Parallel Computing I

Cluster: Communication and Load Balancing

Looking Back, Looking Forward

Last week:

- Mid-term exam
- Introduction to cluster parallel programming
  - cluster architecture review
  - cluster middleware
  - communication operations

This week:

- communication buffers and data slicing
- load balancing
- communication overhead

Buffers

A cluster parallel program uses a buffer object to designate a data source or data destination.

Different abstract base classes provide buffers for each of Java's primitive and nonprimitive types.

Java Type	Buffer Class
`boolean`	`edu.rit.mp.BooleanBuf`
`byte`	`edu.rit.mp.ByteBuf`
`char`	`edu.rit.mp.CharBuf`
`double`	`edu.rit.mp.DoubleBuf`
`float`	`edu.rit.mp.FloatBuf`
`int`	`edu.rit.mp.IntBuf`
`long`	`edu.rit.mp.LongBuf`
`short`	`edu.rit.mp.ShortBuf`
`Object`	`edu.rit.mp.ObjectBuf`

Buffers

A cluster parallel program uses a buffer object
to designate a data source or data destination.

Different abstract base classes provide buffers
for each of Java's primitive and nonprimitive types.

Create a buffer object using a static factory method
in the appropriate buffer class.

Object Buffers

An `ObjectBuf` is for data items of any *non-primitive* type.

Parallel Java uses *Java Object Serialization* to send and receive objects.

- In an outgoing message,
  the objects are serialized (into a sequence of bytes)
  and then sent.
- In an incoming message,
  the sequence of bytes are received
  and then deserialized (into objects).

Such objects must implement `java.io.Serializable`.

- Most classes in Java Collections Framework are serializable.

Single-Item Buffers

```
IntegerItemBuf buf = IntegerBuf.buffer();

buf.item = 42;
System.out.println(buf.item);
```

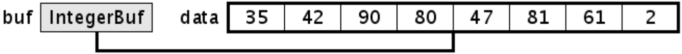
Single-Item Buffers

```
IntegerItemBuf buf = IntegerBuf.buffer();

buf.item = 42;
System.out.println(buf.item);
```

Note: A `ObjectItemBuf` holds a "single-item" of type `Object`,
which can have multiple fields of various types
or can be an entire collection data structure.

Buffer for an Array

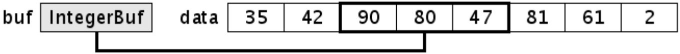
```
int[] data = new int[8];
IntegerBuf buf = IntegerBuf.buffer(data);
```



View the buffer object as a "handle" that refers to the data items.
To access the data items, get or set the elements of the array.

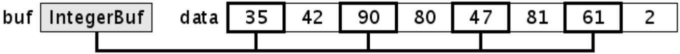
Buffer for an Array Slice

```
int[] data = new int[8];
Range slicerange = new Range(2, 4);
IntegerBuf buf = IntegerBuf.sliceBuffer(data, slicerange);
```

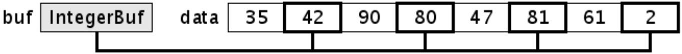


Buffer for an Array Slice

```
int[] data = new int[8];
Range evenrange = new Range(0, 6, 2);
IntegerBuf buf = IntegerBuf.sliceBuffer(data, evenrange);
```

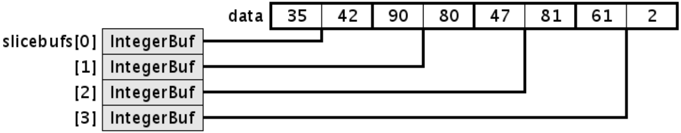



```
int[] data = new int[8];
Range oddrange = new Range(1, 7, 2);
IntegerBuf buf = IntegerBuf.sliceBuffer(data, oddrange);
```



Buffers to Partition an Array

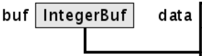
```
int[] data = new int[8];
Range[] sliceranges = new Range(0,7).subranges(4);
IntegerBuf[] slicebufs =
   IntegerBuf.sliceBuffers(data, sliceranges);
```



Useful for scatter or gather operations,
where we need an array of buffers,
one buffer for each process in the cluster parallel program.

Buffer for a Matrix

```
int[][] data = new int[4][8];
IntegerBuf buf = IntegerBuf.buffer(data);
```



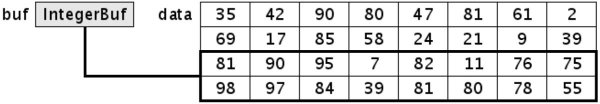
35	42	90	80	47	81	61	2
69	17	85	58	24	21	9	39
81	90	95	7	82	11	76	75
98	97	84	39	81	80	78	55

View the buffer object as a "handle" that refers to the data items.
To access the data items, get or set the elements of the matrix.

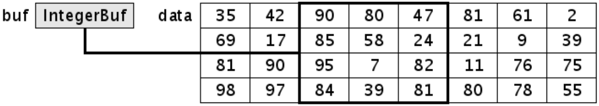
Elements of the matrix communicated in *row major order*.

Buffer for a Matrix Slice

```
int[][] data = new int[4][8];
Range rowrange = new Range(2, 3);
IntegerBuf buf = IntegerBuf.rowSliceBuffer(data, rowrange);
```

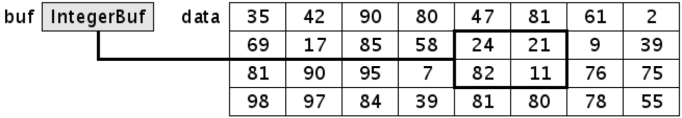



```
int[][] data = new int[4][8];
Range colrange = new Range(2, 4);
IntegerBuf buf = IntegerBuf.colSliceBuffer(data, colrange);
```



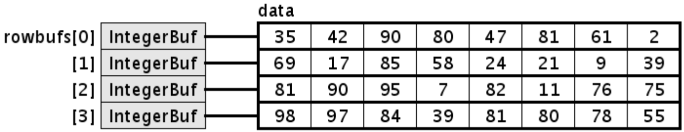
Buffer for a Matrix Slice

```
int[][] data = new int[4][8];
Range rowrange = new Range(1, 2);
Range colrange = new Range(4, 5);
IntegerBuf buf =
  IntegerBuf.patchBuffer(data, rowrange, colrange);
```



Buffers to Partition an Matrix

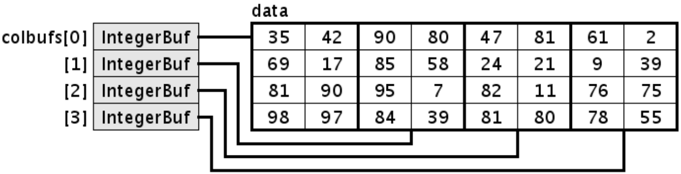
```
int[] data = new int[4][8];
Range[] rowranges = new Range(0,3).subranges(4);
IntegerBuf[] rowbufs =
   IntegerBuf.rowSliceBuffers(data, rowranges);
```



Useful for scatter or gather operations,
where we need an array of buffers,
one buffer for each process in the cluster parallel program.

Buffers to Partition an Matrix

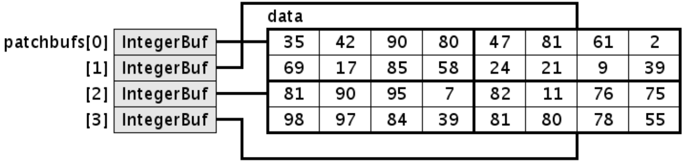
```
int[] data = new int[4][8];
Range[] colranges = new Range(0,7).subranges(4);
IntegerBuf[] colbufs =
   IntegerBuf.colSliceBuffers(data, colranges);
```



Useful for scatter or gather operations,
where we need an array of buffers,
one buffer for each process in the cluster parallel program.

Buffers to Partition an Matrix

```
int[] data = new int[4][8];
Range[] rowranges = new Range(0,3).subranges(2);
Range[] colranges = new Range(0,7).subranges(2);
IntegerBuf[] patchbufs =
   IntegerBuf.patchBuffers(data, rowranges, colranges);
```



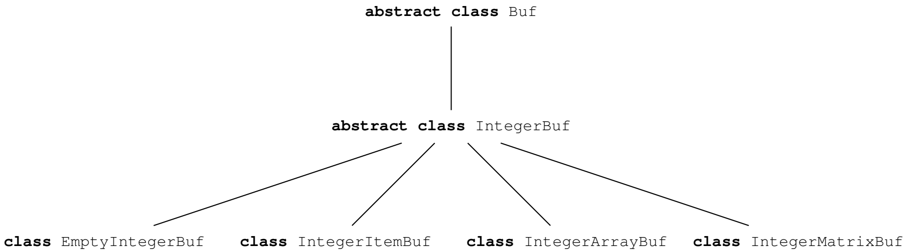
Useful for scatter or gather operations,
where we need an array of buffers,
one buffer for each process in the cluster parallel program.

Buffers

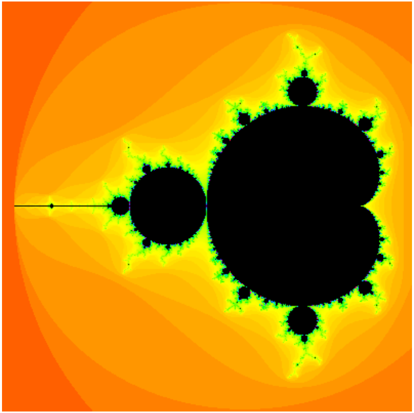
To create an `IntegerBuf`,
use one of the following static factory methods:

- `IntegerBuf emptyBuffer()`
- `IntegerItemBuf buffer()`
- `IntegerItemBuf buffer (int)`
- `IntegerBuf buffer (int[])`
- `IntegerBuf sliceBuffer (int[], Range)`
- `IntegerBuf[] sliceBuffers (int[], Range[])`
- `IntegerBuf buffer (int[][])`
- `IntegerBuf rowSliceBuffer (int[][], Range)`
- `IntegerBuf[] rowSliceBuffers (int[][], Range[])`
- `IntegerBuf colSliceBuffer (int[][], Range)`
- `IntegerBuf[] colSliceBuffers (int[][], Range[])`
- `IntegerBuf patchBuffer (int[][], Range, Range)`
- `IntegerBuf[] patchBuffers (int[][], Range[], Range[])`

Buffers


abstract class Buf
abstract class IntegerBuf
class EmptyIntegerBuf
class IntegerItemBuf
class IntegerArrayBuf
class IntegerMatrixBuf

Mandelbrot Set



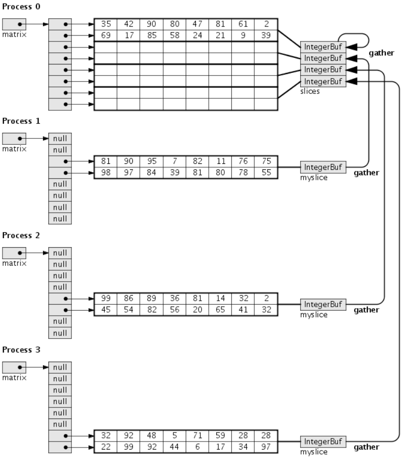
How do we convert the sequential Mandelbrot set program
to a *cluster* parallel Mandelbrot set program?

MandelbrotSetClu.java

`code/MandelbrotSetClu.java`

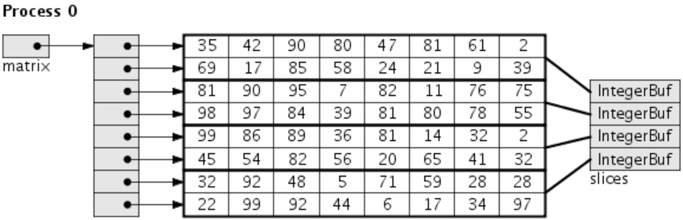
- (implicitly) scatter the rows among the processes
- in parallel, each process computes pixel data for its rows
- gather the rows into one process
- write the rows into an image file

MandelbrotSetClu.java

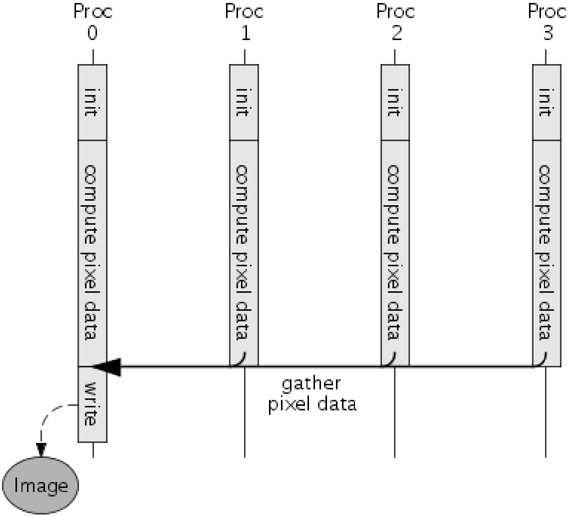


Why does every process have a `matrix` with `height` elements?

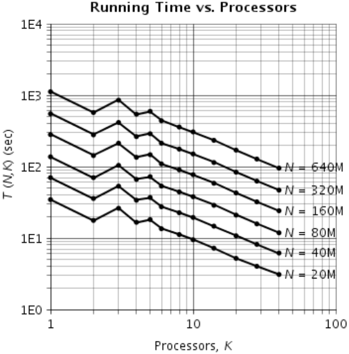
MandelbrotSetClu.java

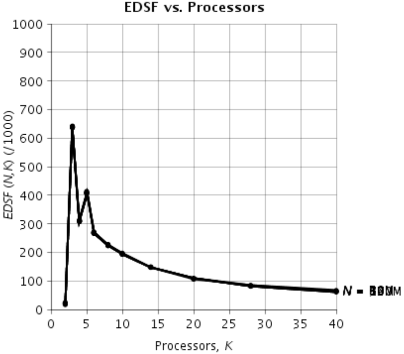


MandelbrotSetClu.java

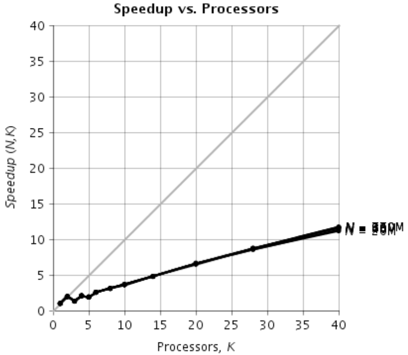


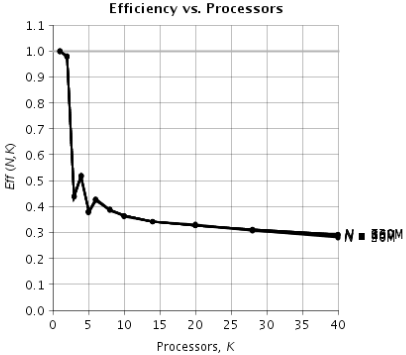
MandelbrotSetClu Running Time and EDSF



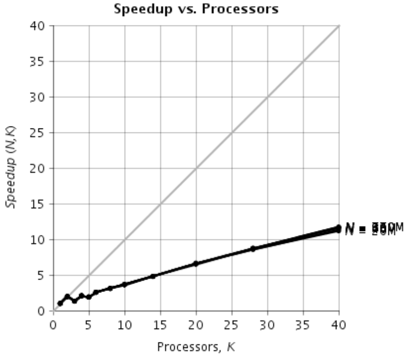


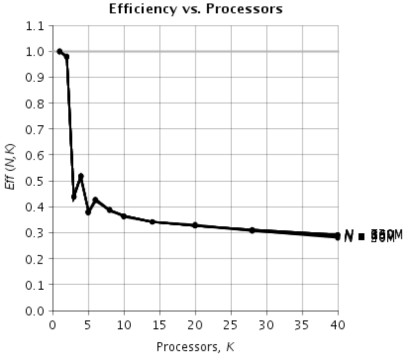
MandelbrotSetClu Speedup and Efficiency





`MandelbrotSetClu` Speedup and Efficiency





We recognize the pattern at low numbers of processors;
explain the behavior at high numbers of processors.

MandelbrotSetClu

Like our first SMP parallel program for Mandelbrot set,
the running times fail to diminish in proportion to $1/K$.

- The speedups and efficiencies are far from ideal.

Like our first SMP parallel program for Mandelbrot set,
the problem is the *unbalanced load* among processors.

- Should not divide the pixel data matrix into equal-sized slices.

How to balance load in a cluster parallel program?

Master-Worker Pattern

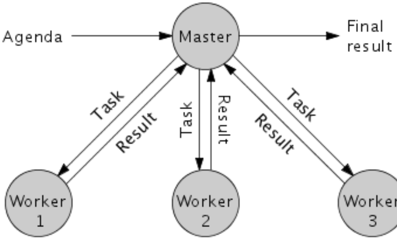
The *Master-Worker* pattern is designed to balance load.

- The master sends tasks one at a time to the workers.
- The worker calculates the pixel data for the assigned slice and sends the calculated pixel data to the master.
- The master accumulates the pixel data from all slices into its own pixel data matrix.

Master-Worker Pattern

The *Master-Worker* pattern is designed to balance load.

- The master sends tasks one at a time to the workers.
- The worker calculates the pixel data for the assigned slice and sends the calculated pixel data to the master.
- The master accumulates the pixel data from all slices into its own pixel data matrix.



Master-Worker Pattern

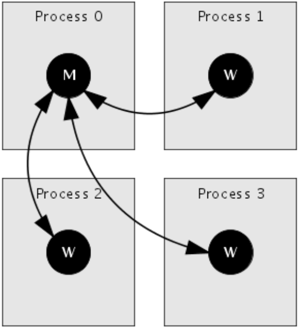
In SMP parallel programs, an `IntegerForLoop` partitions the range according to the specified schedule.

In cluster parallel programs, the master process partitions the range according to the specified scheule, using methods of the `IntegerSchedule` class:

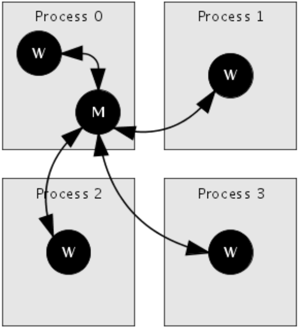
```
void IntegerSchedule.start(int K, Range theLoopRange);

Range IntegerSchedule.next(int theThreadIndex);
```

Master-Worker Pattern

The master process requires some computational resources.
Two choices for allocating master & worker processes among cluster nodes.



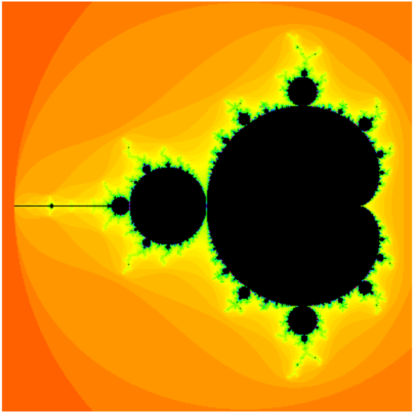
1 master and $K - 1$ workers



1 master and K workers

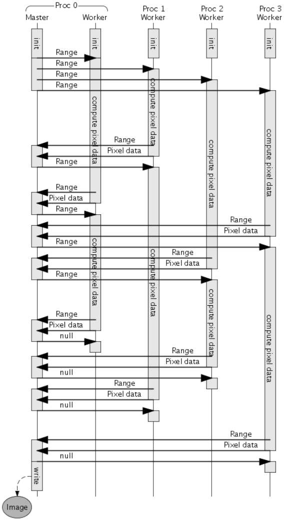
Generally prefer the second choice; why is this justified?

MandelbrotSetClu2.java

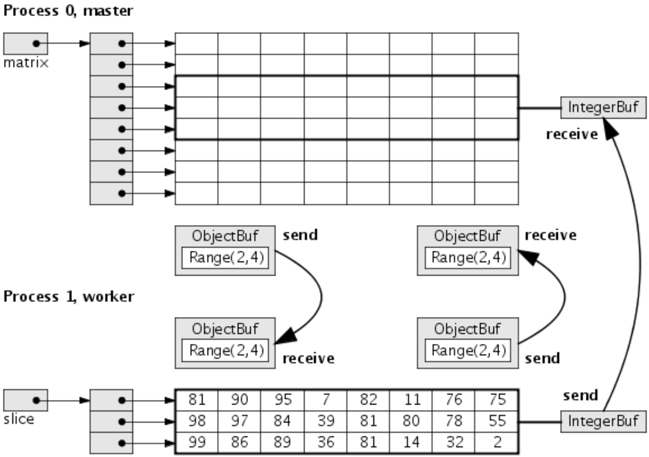


How do we design a *cluster* parallel Mandelbrot set program to use the master-worker pattern?

MandelbrotSetClu2.java



MandelbrotSetClu2.java



Message Tags

Three kinds of messages:

- message sent to a worker containing a range
- message sent to the master containing a range
- message sent to the master containing pixel data

Useful to distinguish the various kinds of messages.

Message Tags

Three kinds of messages:

- message sent to a worker containing a range
- message sent to the master containing a range
- message sent to the master containing pixel data

Useful to distinguish the various kinds of messages.

Parallel Java provides *message tags* for this purpose:

```
world.send (toRank, tag, buffer);

world.receive (fromRank, tag, buffer);
```

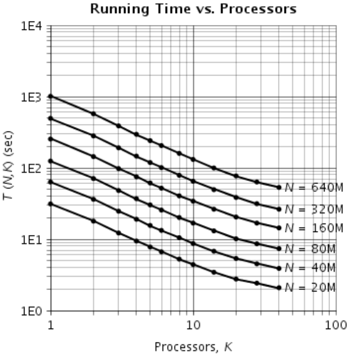
A `receive` with a tag will only match a `send` with the same tag.

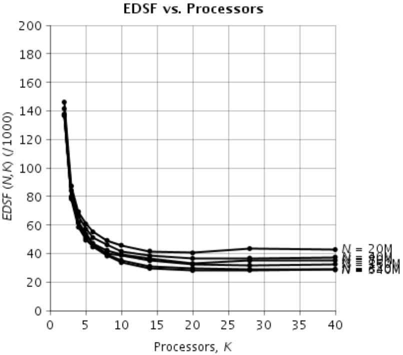
- The `tag` defaults to `0`.

MandelbrotSetClu2.java

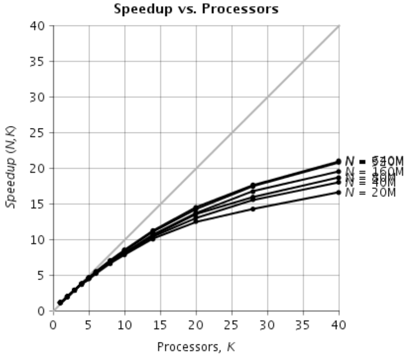
code/MandelbrotSetClu2.java

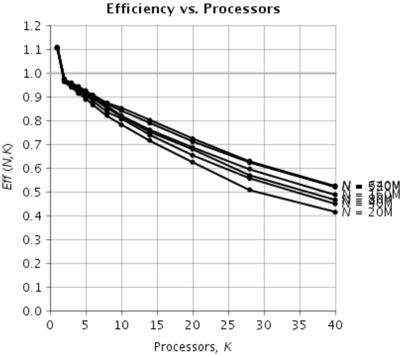
`MandelbrotSetClu2` Running Time and EDSF





MandelbrotSetClu2 Speedup and Efficiency





`MandelbrotSetClu` Results

Classic Amdahl's Law behavior.

- Speedups approaching a limit
- Efficiencies continually decreasing as K increases
- Constant sequential fraction

In fact, a very large sequential fraction.

Why does `MandelbrotSetClu` have a larger sequential fraction?

`MandelbrotSetClu` Results

Classic Amdahl's Law behavior.

- Speedups approaching a limit
- Efficiencies continually decreasing as K increases
- Constant sequential fraction

In fact, a very large sequential fraction.

Why does `MandelbrotSetClu` have a larger sequential fraction?

- initialization before master and worker processing starts
- allocation of pixel data matrix or slice at the beginning of processing
- generation of PJG image file at the end of processing

`MandelbrotSetClu` Results

Classic Amdahl's Law behavior.

- Speedups approaching a limit
- Efficiencies continually decreasing as K increases
- Constant sequential fraction

In fact, a very large sequential fraction.

Why does `MandelbrotSetClu` have a larger sequential fraction?

- initialization before master and worker processing starts
- allocation of pixel data matrix or slice at the beginning of processing
- generation of PJG image file at the end of processing
- message passing!
  - While the worker is sending its slice message and its pixel data message to the master and is waiting to receive its next slice message from the master, the worker is not computing any pixels.

Measuring Communication Overhead

How long does it take to send a message from one processor to another?
Many factors:

- network hardware
- network topology
- network protocol
- communication layer software

Measuring Communication Overhead

For a given network (hardware, topology, protocol), two principal components:

- latency (L; seconds)
- bandwith (B; bits per second)

Measuring Communication Overhead

For a given network (hardware, topology, protocol), two principal components:

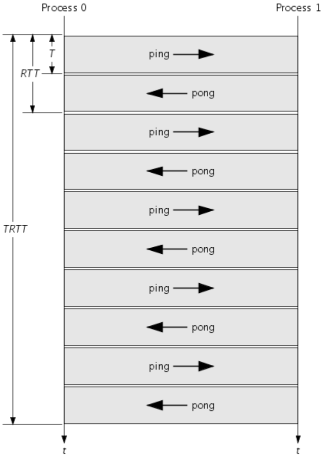
- latency ($\boldsymbol{L}$; seconds)
- bandwith ($\boldsymbol{B}$; bits per second)

$$\boldsymbol{T(b) = L + \frac{1}{B} b}$$

Would like to *experimentally* determine $\boldsymbol{L}$ and $\boldsymbol{B}$,
without using cumbersome low-level network analyzers or packet sniffers.

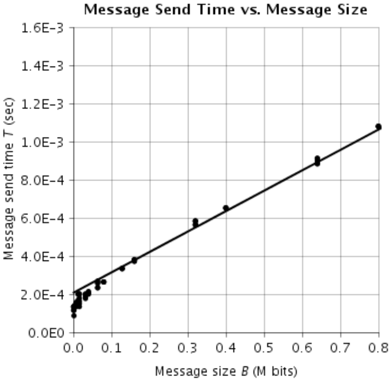
Measuring Communication Overhead

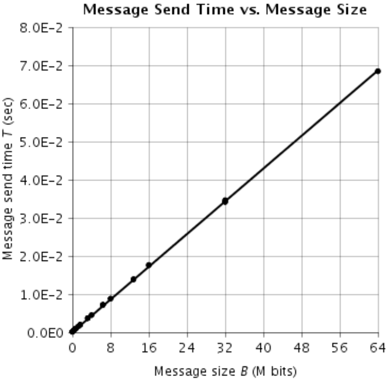
Measuring message time with *ping-pong* messages:



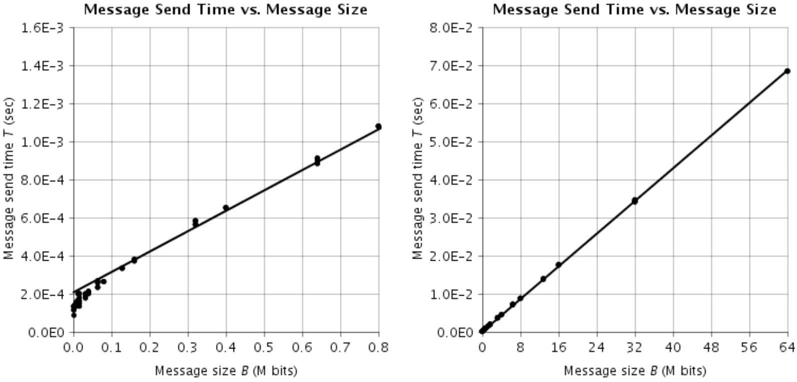
code/TimeSendByte.java

Message Send-Time Model





Message Send-Time Model



$$T(b) = 2.08 \times 10^{-4} + 1.07 \times 10^{-9} b$$

Message Send-Time Model

$$T(b) = 2.08\times10^{-4} + 1.07\times10^{-9}b$$

- latency

$$L = 2.08\times10^{-4} \text{ sec}$$

- bandwith

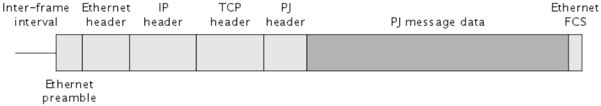
$$\frac{1}{B} = 1.07\times10^{-9} \text{ sec/bit}$$
$$B = 9.35\times10^{8} \text{ bit/sec} = 0.935 \text{ Gbps}$$

The Ethernet's advertised bandwith is 1 Gbps.
Why is the actual bandwith less than 1 Gbps?

Message Send-Time Model

Why is the actual bandwith less than **1** Gbps?

- Protocol overhead:



  - **91** bytes of protocol overhead
  - **0.941** Gbps for actual data

- Other overhead:
  - Copying data from source buffer to outgoing TCP buffer
  - Any time the sending TCP layer needs to wait to receive an ack for a TCP segment from the receiving TCP layer
  - Any time the sending TCP layer has to stop sending data temporarily because the receiving TCP layer's incoming buffer is full (*flow control*)
  - Copying data from incoming TCP buffer to destination buffer

Message Send-Time Model: Lessons

The program must have much more computation than communication.

- Communication contributes to sequential fraction.
- Best if computation time is asymptotically greater than communication time.

A few large messages are better than many small messages.

- Latency typically several orders of magnitude greater than (inverse) bandwith.

Design with Reduced Message Passing

In `MandelbrotSetClu2.java`, the majority of the message-passing time is due to the messages conveying pixel data from the workers to the master.

Design with Reduced Message Passing

In `MandelbrotSetClu2.java`, the majority of the message-passing time is due to the messages conveying pixel data from the workers to the master.

Alternative: each worker writes to a separate image file.

- Each worker writes its row slices directly to its own image file
- Eliminates a significant portion of message-passing time
- Parallelizes the image-writing time

PJG image file format supports images
with multiple chunks scattered among separate files.

Design with Reduced Message Passing

In `MandelbrotSetClu2.java`, the majority of the message-passing time is due to the messages conveying pixel data from the workers to the master.

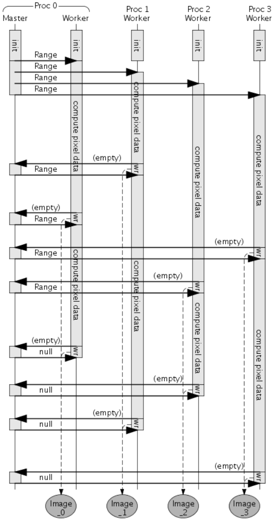
Alternative: each worker writes to a separate image file.

- Each worker writes its row slices directly to its own image file
- Eliminates a significant portion of message-passing time
- Parallelizes the image-writing time

PJG image file format supports images
with multiple chunks scattered among separate files.

But, honestly, this seems like cheating.
Someone, at some time, needs to bring all the data together
in the same place to "see" the result.
We're just not counting that time in our program measurement.

MandelbrotSetClu3.java



MandelbrotSetClu3.java

code/MandelbrotSetClu3.java

Improved scalability:

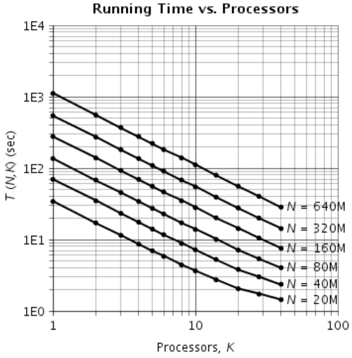
MandelbrotSetClu2.java:

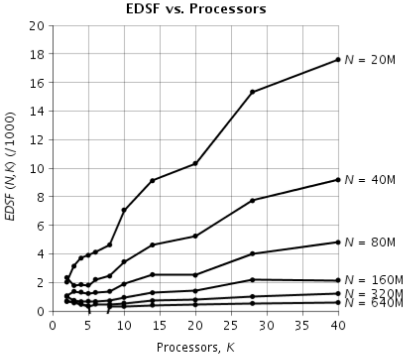
- Computation: $O(n^2)$ (ignoring balance)
- Communication: $O(n^2)$
- Comp-to-Comm: $O(1)$

MandelbrotSetClu3.java:

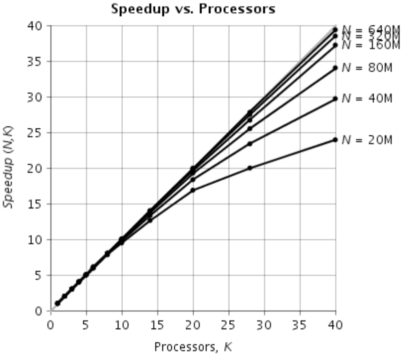
- Computation: $O(n^2)$ (ignoring balance)
- Communication: $O(n)$
- Comp-to-Comm: $O(n)$

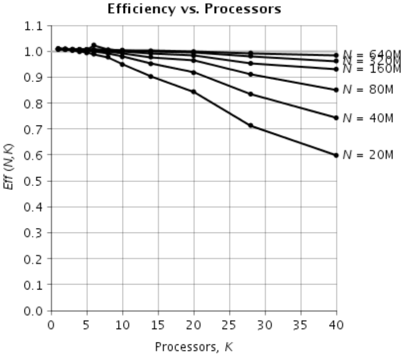
MandelbrotSetClu2 Running Time and EDSF





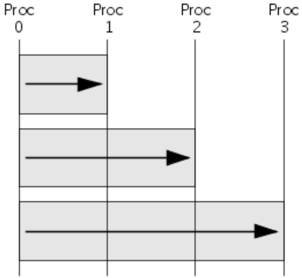
MandelbrotSetClu3 Speedup and Efficiency



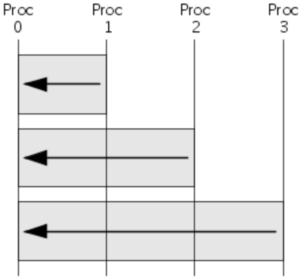


Message Scatter- and Gather-Time Models

PJ implementation of collective communication
is sequence of point-to-point communications
(but optimized self-to-self communication):



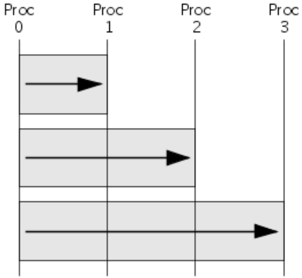
scatter



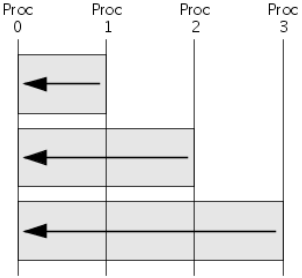
gather

Message Scatter- and Gather-Time Models

PJ implementation of collective communication
is sequence of point-to-point communications
(but optimized self-to-self communication):



scatter



gather

$$T(b, K) = (2.08\times10^{-4} + 1.07\times10^{-9} b) \cdot (K - 1)$$

Message Scatter- and Gather-Time Models

PJ implementation of collective communication
is sequence of point-to-point communications
(but optimized self-to-self communication).

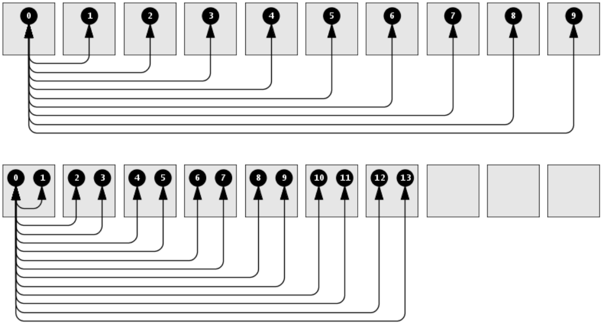
- Targeted primarily for commodity clusters,
  in which each backend node has one Ethernet network interface.

Any collective communication operation's message-time model
is implementation dependent.

Futhermore, any (high-performance) implementation is network dependent.

Inter- and Intra-Node Message Passing

On a cluster with SMP backend nodes,
distinguish between processes on same node and on different nodes:



Inter-node messages

- messages between different processes on the same node
- do not travel over the backend network
- go from source process to operating system to destination process

Inter- and Intra-Node Message Passing

Inter-node message send-time model

$$T(b) = 2.08\times10^{-4} + 1.07\times10^{-9} b$$

$$L = 2.08\times10^{-4} \text{ sec}$$
$$B = 0.935 \text{ Gbps}$$

Intra-node message send-time model

$$T(b) = 7.89\times10^{-5} + 2.26\times10^{-10} b$$

$$L = 7.89\times10^{-5} \text{ sec}$$
$$B = 4.425 \text{ Gbps}$$

Intra-node communication has lower latency and higher bandwith.