Advanced Message Passing
ASD Distributed Memory HPC Workshop

Computer Systems Group

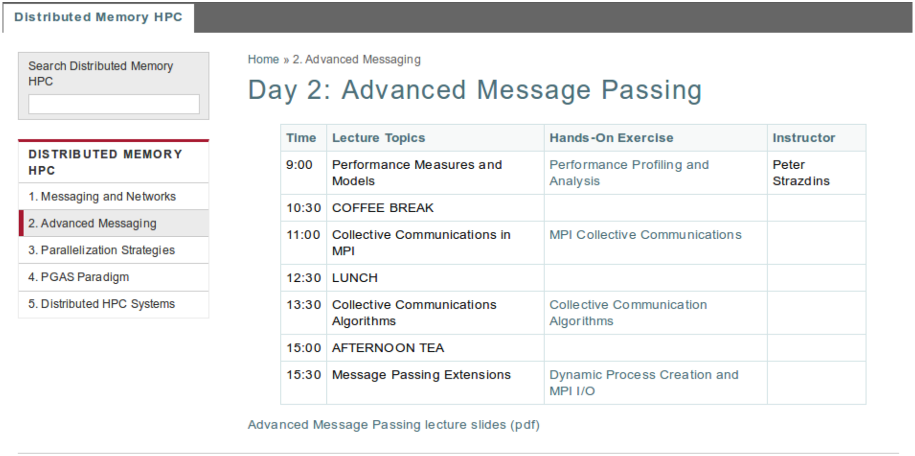
Research School of Computer Science
Australian National University
Canberra, Australia

October 31, 2017

Australian National University

ANU College of
Engineering & Computer Science

Day 2 – Schedule



Distributed Memory HPC

Search Distributed Memory HPC

Home » 2. Advanced Messaging

Day 2: Advanced Message Passing

Time	Lecture Topics	Hands-On Exercise	Instructor
9:00	Performance Measures and Models	Performance Profiling and Analysis	Peter Strazdins
10:30	COFFEE BREAK		
11:00	Collective Communications in MPI	MPI Collective Communications	
12:30	LUNCH		
13:30	Collective Communications Algorithms	Collective Communication Algorithms	
15:00	AFTERNOON TEA		
15:30	Message Passing Extensions	Dynamic Process Creation and MPI I/O	

Advanced Message Passing lecture slides (pdf)

Outline

Overview: Performance Measures and Models

- granularity of parallel programs
- parallel speedup and overhead
- Amdahls Law
- efficiency and cost
- example: adding n numbers
- scalability and strong/weak scaling
- measuring time

Ref: Grama et al. sect 3.1, ch 5; Lin & Synder

Granularity

MIMD divides computation into multiple tasks or processes that execute in parallel

- **granularity**: size of the tasks
  - **coarse grain**: large tasks/lots of instructions
  - **fine grain**: small tasks/few instructions
- granularity metric: $\frac{t_{\text{compute}}}{t_{\text{communication}}}$
  Would the startup part of communication time be better?
- granularity may depend on numbers of processors (why?)
  Case study: parallel LU factorization
- *aim: to increase granularity (why?)*

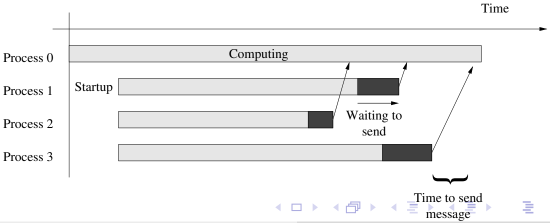
Speedup

Australian National University

- the relative performance between single and multiprocessor systems
  $S(n) = \frac{\text{execution time on single processor}}{\text{execution time using } p \text{ processors}} = \frac{t_{\text{seq}}}{t_{\text{par}}}$
- (should we use walltime or CPU time?)
- t_{seq} should be for the *fastest* known sequential algorithm
  - best parallel algorithm may be different
- may also consider speedup in terms of operation count
  $S_{\text{op}}(p) = \frac{\text{operation count rate with } p \text{ processors}}{\text{operation count rate on single processor}}$
- **linear speedup**: maximum possible speedup is n on n processors, i.e. assuming no *overhead*, etc $S(p) = \frac{t_{\text{seq}}}{t_{\text{seq}}/p} = p$
- **super-linear speedup**: when $S(p) > p$
  - may imply a sub-optimal sequential algorithm : go back and re-implement parallel algorithm on 1 processor!
  - may arise from unique features of architecture that favour parallel computation – *suggestions?*

Parallel Overhead

- factors that limit parallel scalability:
  - periods when not all processors perform useful work, including times when just one processor is active on sequential parts of the code
  - load imbalance
  - extra computations not in the sequential code, e.g. re-computation of intermediates locally (may be quicker than send from another processor)
  - communication times
- Jumpshot and VAMPIR are tools that give graphical display of parallel computation. See also details on profiling an MPI application on Raijin

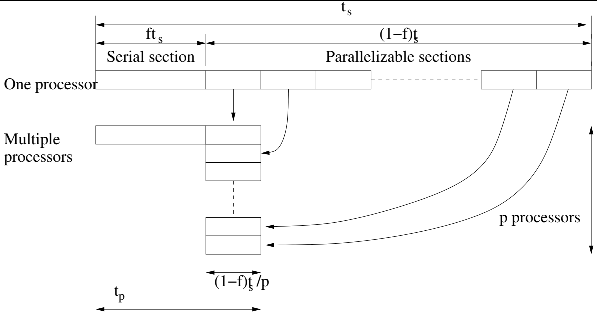


- timeline visualization

Amdahl's Law #1

Assume some part cannot be divided (f), while rest is perfectly divided (no overhead):

$$t_{\text{par}} = f t_{\text{seq}} + (1-f) t_{\text{seq}}/p$$

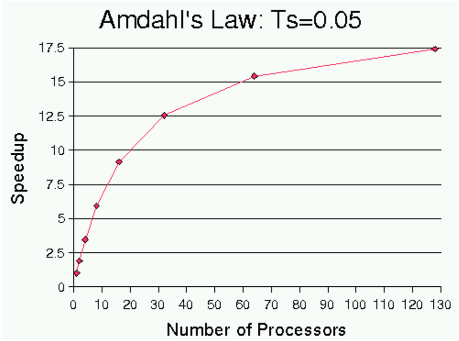


$$S(p) = \frac{t_{\text{seq}}}{f t_{\text{seq}} + (1-f) t_{\text{seq}}/p} = \frac{p}{1+(p-1)f}$$

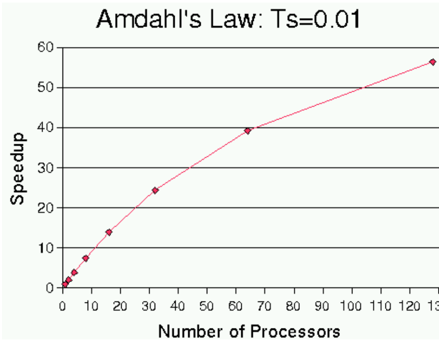
$$\underset{p\to\infty}{S(p)} = 1/f$$

Amdahl's Law #2: Speedup Curves

$f = 0.05$



$f = 0.01$



"Better to have two strong oxen pulling your plough across the country than a thousand chickens. Chickens are OK, but we can't make them work together yet" (...or can we?)

Efficiency and Cost

- **efficiency**: how well are you using the processors

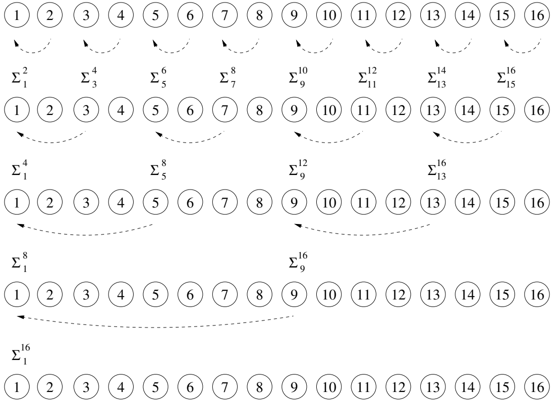
$$\begin{aligned} E &= \frac{t_{\text{seq}}}{t_{\text{par}}}/p \\ &= \frac{S(p)}{p} \times 100\% \end{aligned}$$

- **cost**: product of the parallel execution time and the total number of processors used

$$t_{\text{par}} \times p = \frac{t_{\text{seq}} p}{S(p)} = \frac{t_{\text{seq}}}{E}$$

- **cost optimal**: if the cost of solving a problem on a parallel computer has the same asymptotic growth as a function of the input size as the fastest known sequential algorithm on a single processor

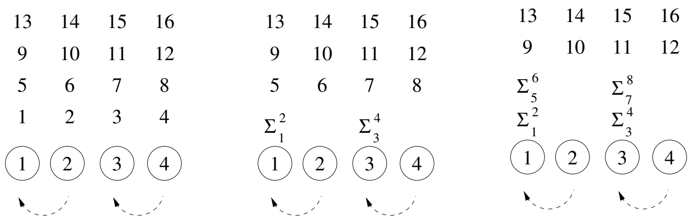
Adding *n* numbers on *n* processors



Σ_1^2 Σ_3^4 Σ_5^6 Σ_7^8 Σ_9^{10} Σ_{11}^{12} Σ_{13}^{14} Σ_{15}^{16}
Σ_1^4 Σ_5^8 Σ_9^{12} Σ_{13}^{16}
Σ_1^8 Σ_9^{16}
Σ_1^{16}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

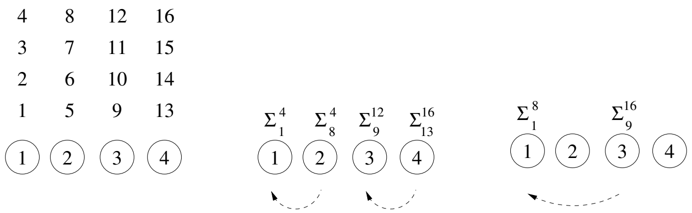
- speedup over sequential is $O(\frac{n}{\lg n})$
- cost is $O(n \lg n)$, so not cost optimal!

Adding *n* numbers on *p* processors #1



- algorithm takes $O(n/p \lg p)$ to communicate numbers, then $O(n/p)$ to add partial sums. Thus total execution time is $O(n/p \lg p)$
- cost is $O(n \lg p)$ which is not cost optimal - either!!

Adding n numbers on p processors #2



4 8 12 16
3 7 11 15
2 6 10 14
1 5 9 13
1 2 3 4
Σ_1^4 Σ_8^4 Σ_9^{12} Σ_{13}^{16}
1 2 3 4
Σ_1^8 Σ_9^{16}
1 2 3 4

- algorithm takes $O(n/p + \lg p)$
- cost is $O(n + p \lg p)$ so if $n = \Omega(p \lg p)$ (i.e. $n \geq p \lg p$), cost is $O(n)$, which is cost-optimal

Scalability

Imprecise measure:

- **hardware scalability**: does increasing the size of the basic hardware give increased performance?
  - consider ring, crossbar, hypercube topologies and what changes as we add processors
- **algorithmic scalability**: can the basic algorithm accommodate more processors?
- **combined**: an increased problem size can be accommodated on increased processors
- consider effect of doubling computation size:
  - for two $N \times N$ matrices, doubling the value of N increases the cost of addition by a factor of 4, but the cost of multiplication by a factor of 8.

Gustafson's Law: Strong/Weak Scaling

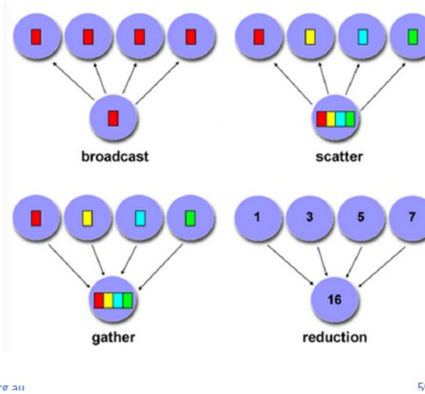
- recall we assume a serial computation can be split to serial and parallel parts: $t_{\text{seq}} = ft_{\text{seq}} + (1-f)t_{\text{seq}}$ and parallel time is given by $t_{\text{par}} = ft_{\text{seq}} + (1-f)t_{\text{seq}}/p$ and the speedup is $S(p) = t_{\text{seq}}/t_{\text{par}}$
- **Amdahl's Law**: constant problem size scaling (**strong scaling**) $S(p) = \frac{p}{1+(p-1)f}$
- **Gustafson's Law**: time constrained scaling (i.e. problem size is dependent on processor count, **weak scaling**)
  - *assumes* parallel execution time t_{par} is fixed (for simplicity, assume $t_{\text{par}} = 1$)
    and the sequential time component ft_{seq} is a constant
  - yielding a speedup of:
    $S(p) = p + (1-p)ft_{\text{seq}}$
  - speedup a line of negative slope rather the rapid reduction observed previously
  - 5% serial on 20 processors implies $S(p) = 19.05$ but under Amdahl's Law, $S_(p) = 10.26$

Hands-on Exercise: Performance Profiling

Outline

Collective Communications: Basic Ideas

- **synchronization**: barrier to inhibit further execution until all processes have participated
  - e.g. use simple pingpong between two processes
- **broadcast**: send same message to many processes
  - must define the source of the message
- **scatter**: 1 process sends unique data to every other in group
- **gather**: reverse of above



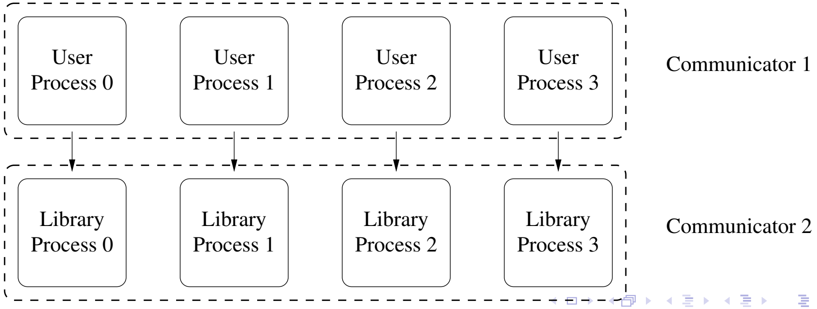
(courtesy LLNL)

- **reduction**: gather and combined with arithmetic/logical operation
  - result can go to just one process, or goes to all processes

All of these can be constructed from simple sends and receives, and all require the group of participating processes to be defined.

MPI Communicators

- a **communicator** is a group that MPI processes can join
- `MPI_COMM_WORLD` is the communicator defined in `MPI_Init()`, and contains all processes created at that point
- these can be used to specify the group of processes in a collective communication
- they can also prevent conflict between messages, e.g. that are internal to a library and those used by the application program



Collective Operations and Communications

- by definition, a **collective operation** in MPI requires *all* processes in the specified communicator to participate
    - this is most often for a collective communication (but can also be for communicator creation / destruction, I/O etc)
    - usually this provides a degree of synchronization as well
    - if any process fails to participate in the collective, you will get deadlock!
- MPI **collective communications** provide convenient ways of expressing widely-used communication patterns
- they are normally also highly optimized, with algorithms optimized on
    - varying numbers of process,
    - small or large message sizes
    - various communication transports (e.g. shared memory, TCP/IP, Infiniband)

It is worth learning them!

Simple MPI Collective Communications

- `MPI_Barrier(MPI_Comm comm)`: barrier synchronization for all processes (in `comm`)
- `MPI_Bcast(void *buf, int count, MPI_Datatype dt, int root, MPI_Comm comm)`: broadcast message from process `root` to all others
- `MPI_Reduce(const void *sbuf, void *rbuf, int count, MPI_Datatype dt, MPI_Op op, int root, MPI_Comm comm)`: apply reduction `op` element-wise on send buffer, storing result in receive buffer on process `root`
  `op` may be `MPI_MAX`, `MPI_SUM` or any other well-known associative operator on numeric types; or a user-defined operation
- `MPI_Allreduce(const void *sbuf, void *rbuf, int count, MPI_Datatype dt, MPI_Op op, MPI_Comm comm)`: similar, except result is stored on all processes.
  Equivalent to `MPI_Reduce(sbuf, rbuf, count, dt, op, 0, comm)`, followed by `MPI_Bcast(rbuf, count, dt, op, 0, comm)`.

Simple MPI Collectives Example

```c
#define NP 4
int i, np, rank; MPI_COMM comm;
static int sbuf[NP], rbuf[NP], arbuf[NP];
MPI_Init(&argc, &argv); comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &rank); MPI_Comm_size(comm, &np);
assert(np == NP); //i.e. invoked with mpirun -np NP ...
if (rank == 0)
  for (i=0; i < np; i++)
    sbuf[i] = i + 1;
MPI_Bcast(sbuf, np, MPI_INT, 0, comm);
MPI_Reduce(sbuf, rbuf, np, MPI_INT, MPI_SUM, 0, comm);
MPI_Allreduce(sbuf, arbuf, np, MPI_INT, MPI_SUM, comm);
MPI_Barrier(comm); // has no real effect here
```

sbuf:

0:	1	2	3	4
1:	1	2	3	4
2:	1	2	3	4
3:	1	2	3	4

rbuf:

0:	4	8	12	16
1:	0	0	0	0
2:	0	0	0	0
3:	0	0	0	0

arbuf:

0:	4	8	12	16
1:	4	8	12	16
2:	4	8	12	16
3:	4	8	12	16

MPI Scatter and Gather

Australian National University

```
int MPI_Scatter(const void *sbuf, int scount, MPI_Datatype sdt,
                void *rbuf, int rcount, MPI_Datatype rdt,
                int root, MPI_Comm comm);
int MPI_Gather(const void *sbuf, int scount, MPI_Datatype sdt,
               void *rbuf, int rcount, MPI_Datatype rdt,
               int root, MPI_Comm comm)
```

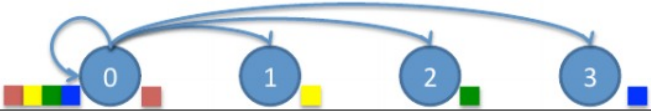
The **scatter** is equivalent to (`extent(dt)` is # bytes in `dt`):

```
assert (extent(sdt)*scount==extent(rdt)*rcount);
if (rank == root) //rank is process id, np is #processes in comm
  for (i=0; i < np, i++) //sbuf holds np*scount elements of sdt
    MPI_Send(sbuf+i*scount*extent(sdt), scount, sdt, i, tag, comm);
 MPI_Recv(rbuf, rcount, rdt, root, comm, ...);
```

and its inverse, **gather**, is equivalent to:

```
MPI_Send(sbuf, scount, sdt, root, comm);
if (rank == root)
  for (i=0; i < np, i++) //rbuf holds np*rcount elements of rdt
    MPI_Recv(rbuf+i*rcount*extent(rdt), rcount, rdt, i, tag, comm,
        ...);
```

MPI Collective Communication Example



```c
#define NP 4
int np, rank; MPI_COMM comm;
int sbuf[NP*NP] = {1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16};
static int rbuf[NP], gbuf[NP];
MPI_Init(&argc, &argv); comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &rank); MPI_Comm_size(comm, &np);
assert(np == NP); //i.e. invoked with mpirun -np NP ...
// both send count and receive count equal np
MPI_Scatter(sbuf, 1, MPI_INT, rbuf, 1, MPI_INT, 0, comm);
MPI_Gather(rbuf, 1, MPI_INT, gbuf, 1, MPI_INT, 3, comm);
```

rbuf:

0:	1	2	3	4
1:	5	6	7	8
2:	9	10	11	12
3:	13	14	15	16

gbuf:

0:	0	0	0	0
1:	0	0	0	0
2:	0	0	0	0
3:	1	5	9	13

MPI All-to-all Collective Communications

Australian National University

- MPI_Allgather(sbuf, scount, sdt, rbuf, rcount, rdt, comm) is like a gather, but all processes have the combined result. Equivalent to:

```
MPI_Gather(sbuf, scount, sdt, rbuf, rcount, rdt, 0, comm);
MPI_Bcast(rbuf, np*rcount, rdt, 0, comm);
```

- MPI_Alltoall(sbuf, scount, sdt, rbuf, rcount, rdt, comm) allows each process to send a different message to all others. Equivalent to:

```
for (i=0; i < np, i++) //sbuf holds np*scount elements of std
  MPI_Send(sbuf+i*scount*extent(sdt), scount, sdt, i, tag,
      comm);
for (i=0; i < np, i++) //rbuf holds np*rcount elements of rtd
  MPI_Recv(rbuf+i*rcount*extent(rdt), rcount, rdt, i, tag,
      comm, ...);
```

MPI All-to-all Collectives Example

```c
#define NP 4
int np, rank; MPI_COMM comm;
int i, sbuf[NP], rbuf[NP], gbuf[NP*NP];
MPI_Init(&argc, &argv); comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &rank); MPI_Comm_size(comm, &np);
assert(np == NP); //i.e. invoked with mpirun -np NP ...
for (i=0; i < np; i++)
  sbuf[i] = rank*np + i;
MPI_Alltoall(sbuf, 1, MPI_INT, 1, np, MPI_INT, comm);
MPI_Allgather(rbuf, np, MPI_INT, gbuf, np, MPI_INT, comm);
```

sbuf:

0:	0	1	2	3
1:	4	5	6	7
2:	8	9	10	11
3:	12	13	14	15

rbuf:

0:	0	4	8	12
1:	1	5	9	13
2:	2	6	10	14
3:	3	7	11	15

gbuf (all procs.):

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Interpreting `sbuf` and `rbuf` across all processes as matrices, the **all-2-all** has performed a **transposition**. Basis of an n-way || FFT (`n = nl*np`):

- local `nl`-way FFT; transpose; do `nl/np` local `np`-way FFTs

MPI All-to-All Collectives (II): Reduce-Scatter

`MPI_Reduce_scatter(sbuf, rbuf, rcounts, MPI_Datatype dt, MPI_Op op, MPI_Comm comm)`: performs a reduction on `sbuf` (of size `n`$= \sum_{i=0}^{\mathtt{np}-1}$ `rcounts[i]`), and sends ith segment (of size `rcounts[i]`) to process i, storing result in `rbuf`

```
#define NP 4
int np, rank; MPI_COMM comm;
int i, sbuf[NP], rbuf[1], rcounts[] = {1,1,1,1};
MPI_Init(&argc, &argv); comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &rank); MPI_Comm_size(comm, &np);
assert(np == NP); // assert np == sum(rcounts)
for (i=0; i < np; i++)
  sbuf[i] = rank*np + i;
MPI_Reduce_scatter(sbuf, rbuf, rcounts, MPI_INT, MPI_SUM, comm);
```

sbuf:

0:	0	1	2	3
1:	4	5	6	7
2:	8	9	10	11
3:	12	13	14	15

rbuf:

0:	24
1:	28
2:	32
3:	36

Varying Message Sizes in Collectives

- so far, with gather, scatter, all-to-all, etc, the messages are all of equal size
- 'vector' versions of these collectives allow us to specify differing message sizes to and/or from each process
- e.g.

  ```
  MPI_Alltoallv(void *sbuf, int scounts[], int sdispls[], MPI_Datatype sdt,
  void *rbuf, int rcounts[], int rdispls[], MPI_Datatype rdt, MPI_Comm comm)
  ```

  is equivalent to:

```
for (i=0; i < np, i++) //sbuf holds np*scount elements of std
  MPI_Send(sbuf+sdispls[i]*extent(sdt), scount[i], sdt, i, tag
        , comm);
for (i=0; i < np, i++) //sbuf holds np*scount elements of std
  MPI_Recv(rbuf+rdisps[i]*extent(rdt), rcount[i], rdt, i, tag,
         comm, ...);
```

Question: why does MPI provide us with collectives that are easily expressed as combinations of other collectives?

Hands-on Exercise: MPI Collectives

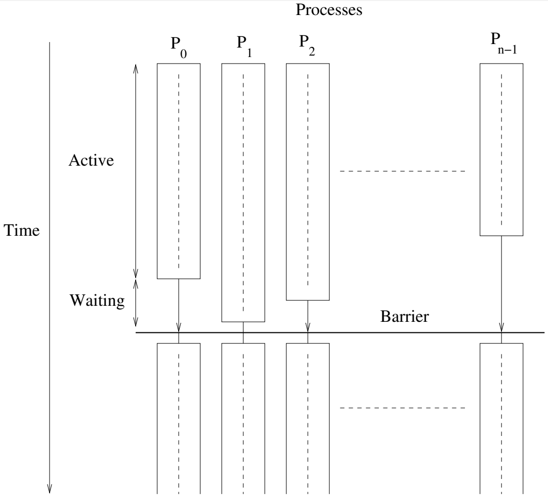
Outline

Barriers

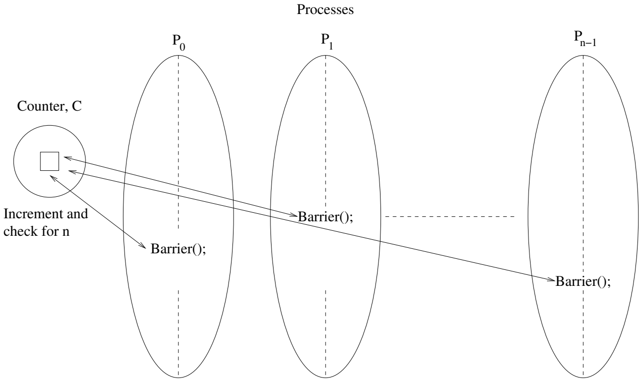
Australian National University

- recall that a **barrier** is a point at which all processes must wait until all other processes have reached that point
  `MPI_Barrier(MPI_Comm comm);`
- **mutual exclusion**: a barrier that prevents other processes from entering the following region if another process is already in that region
  - common in shared memory parallel programs
  - necessary for some MPI-2 operations
- both are possible sources of overhead

Barrier - Schematic



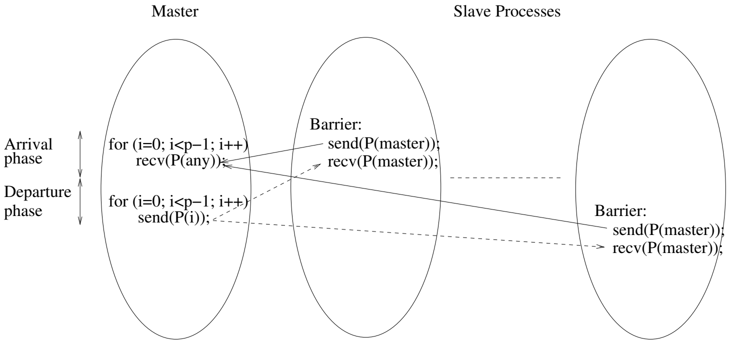
Counter-based or Linear Barriers



- one process counts the arrival of the other processes
  - when all processes have arrived, they are each sent a release message

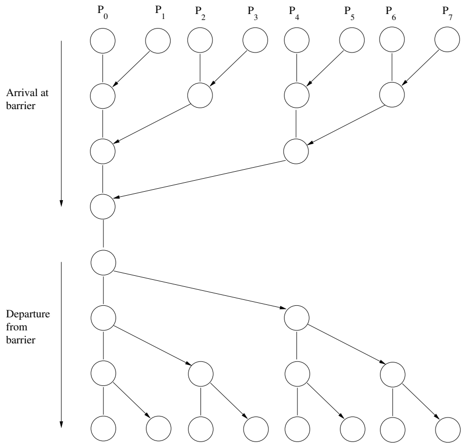
Implementation

- *arrival phase:* process sends message to central counter
- *departure phase:* process receives message from central counter



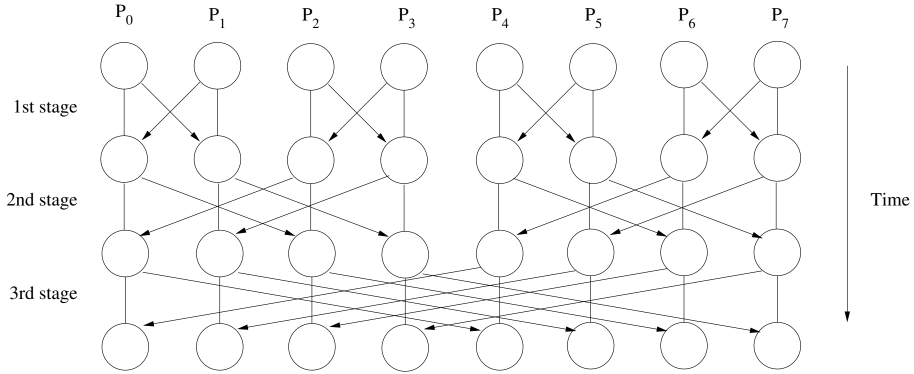
- implementations must handle possible time delays
- the central process is the bottleneck, cost is $2t_s(p-1) = O(p)$,

Tree-Based Barriers



- note: broadcast does not ensure synchronization
- cost $2 \lg p \cdot t_s$ or $O(\lg p)$

Butterfly Barrier (Butterfly/Omega Network)



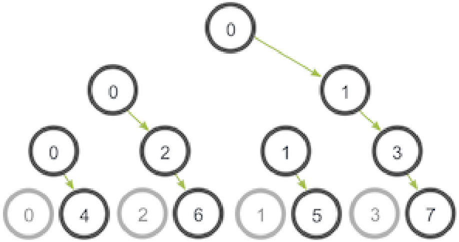
- cost is $2 \lg p \cdot t_s$ or $O(\lg p)$

Broadcast

- the broadcast `bcast(buf, m, root)` can be a naively implemented as:

```
if (rank == root) {
  for (i=0; i < p; i++)
    if (i != root)
      send(buf, m, i);
} else
  recv(buf, m, root);
```

- cost is $(p-1)(t_s + mt_w) = O(p)$! Using a tree-like structure:



(courtesy mpitutorial.com)

- more efficient: overall cost is $\lg p(t_s + mt_w) = O(\lg p)$
- this is also the maximal per-process cost (in this case, process 0)

Broadcast: Tree and Pipelined

- assuming `root` is process 0, and `pceil`$= 2^{\lceil \lg p \rceil}$, the tree broadcast can be implemented as:

```
for (d = pceil/2; d >= 1; d/=2) {
  if (rank % (2*d) == 0  &&  rank + d < p)
    send(buf, m, rank + d);
  if (rank % (2*d) == d)
    recv(buf, m, rank - d);
}
```

- the **pipelined broadcast**: $\boxed{0} \longrightarrow \boxed{1} \longrightarrow \boxed{2} \longrightarrow \boxed{3}$

```
if (rank != 0)
  recv(buf, m, rank-1);
if (rank != p-1)
  send(buf, m, rank+1);
```

- total cost is $(p-1)(t_s + mt_w) = O(p)$
- but, max. cost per process is $t_s + mt_w$
- cost of p consecutive broadcasts is $(2p-1)(t_s + mt_w)$. For tree?

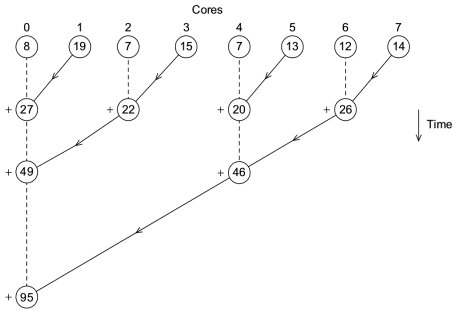
Scatter

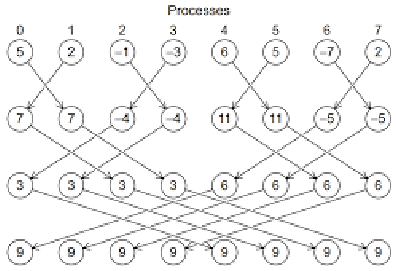
- recall that a **scatter** can be implemented as $p-1$ sends from the root process
- total cost is $(p-1)(t_s + mt_w)$, where m is the send count
- however, applying the tree communication pattern, and sending halved amounts of data at each stage

```
for (d = pceil/2; d >= 1; d/=2) {
  if (rank % (2*d) == 0  &&  rank + d < p)
    send(&buf[d*m], d*m, rank + d);
  if (rank % (2*d) == d)
    recv(buf, d*m, rank - d);
} //result of scatter is in msg[0..m-1]
```

- above scheme is an example of **recursive halving**
- noting $p/2 + p/4 + \ldots + 1 = p-1$, total cost is $\lg p \cdot t_s + (p-1)mt_w$
- improvement for small m;
- also maximum 'fan-out' is reduced from $p-1$ to $\lg p$

Binary Tree-Based Reduce and All-Reduce





(courtesy CPSC425, http://cs.umw.edu)

```
for (d = 1; d < pceil; d*=2) {
  if (rank % (2*d) == d)
    send(buf, m, rank - d);
  if (rank % (2*d) == 0 && ...)
    recvAdd(buf, m, rank + d); }
```

```
assert (p == pceil);
for (d = 1; d < pceil; d*=2) {
  sd = (rank%(2*d) >= d)? -d: +d
  send(buf, m, rank+sd);
  recvAdd(buf, m, rank+sd);}
```

Cost is $\lg p(t_s + mt_w)$ in both cases. Issues?

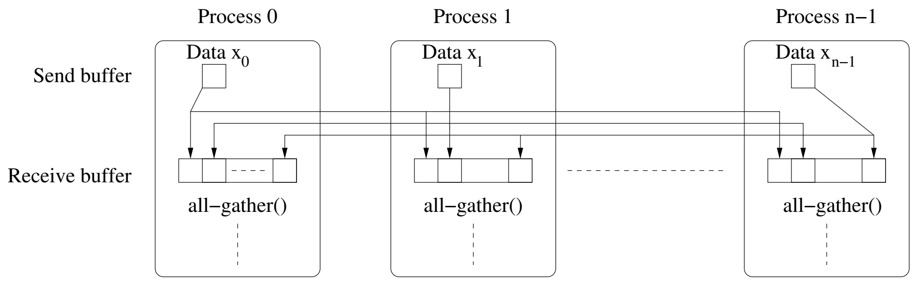
Gather

- recall that a **gather** can be implemented as $p-1$ receives to the root process
- applying reduce's tree communication pattern, and sending *doubled* amounts of data at each stage:

```
//assume the send buffer is in msg[0..m-1]
for (d = 1; d < pceil; d*=2) {
  if (rank % (2*d) == d)
    send(msg, d*m, rank - d);
  if (rank % (2*d) == 0  &&  rank + d < p)
    recv(&msg[d*m], d*m, rank + d);
}
```

- above scheme is an example of **recursive doubling**
- similarly total cost is $\lg p \cdot t_s + (p-1)mt_w$
- improvement also over max. 'fan-in', from $p-1$ to $\lg p$

All-Gather



Can be (simplistically) implemented as:

```
for (i = 0; i < p; i++)
  send(sbuf, m, /*process*/ i);
for (i = 0; i < p; i++)
  recv(&rbuf[i*m], m, /*process*/ i);
```

Neglecting the cost of a self-send, the cost is $(p-1)(t_s + mt_w)$. Further issues?

All-Gather: Recursive Doubling

- **all-reduce** pattern with **recursive doubling** gives:

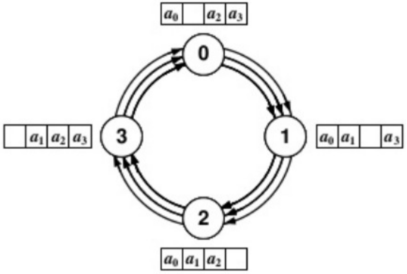
```
//assume the send data is in msg[rank*m..rank*m+m-1]
for (d = 1; d < p; d*=2) {
  sd = (rank%(2*d) >= d)? -d: +d;
  rd = (rank / d) * d;
  send(&msg[rd*m], d*m, rank + sd);
  recv(&msg[(rd+sd)*m], d*m, rank + sd);
}
```

- the cost is $\lg p \cdot t_s + (p-1)mt_w$ (good for small m)
- as with all-reduce, contention may be an issue on some networks
- how to implement for non-power-of-2 p?

All-Gather: Ring-based

- a simpler algorithm can avoid contention, and works for all p

```
//send data in msg[sk*m..]
l = (rank + 1)%p;
r = (p + rank - 1)%p;
sk = rank;
for (k=0; k < p-1; k++) {
   send(&msg[sk*m], m, r);
   sk = (p + sk - 1) % p;
   recv(&msg[sk*m], m, l);
}
```



(courtesy slideshare.net)

- can be thought of as p pipelined broadcasts from each process, in parallel
- but cost is still $(p-1)(t_s + mt_w)$
- often these patterns works over a subset of all processes, e.g. a row or column in a logical 2-D process grid,
  so the $O(p)t_s$ term is not so much of a disadvantage

Reduce-Scatter

- can use an **all-reduce** pattern, with **recursive halving**

```
for (d = p/2; d >= 1; d/=2) {
  sd = (rank%(2*d) >= d)? -d: +d;
  rd = (rank / d) * d;
  send(&msg[(rd+sd)*m], d*m, rank + sd);
  recv(buf, d*m, rank + sd);
  add(buf, &msg[rd*m], d*m);
}
// result is in msg[rd*m..rd*m+m-1]
```

- we can similarly use ring-based methods

```
sk = (p + rank - 1) % p;
for (k=0; k < p-1; k++) {
  send(&msg[sk*m], m, r);
  sk = (p + sk - 1) % p;
  recv(buf, m, l);
  add(buf, &msg[sk*m], m);
}
// result is in msg[sk*m..sk*m+m-1]
```

Hands-on Exercise: Collective Algorithms

Outline

MPI: Early History

MPI-1 (May 94)

- fixed process model
- point-point communications
- collective operations
- communicators for safe library writing
- utility routines

MPI-2 (July 97)

- dynamic process management
- **one-sided communications**
- cooperative I/O
- (other small things!) C++/Fortran 90 binding, extended collectives, etc.

Much more complicated, and much slower vendor uptake...

Dynamic Process Management

- MPI-1 had a static or fixed number of processes
  - could not add or delete processes
  - (you could have a fixed pool of processes and only use some of them, but cost of having idle processes may be large – implementation dependent)
- some applications favour dynamic spawning:
  - run-time assessment of environment
  - serial applications with parallel modules
  - scavenger applications

  Dynamic spawning also supports coupled simulations (e.g. climate models).
- *caution:* task initiation is expensive and should be used with careful thought

MPI-2 Process Management

Features:

- parents can spawn children
- existing MPI applications can connect
- formerly independent sub-applications can tear down communications and become independent again

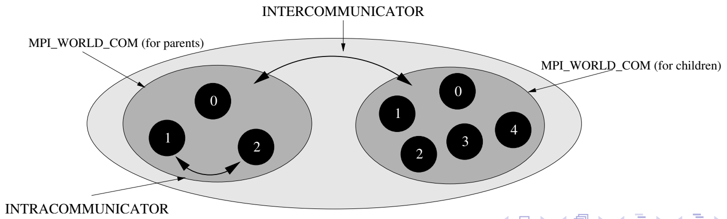
Task spawning:

```
MPI_Comm_spawn(command, argv, nprocs, info, root,
               parent_intracomm, intercomm, errcodes);
```

- this is a **collective operation** over the parent processes' communicator
- `info` parameter: details on how to start the new processes (host, architecture, work directory, path etc)
- `intercomm` and `errcodes` are returned values

Communicators

- MPI processes are identified by (group, rank) pairs
- communicators are either:
  - **intra-group**
  - **inter-group**: ranks refer to processes in the remote group
- processes in the parent and children groups **each** have their **own** `MPI_COMM_WORLD`
- `MPI_Send/Recv()` etc have a destination **and** an inter/intra communicator
- it is possible to merge processes or free parents from children (!) `MPI_Intercomm_merge()` and `MPI_Comm_free()`



MPI Dynamic Process Hello World

- parent code:

```
MPI_Comm childInterComm; int nChilds=4; char msgBuf[64];
MPI_Comm_spawn("helloChild", MPI_ARGV_NULL, nChilds, ...,
               0, MPI_COMM_WORLD, &childInterComm, ...);
MPI_Comm_remote_size(childInterComm, &nChilds);
strncpy(msgBuf, argv[0], 64);
root = (rank == 0)? MPI_ROOT: MPI_PROC_NULL;
MPI_Bcast(msgBuf, 64, MPI_CHAR, root, childInterComm);
printf("Hello from proc %d of %d, parent of %d %s childs\n",
       rank, nprocs, nChilds, CHILD_NAME);
```

- child code (`helloChild.c`):

```
MPI_Comm parentInterComm; int nParents; char msgBuf[64];
MPI_Comm_get_parent(&parentInterComm);
MPI_Comm_remote_size(parentInterComm, &nParents);
MPI_Bcast(msgBuf, 64, MPI_CHAR, 0, parentInterComm);
printf("Hello from proc %d of %d, child of %d %s parents\n",
       rank, nprocs, nParents, msgBuf);
```

- note the specification of the root of the broadcast in each case

One-Sided Communications

In traditional message passing:

- one process sends, the other receives (cooperative data transfer)
    - there is an implicit synchronization – although it may be delayed by asynchronous message passing

One-sided communication:

- paradigm was strongly driven by Cray SHMEM library (T3D/T3E systems), although the MPI-2 model is a bit unusual!
- one process specifies all communication parameters
    - data transfer and synchronization are separate
- typical operations are put, get, accumulate:

```
MPI_Put(origin_addr, origin_count, origin_datatype,
        target_rank, target_disp, target_count,
        target_datatype, win);
```

MPI-2 Remote Memory Access (RMA)

- processes assign a portion (or **window**) of their address space that they explicitly expose to RMA operations

```
MPI_Win win;
MPI_Win_allocate(size, disp_unit, info, comm, baseptr, &win);
```

- two types of targets:
  - **active target RMA**: requires all processors that created the window to call `MPI_Win_fence()` before *any* RMA operation is guaranteed complete
    - communication is one-sided: no process is required to post a receive
    - communication is cooperative in that all processes must synchronize
  - **passive target RMA**: the only requirement is that the originating process places `MPI_Win_lock/unlock()` before & after the data transfer
    - transfer is guaranteed to have completed on return from `MPI_Win_unlock()`
    - this is known as (Cray SHMEM) **one-sided communication**
- potential for one process to get and a second process to put to the same location on a 3rd process – this will give arbitrary results
  - we can avoid this by using locks or mutexes

MPI-2 File Operations

Positioning:

- explicit offset
- shared pointer / individual pointers

Synchronization:

- blocking / non-blocking (asynchronous)

Coordination:

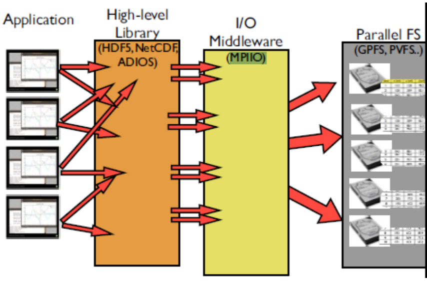
- collective / non-collective

Filetypes:

- a **filetype** is a datatype made up of elementary types (etypes), e.g. `MPI_INT`
- allows us to specify non-contiguous accesses
- files can be **tiled**, such that process i writes to block $i, i+p, i+2p, \ldots$ block of the file (p is the number of processes)

MPI-IO Usage

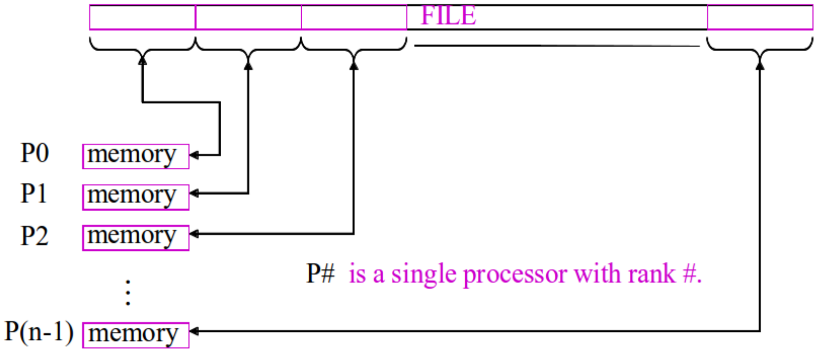
- every process writes its own data to a separate file
  - this is what we have now, i.e. just using language-specific I/O
- processes can append data to a common file, e.g. a log file
  - no tiling, non-collective operations, common shared file pointer
- processes can cooperatively write a large matrix to a file
  - create a **filetype** to tile the file
  - use individual pointers
  - use collective operations to allow data shuffling
- a parallel file system can be used, but appears appears like a normal file system
  - it employs multiple I/O servers for high sustained throughput


Application
High-level Library (HDF5, NetCDF, ADIOS)
I/O Middleware (MPIIO)
Parallel FS (GPFS, PVFS..)

We will concentrate on cooperative file operations with individual pointers.

Simple MPI-IO

Each MPI processes reads or writes to a single block in the file.



Simple MPI I/O

Australian National University

- for each MPI processes to read/write a single block in the file, the following 3 steps are required:
- `MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)`: a **collective** over `comm`, creating both an individual and shared file pointer
  - the `info` parameter allows us to send extra hints about the file (e.g. performance tuning, special case handling)
- the subsequent read/write generally requires a positioning to occur:
  - `MPI_File_seek(fh, offset, ...); MPI_File_read(fh, ...)` (use individual file pointer)
  - `MPI_File_Read_at(fh, offset, ...);` (directly read at desired offset)
  - `MPI_File_seek_shared(fh, offset, ...); MPI_File_read_shared(fh, ...);` (use shared file pointer; note: the shared seek is a collective!)

  The read/write calls specify a buffer, count and datatype (like normal recv/send).
- `MPI_File_close(fh)`: also a collective

MPI-IO Hello World

```c
const int msize = 6;
char *helloMsg[] = {"Hello ", "World!"};
char msg[msize];
int rank;
MPI_File fh; MPI_Offset offset;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_File_open(MPI_COMM_WORLD, "hello.txt",
              MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
offset = msize * rank;
MPI_File_seek(fh, offset, MPI_SEEK_SET);
memcpy(msg, helloMsg[rank % 2], msize);
MPI_File_write(fh, msg, msize, MPI_CHAR, MPI_ANY_STATUS);
MPI_File_close(&fh);
MPI_Finalize();
```

```
$ mpirun -np 4 ./helloMPIIO
$ cat hello.txt
Hello World!Hello World!$
```

MPI-2 and Beyond

- MPI-2 added a lot of new functionality
  - uptake of new features was much much slower than for MPI-1
  - vendor-specific implementations were for long incomplete
- MPI-3 (2012, 2015): improved one-sided communications, non-blocking collectives
- portable (and open-source!) implementations are widely used MPICH (mid 90's) and OpenMPI (2004)
- issues in modern MPI implementations (ref: MPI-3 and Beyond, by William Gropp)
  - must support the major 'transports', e.g. ShMem, TCP/IP, IB
  - when p becomes large (case study: UM profiling)
    - spawning overheads
    - must each process establish p connections, allocate p message buffers?
- MPI+X, X=C/C++/Fortran, continues to be the dominant programming model for supercomputing
- future challenges: dealing with many threads, GPUs and other devices fault tolerance: User-Level Fault Mitigation MPI pilot (case study)

Summary

Topics covered today:

- performance measures and models
  speedup, overheads, Amdahl's Law, efficiency & cost-optimality, strong/weak scaling
- collective communications in MPI:
  basic ideas, API
- collective communication algorithms
  naive vs tree/recursive vs ring;
  performance (end-to-end, per process, congestion)
- message passing extensions:
  dynamic process managements, intra/inter-communicators, MPI I/O

Tomorrow - parallelization strategies

Hands-on Exercise: Dynamic Processes, MPI I/O