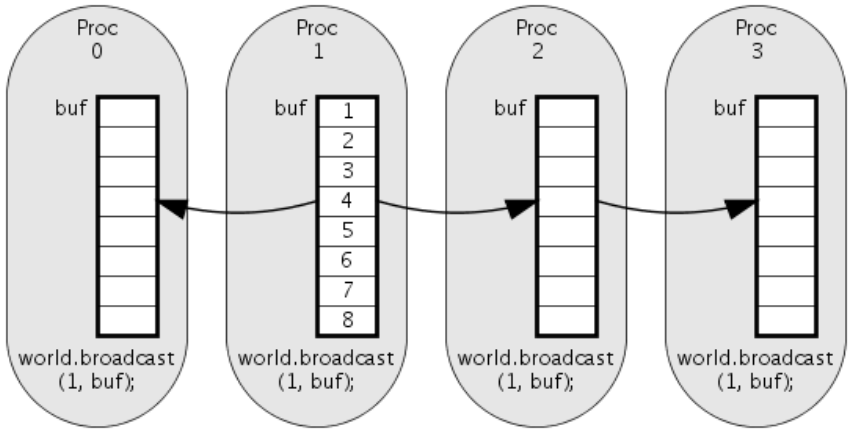
```
CommRequest request = new CommRequest()
world.sendReceive (toRank, srcBuf,
                   fromRank, dstBuf,
                   request);
// Other processing
CommStatus status = request.waitForFinish();
```

Broadcast

```
world.broadcast (root, buf);
```

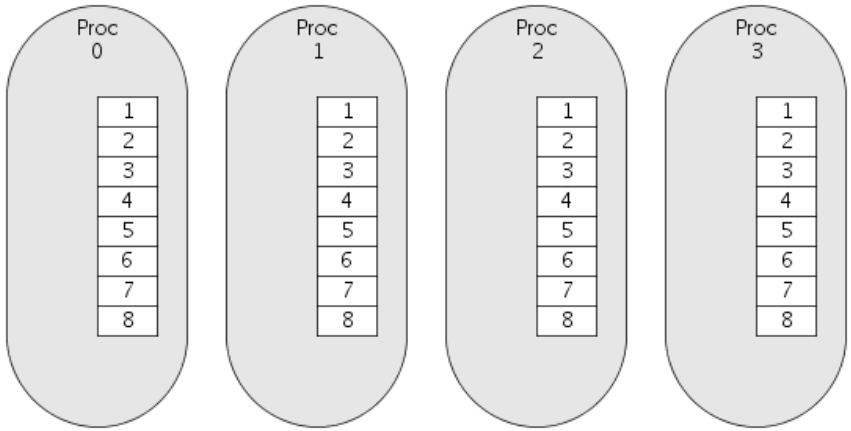
Broadcast

```
world.broadcast (root, buf);
```



Broadcast

```
world.broadcast (root, buf);
```

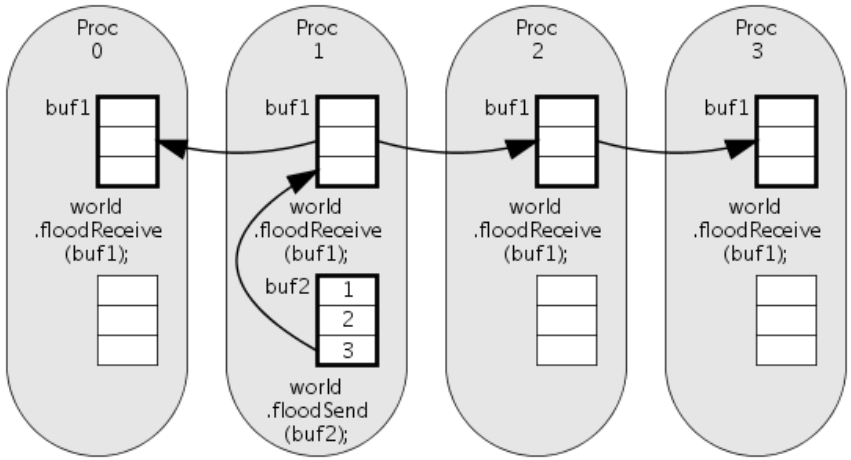


Flood

```
CommStatus status = world.floodReceive (dstBuf);

world.floodSend (srcBuf);
```

Flood

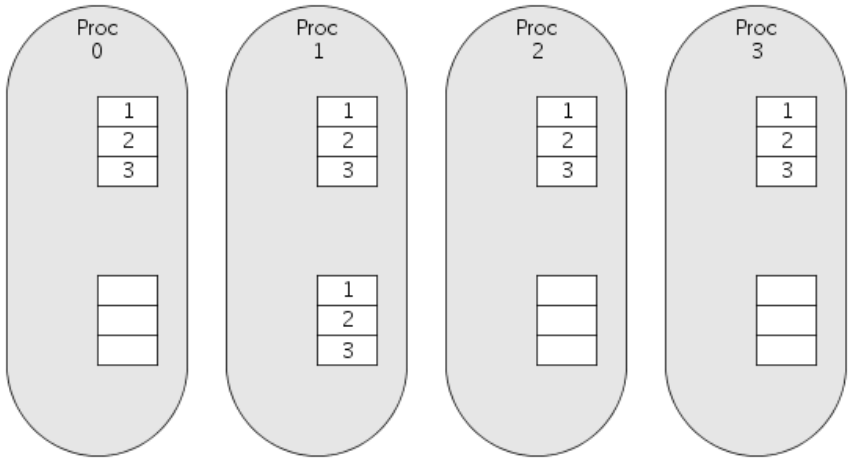
```
CommStatus status = world.floodReceive (dstBuf);

world.floodSend (srcBuf);
```



The `floodSend()` and `floodReceive()` methods block until every process has called `floodReceive()` and one process has called `floodSend()`.

Flood

```
CommStatus status = world.floodReceive (dstBuf);

world.floodSend (srcBuf);
```



The `floodSend()` and `floodReceive()` methods block until every process has called `floodReceive()` and one process has called `floodSend()`.

Nonblocking Flood

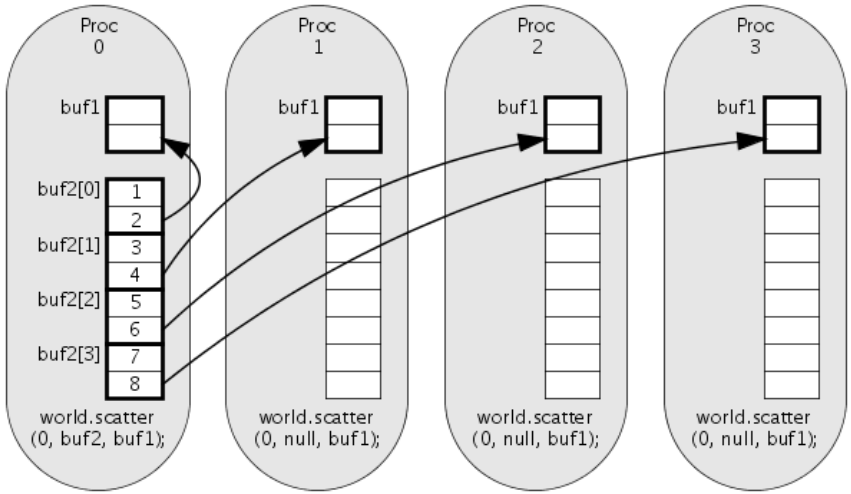
```
CommRequest request = new CommRequest()
world.floodReceive (dstBuf, request);
// Other processing
CommStatus status = request.waitForFinish();

CommRequest request = new CommRequest()
world.floodSend (srcBuf, request);
// Other processing
CommStatus status = request.waitForFinish();
```

Scatter

```
world.scatter (root, srcBufArray, dstBuf);
```

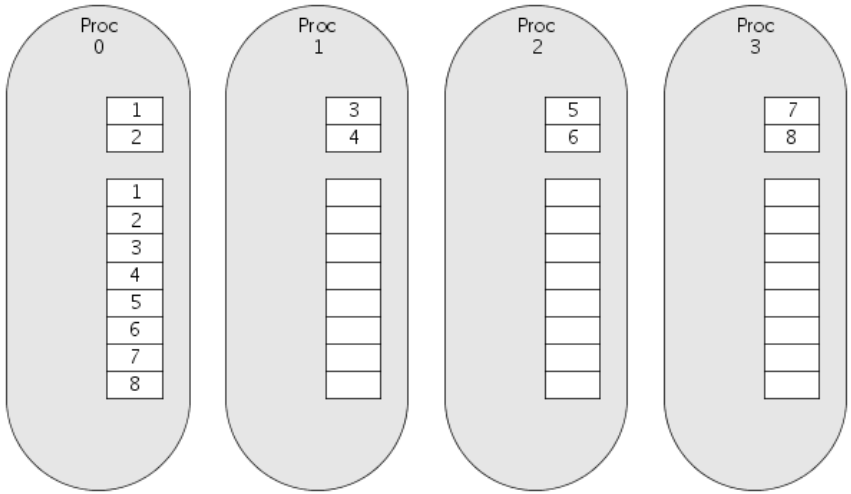
Scatter

```
world.scatter (root, srcBufArray, dstBuf);
```



Scatter

```
world.scatter (root, srcBufArray, dstBuf);
```

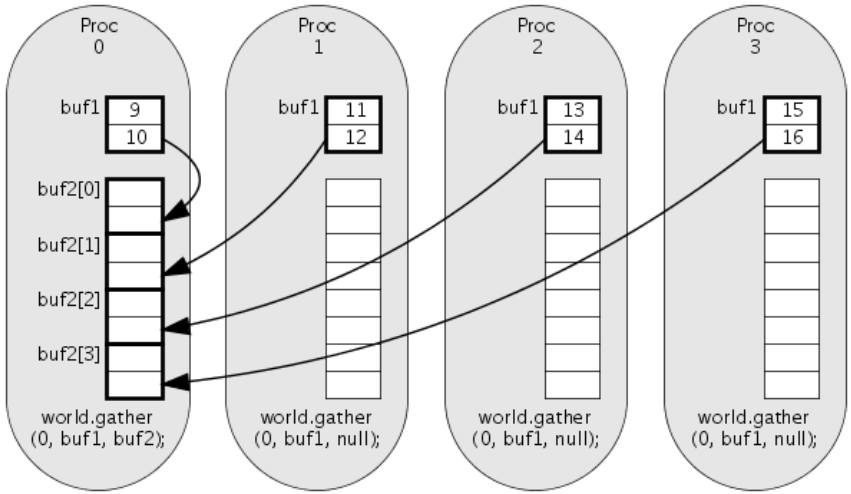


Gather

```
world.gather (root, srcBuf, dstBufArray);
```

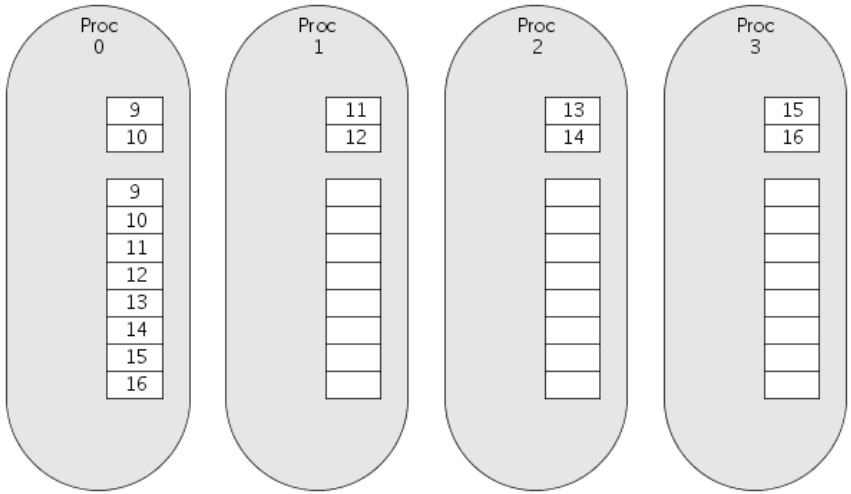
Gather

```
world.gather (root, srcBuf, dstBufArray);
```



Gather

```
world.gather (root, srcBuf, dstBufArray);
```

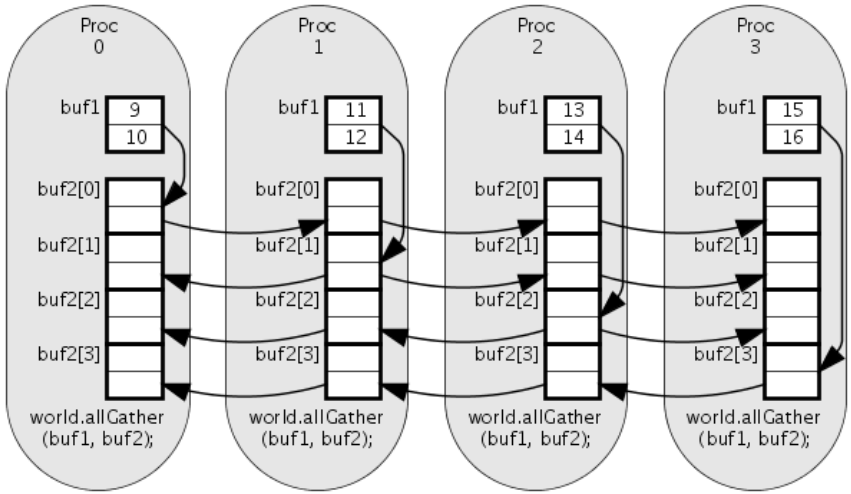


All-Gather

```
world.allGather (srcBuf, dstBufArray);
```

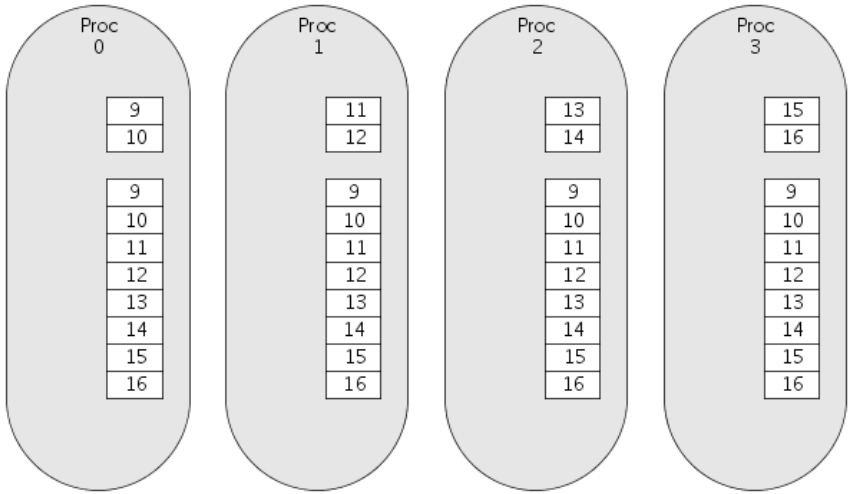
All-Gather

```
world.allGather (srcBuf, dstBufArray);
```



All-Gather

```
world.allGather (srcBuf, dstBufArray);
```

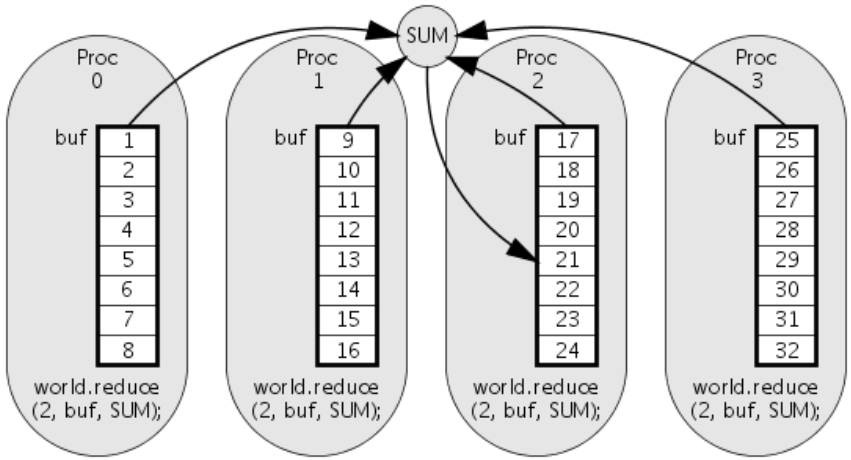


Reduce

```
world.reduce (root, buf, op);
```

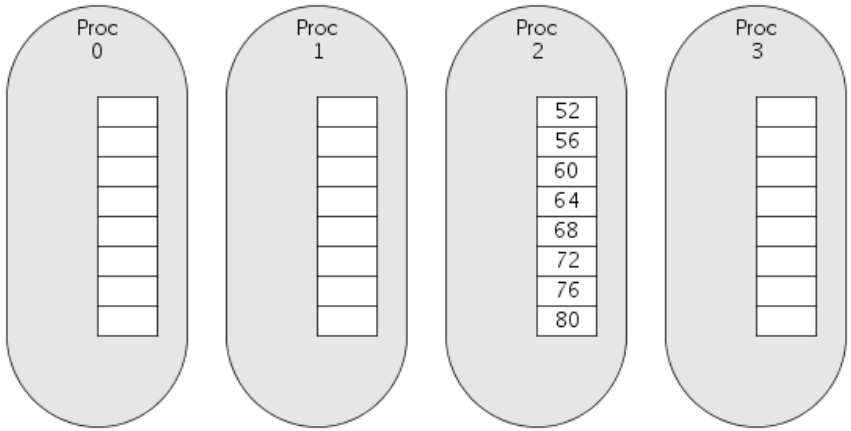
Reduce

```
world.reduce (root, buf, op);
```



Reduce

```
world.reduce (root, buf, op);
```

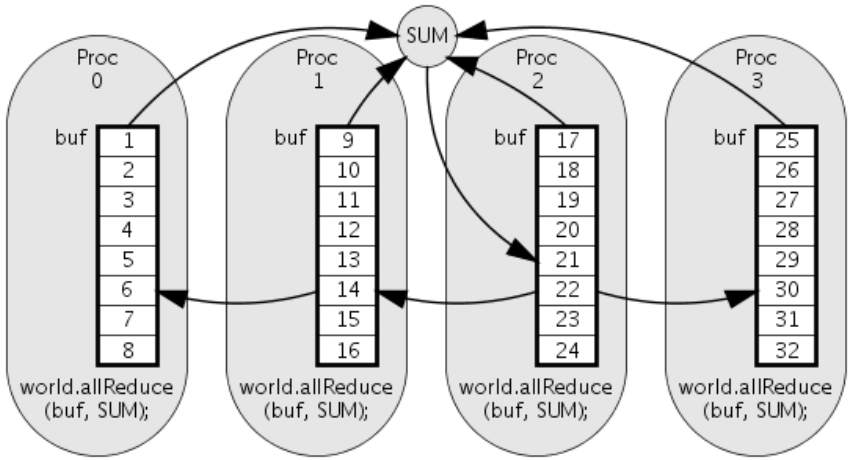


All-Reduce

```
world.allReduce (buf, op);
```

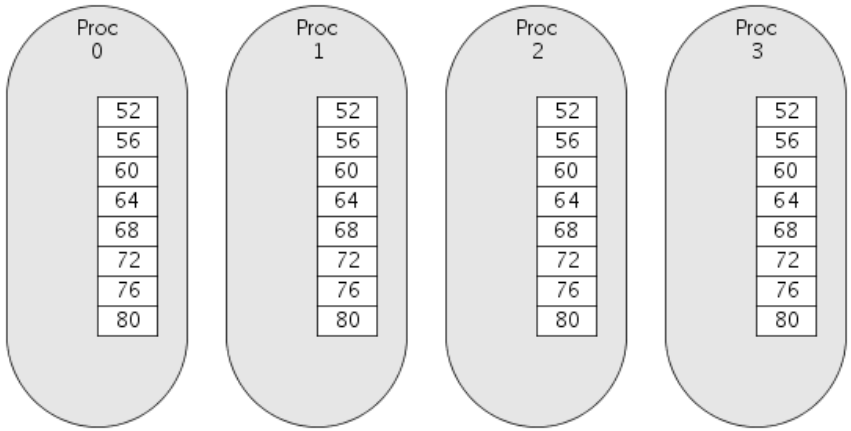
All-Reduce

```
world.allReduce (buf, op);
```



All-Reduce

```
world.allReduce (buf, op);
```

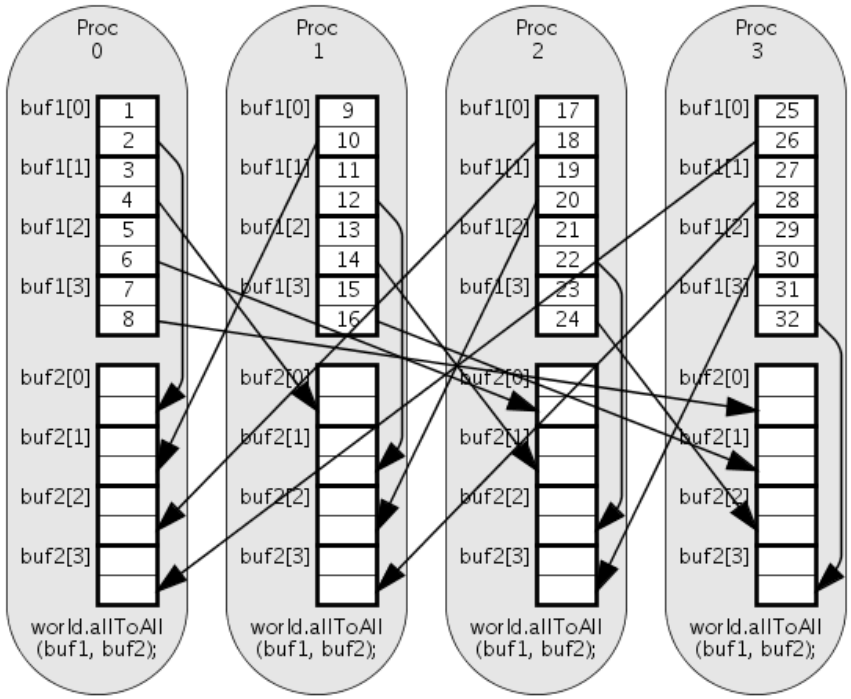


All-to-All

```
world.allToAll (srcBufArray, dstBufArray);
```

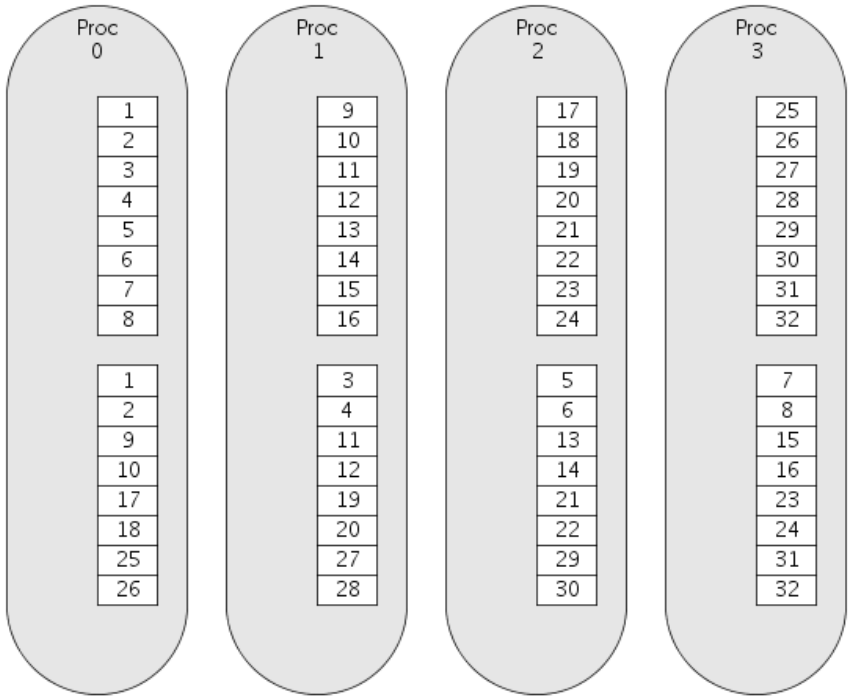
All-to-All

```
world.allToAll (srcBufArray, dstBufArray);
```



All-to-All

```
world.allToAll (srcBufArray, dstBufArray);
```

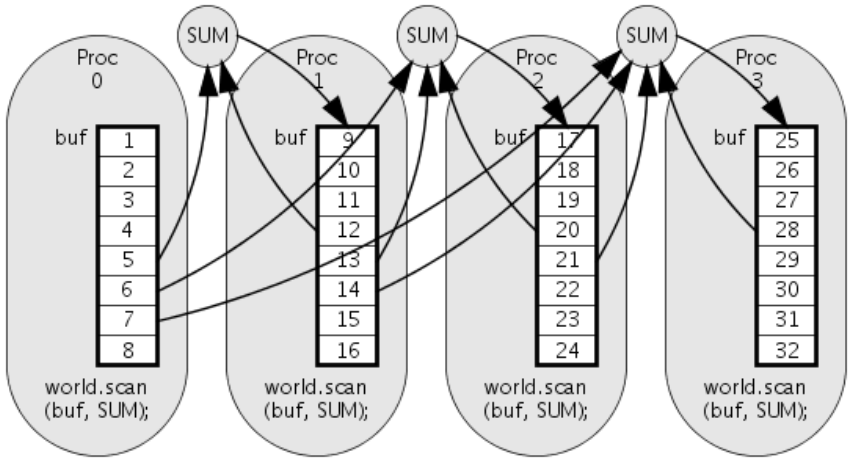


Scan

```
world.scan (buf, op);
```

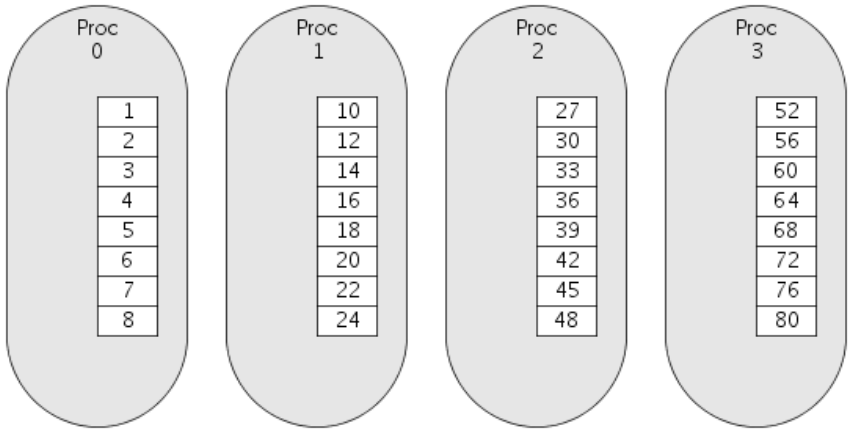
Scan

```
world.scan (buf, op);
```



Scan

```
world.scan (buf, op);
```

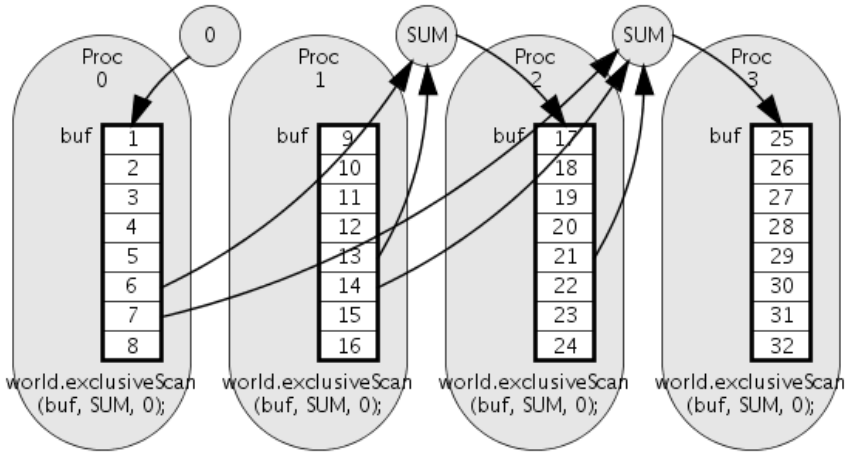


Exclusive Scan

```
world.exclusiveScan (buf, op);
```

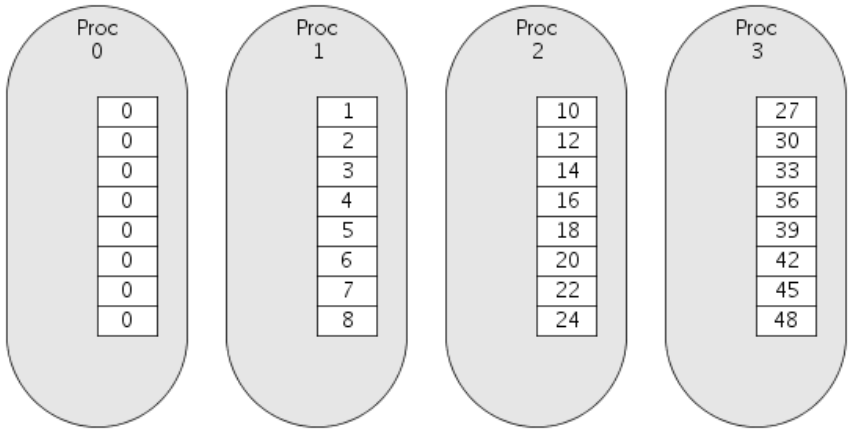
Exclusive Scan

```
world.exclusiveScan (buf, op);
```



Exclusive Scan

```
world.exclusiveScan (buf, op);
```



Barrier

```
world.barrier();
```

Communication Operations

Point-to-Point Communication

- send and receive
- wildcard receive
- nonblocking send and receive
- send-receive

Collective Communication

- broadcast
- flood
- scatter
- gather
- all-gather
- reduce
- all-reduce
- all-to-all
- scan
- barrier

Today, just a brief introduction to communication operations; will study them in detail with applications over next few weeks.

Cluster Parallel Program Design

Revisit AES Partial Key Search:

- Inputs: a plaintext block `p`, a ciphertext block `c`, a partial key `k'` with **256** $-$ ***n*** bits of the `k` (which produced `c` from `p`)
- Outputs: the complete key `k` (which produced `c` from `p`)
- Algorithm: exhaustive search

Cluster Parallel Program Design

Revisit AES Partial Key Search:

- Inputs: a plaintext block `p`, a ciphertext block `c`, a partial key `k'` with **256** $-$ ***n*** bits of the `k` (which produced `c` from `p`)
- Outputs: the complete key `k` (which produced `c` from `p`)
- Algorithm: exhaustive search

SMP parallel program:

- one process with many threads
- shared variables and synchronization
- per-thread variables and cache interference

Cluster Parallel Program Design

Revisit AES Partial Key Search:

- Inputs: a plaintext block `p`, a ciphertext block `c`, a partial key `k'` with **256** $-$ ***n*** bits of the `k` (which produced `c` from `p`)
- Outputs: the complete key `k` (which produced `c` from `p`)
- Algorithm: exhaustive search

Cluster parallel program:

- many processes
- no shared variables (and no synchronization)
- no per-thread variables (and no cache interference)
- send and receive messages to communicate data between processes

FindKeyClu

code/FindKeyClu.java

`FindKeyClu` Experiments

`tardis`: frontend computer

- UltraSPARC-IIe CPU, 650 MHz clock, 512 MB main memory

`dr00` through `dr09`: backend computers

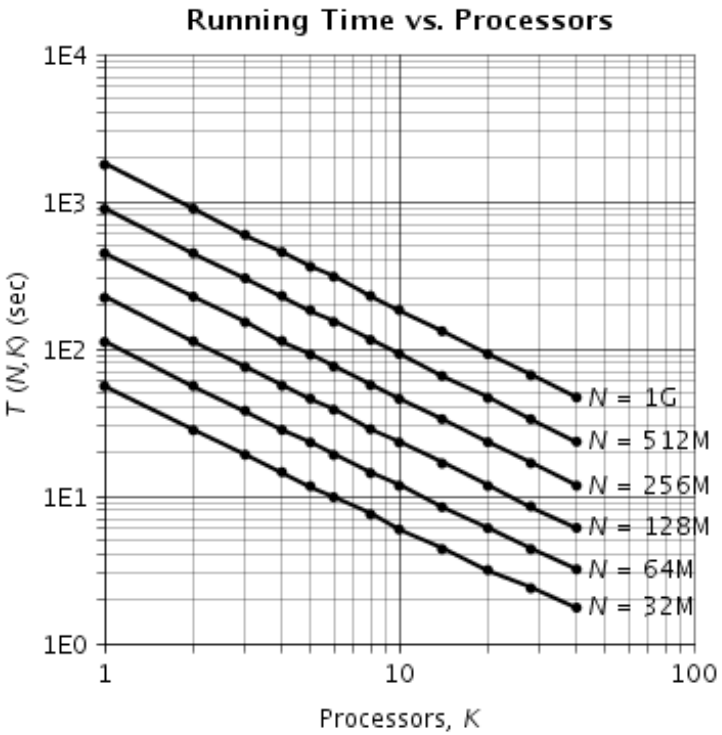
- AMD Opteron 2218 dual-core CPUs (four processors), 2.6 GHz clock, 8 GB main memory

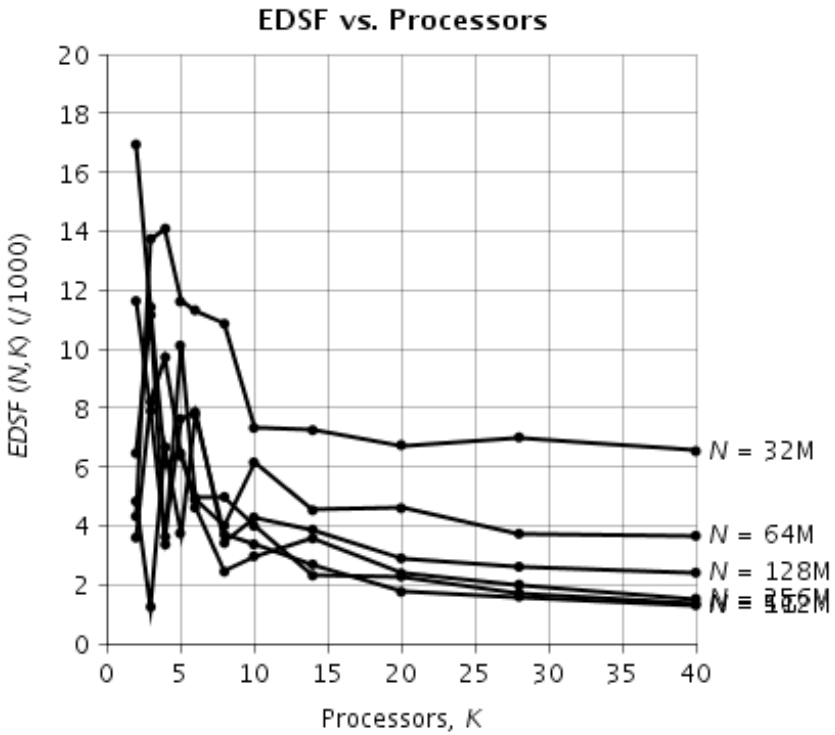
1-Gbps switched Ethernet backend interconnection network
Aggregate 40 processors, 104 GHz clock, 80 GB main memory

A hybrid SMP cluster parallel computer,
but treated as a plain cluster of 40 nodes.

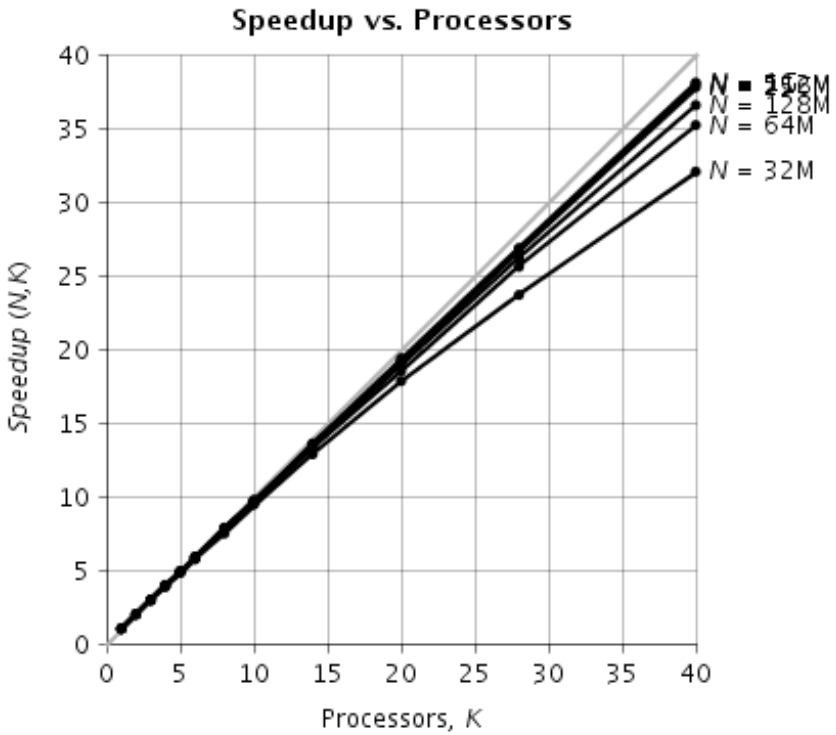
- Run up to 4 processes on each backend,
  but each process will run in its own separate address space
  with no shared memory.

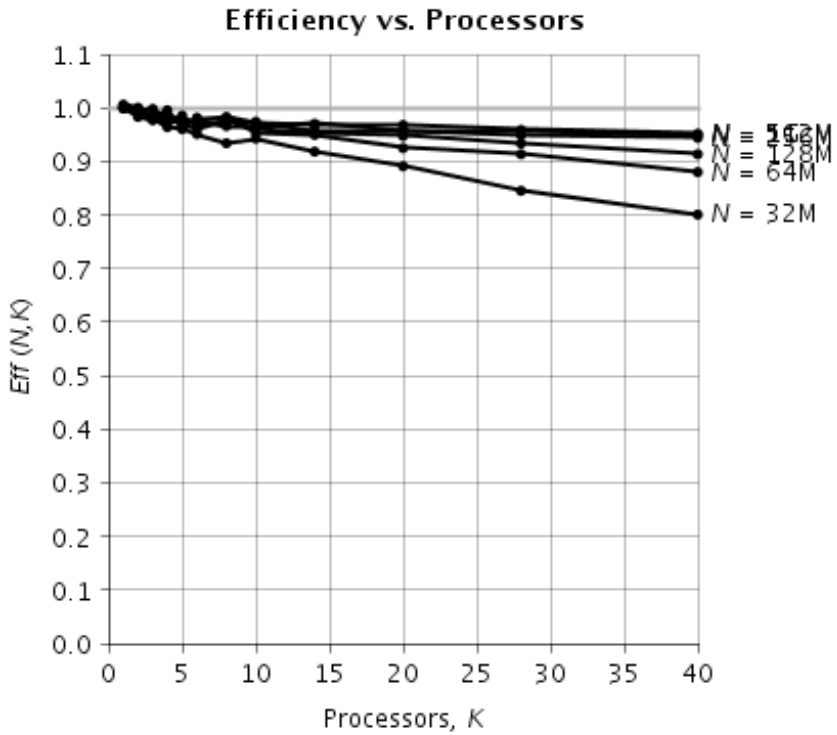
`FindKeyClu` Running Time and EDSF





FindKeyClu Speedup and Efficiency





FindKeyClu and Gustafson's Law

Sequential fraction constant for fixed problem size,
but sequential fraction decreases as problem size increases.

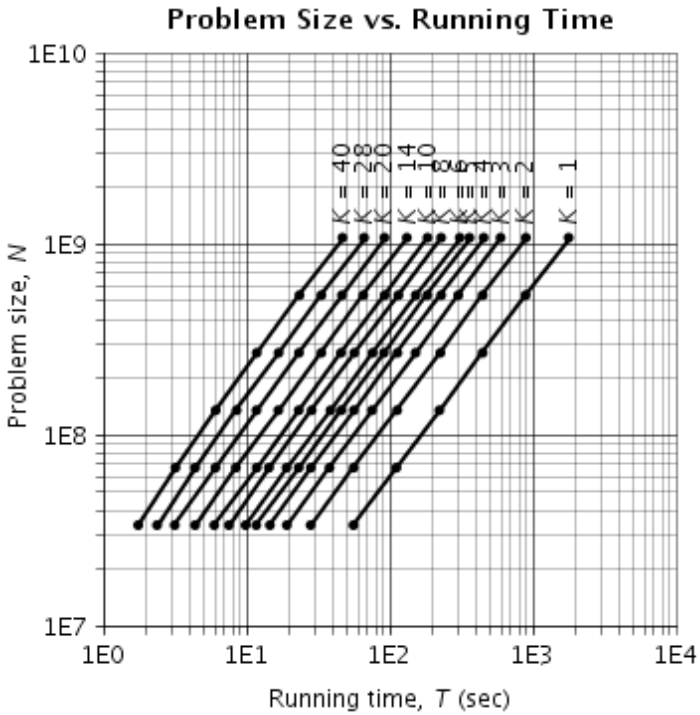
Recall the more realistic running time model:

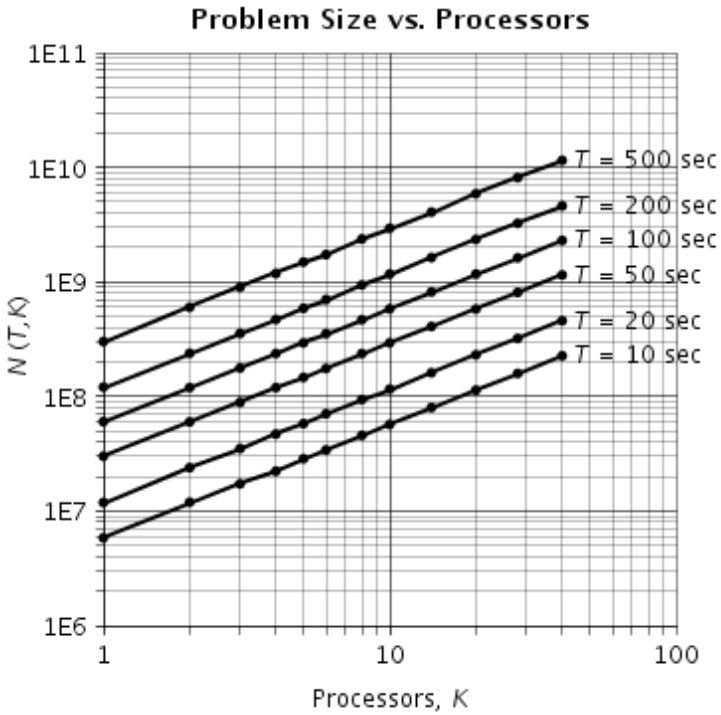
$$T(N, K) = (a + b \cdot N) + \frac{1}{K}(c + d \cdot N)$$

Fitting the experimental data to this model yields:

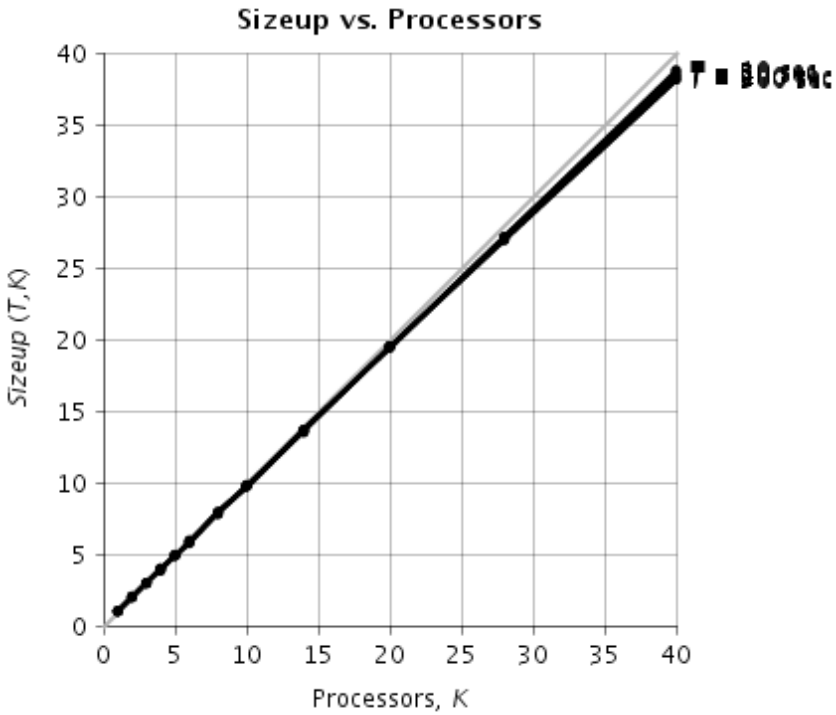
$$a = 312 \text{ msec}$$
$$b = 5.00 \times 10^{-6} \text{ msec}$$
$$c = 312 \text{ msec}$$
$$d = 1.65 \times 10^{-3} \text{ msec}$$

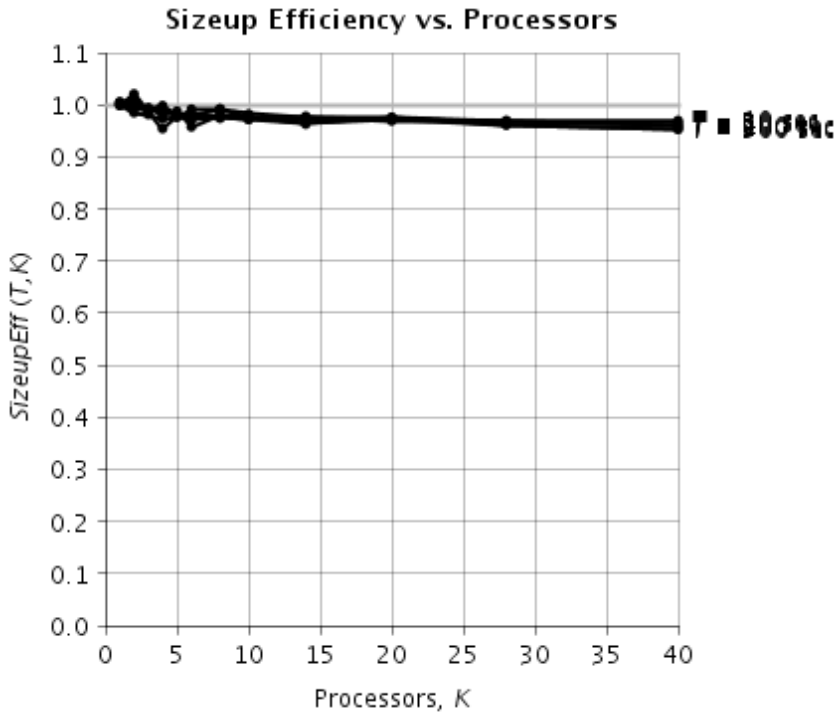
`FindKeyClu` Problem Size



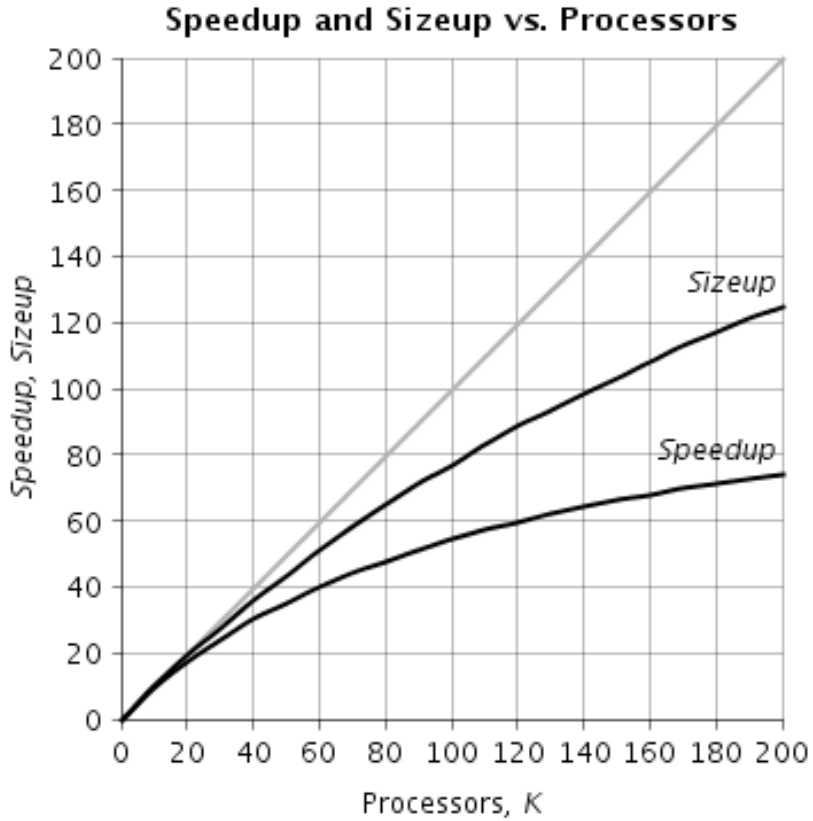


`FindKeyClu` Sizeup and Sizeup Efficiency





FindKeyClu Speedup vs Sizeup



$$N = 32\text{M}$$

$$F = 8.54 \times 10^{-3} \qquad MaxSpeedup = \frac{1}{F} = 117$$

$$G = \frac{b}{d} = 3.03 \times 10^{-3} \qquad MaxSizeup = 1 + \frac{1}{G} = 331$$

`FindKeyClu` Early Loop Exit

Change the cluster parallel program to stop as soon as the key is found.

Need *all* the processes to exit their loops
as soon as *any* process find the key.

`FindKeyClu` Early Loop Exit

Change the cluster parallel program to stop as soon as the key is found.

Need *all* the processes to exit their loops
as soon as *any* process find the key.

Which communication operation lets one process notify all other processes?

`FindKeyClu` Early Loop Exit

Change the cluster parallel program to stop as soon as the key is found.

Need *all* the processes to exit their loops
as soon as *any* process find the key.

Which communication operation lets one process notify all other processes?

- flood

`code/FindKeyClu2.java`