MPI Scatter/Gather

on-line at

http://netlib2.cs.utk.edu/utk/papers/mpi-book/node1.html

Gather:

Used to collect “rows” of an array

```
MPI_Gather(void* sendbuf,
           int sendcount,
           MPI_Datatype sendtype,
           void* recvbuf,
           int recvcount,
           MPI_Datatype recvtype,
           int root,
           MPI_Comm comm)
```

The outcome is *as if* each of the n processes in the group (including the root process) had executed a call to

MPI_Send(sendbuf, sendcount, sendtype, root, ...),

and the root had executed n calls to

MPI_Recv(recvbuf+i-recvcount-extent(recvtype), recvcount, recvtype, i ,...),

where extent(recvtype) is the type extent obtained from a call to MPI_Type_extent().

Gather Example

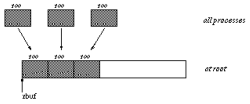


Figure 4.3
The root process gathers 100 ints from each process in the group.

Example 4.4 Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of gsize*100 ints since type matching is defined pairwise between the root and each process in the gather.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

Scatter:

Used to distribute “rows” of an array

```
MPI_Scatter(void* sendbuf,
            int sendcount,
            MPI_Datatype sendtype,
            void* recvbuf,
            int recvcount,
            MPI_Datatype recvtype,
            int root,
            MPI_Comm comm)
```

The outcome is *as if* the root executed n send operations,

MPI_Send(sendbuf+i-sendcount-extent(sendtype), sendcount, sendtype, i,..),

i = 0 to n - 1. and each process executed a receive,

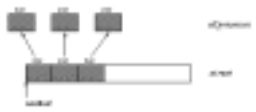
MPI_Recv(recvbuf, recvcount, recvtype, root,...).

Scatter Example

4.7.1 An Example Using MPI_SCATTER

Example 4.11 The reverse of Example 4.2, page 155. Scatter sets of 100 ints from the root to each process in the group. See Figure 4.7.

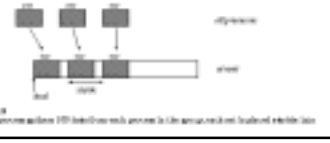
```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```



“Vector Variants”

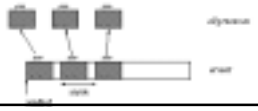
- The Vector Variants allow the gathered/scattered sub-arrays to be of different sizes, by specifying an array of lengths of the sizes. They also allow there to be “gaps” (or “stride”) by making the displacements explicit.

MPI_Gather**v**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf,
int *recvcounts, int *displs, arrays
MPI_Datatype recvtype, int root, MPI_Comm comm)



“Vector Variants”

MPI_Scatter**v**(void* sendbuf,
int *sendcounts, int *displs, arrays
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)



“All” Variants

- The all variants distribute the results of a gather to all processors in the communicator:

MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf,
int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

MPI_Allgather**v**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf,
int *recvcounts, int *displs, arrays
MPI_Datatype recvtype, MPI_Comm comm)

extremely useful

“All” Variants

Alltoall allows everything to be scattered and gathered in one call.

Contents of distinct send buffers are sent to all receive buffers

MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)

MPI_Alltoall**v**(void* sendbuf, int *sendcounts,
int *sdispls, MPI_Datatype sendtype,
void* recvbuf, int *recvcounts, int *rdispls,
MPI_Datatype recvtype, MPI_Comm comm)

PVM vs. MPI

Jack Dongarra

Distinguished Professor, University of Tennessee
Distinguished Scientist, Oak Ridge National Laboratory
Also known for netlib, lapack, etc. Tutorial presentation:
http://www.netlib.org/utk/people/jd-tutorial/Presentation.html

PVM vs. MPI

- MPI = Message-Passing Interface
  - SPMD (Single program, multiple data)
  - Each node runs the same program
  - The program “just exists”, it is not spawned explicitly
- PVM = Parallel Virtual Machine
  - “MPMD” (Multiple program, multiple data)
  - Processes are explicitly spawned
  - Processes are assigned to nodes in separate layer, possibly multiple per node

PVM

- Can be used for heterogeneous network
- Arbitrary topology
- Messages can cross outside of host boundaries.
- Explicit packing and unpacking of messages required in code
- Fault tolerance features
- PVM came before MPI.
- Lower level, but more flexible

PVM

- In PVM *daemon* processes must be resident on nodes prior to spawning PVM processes there.
- Upon command, the daemon launches the process.
- The PVM host file identifies participating nodes, or they can be added manually from the command line.
- Root process is started from pvm console command-line on one host.

PVM

- Processes explicitly spawn child processes.
- Child can determine its parent.
- Processes have their own “task id”.
- Point-to-point send/receive similar to MPI.
- Tags, wildcards similar to MPI.

Code Fragment for simple PVM process (1)
(I have left out the error checking, etc.)

```
int main(int argc, char* argv[]) {
/* find out my task id number */
mytid = pvm_mytid();

/* find my parent's task id number */
myparent = pvm_parent();

/* if I don't have a parent then I am the parent */
if (myparent == PvmNoParent) {

/* spawn the child tasks */
info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0,
```

Code Fragment (2)

```
/* I'm still the parent */

    for (i = 0; i < ntask; i++) {
      /* recv a message from any child process */
      buf = pvm_recv(-1, JOINTAG);

      info = pvm_bufinfo(buf, &len, &tag, &tid);

      info = pvm_upkint(&mydata, 1, 1);
      }
    pvm_exit();
    }
```

Code Fragment (3)

```
  /* I'm a child */

  info = pvm_initsend(PvmDataDefault);
  info = pvm_pkint(&mytid, 1, 1);
  info = pvm_send(myparent, JOINTAG);
  pvm_exit();
}
```

See http://www.netlib.org/pvm/book for more details.

PVM groups

- Processes explicitly join and leave groups, named symbolically.
- Multicast, gather, barriers, etc. are done relative to group.
- Multicast can be into group from outside.
- Reduce operator for +, *, max, min, or *user-defined*.
- A process can be in multiple groups.

PVM

- Multicast to explicit receives, unlike MPI.

PVM

- For further info, examples, and on-line manual, see:
  - http://www.coe.uncc.edu/~abw/parallel/orig_pvm/using_pvm.html

Timing Analysis for Parallel Applications

Time Decompositon

- Parallel execution time can be divided into:
  - Actual computation time +
  - Communication time

    $t_{parallel} = t_{comp} + t_{comm}$
  - If there are m non-parallel message steps overall, then

    $t_{comm} = m * t_{message}$

Message Time Decomposition

- Message time can be divided into:
  - **Latency** (or start-up time) +
  - (number of data communicated)*(delay per datum)

$$t_{message} = t_{startup} + n * t_{datum}$$

$1/ t_{datum}$ is often called **"bandwidth"**, the number of data per unit time.

Some Comparative Times (Pacheco 1997)

Machine	Arithmetic Op	Latency	Delay per Double	Ops/Latency Period
Cray T3D	0.011 ms	21 ms	0.3 ms	1909
IBM SP-2	0.0042 ms	35 ms	0.23 ms	8333
Ethernet	N/A	500 ms	8.9 ms	N/A

Latency Hiding

- In order to prevent $t_{message}$ from destroying any speedup due to parallelism, we can try the following:
  - While a processing element is awaiting a message, perform some computation that doesn't require the message.
- Note that we are really trying to hide the entire communication cost, not just the "latency" component of it.

Latency Hiding (2)

- One technique for hiding latency is "multiprogramming":
  - On a single processor, run more than one process.
  - While one process is awaiting a message, another could be doing useful computational work.
  - This requires that process-switching be relatively efficient (e.g. using threads rather than processes).
- The ratio of processes to processors is sometimes called the "parallel slackness".

Parallel Time Complexity

- We assume familiarity with O, Ω, and Θ notation.
- Their use is to bound the time complexity as a function of the problem size "n".

Complexity Example

- Matrix-vector multiplication:
  - n x n matrix
  - n element vector
- Assume n processors
  - Every processor has a row of the matrix
  - Each row is multiplied by the vector simultaneously
  - It takes O(n) to multiply one row, so

$$t_{comp}(n) \in O(n)$$

Matrix-Vector Multiplication

- If the matrix first had to be *distributed* in order for the multiplication to take place, then the cost of distributing the rows from one processing element is $O(n^2)$, while the cost of collecting the result is O(n).
- Therefore, the asymptotic parallel cost is the same as the obvious sequential cost.

Matrix-Vector Multiplication

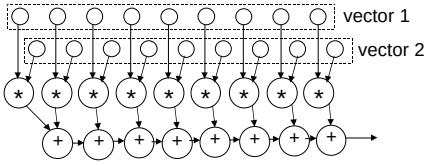
- If the same matrix is to be used many times, then the overhead of distribution gets less and less significant as the number of multiplications increases.
- In this case, the parallel cost approaches O(n), which is an improvement over the $O(n^2)$ serial method.

Cost Optimality

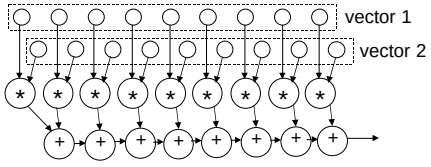
- A **cost-optimal algorithm** is defined to be one in which the **effort**, as a function of problem size, is bounded by a **constant** times the sequential effort.
- One-shot matrix-vector multiplication is not cost-optimal for distributed memory using the technique described, whereas multiplication repeated at least n times is.

Using **Graphs** to Illustrate Algorithms

- The vector inner product can be shown thus (ignoring distribution cost):



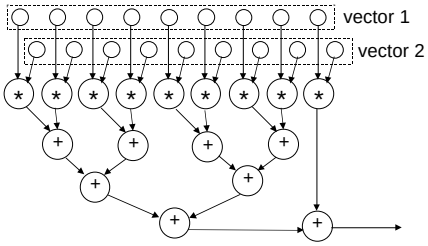
Using Graphs to Illustrate Algorithms



- Assume **unit time** for each operation.
- The time is proportional to path length.
- The longest path length for an n-element vector is O(n), sim. to serial.

Using Graphs to Illustrate Algorithms

- Restructuring the + nodes as a **tree** gives us faster performance on n processors.



Algorithm Analysis

- The previous tree gives us O(log n) on n processors.
- Is it cost optimal?

Scaling Down Processors

- As the size of the vector grows very large, we can divide the additions up among p processors, p << n, adding the elements within a processor sequentially and only using the tree at the end.
- The time is dominated by the sequential adds, which is O(n) for a given p.
- Is this cost optimal?

Generalization of Scale-down

- Whenever the number of operations (including communication as an operation) in the parallel case is proportional to the serial complexity, we can achieve cost optimality by scaling down.
- The general concept is captured by Brent's Lemma.

Brent's Lemma

- If an algorithm A entails m operations and can be done in parallel time t with *some* number of processors,

  then p processors can execute the algorithm in time

  $$t + (m-t)/p$$

  assuming added scheduling time can be ignored.

Brent's Lemma Summarized

- t = time on some number of processors
- m = number of operations (unit time each)
- time on p processors is $\leq t + (m-t)/p$

Application of Brent's Lemma

- To achieve cost optimality for vector inner product, use n/(log n) processors.
- Observed that product can be done in log n with arbitrarily-many processors.
- Brent's lemma says it can be done with n/(log n) processors in

  $$\underbrace{\log n}_{t} + \underbrace{(2n-1-\log n)}_{(m-t)} / \underbrace{(n/\log n)}_{p}$$

Application of Brent's Lemma

log n + (2n-1-log n) / (n/log n)

= log n + 2 log n - (log n)/n - $(\log n)^2/n$

which is O(log n).

Proof of Brent's Lemma (1)

- Consider the graph of the algorithm done with some number of processors in time t.
- Let s_i be the number of operations done at the i^{th} level, i.e. at "time" i.
- On p processors, we can reschedule the s_i operations in time ceiling(s_i/p).

Proof of Brent's Lemma (2)

- On p processors, we can reschedule the s_i operations in time ceiling(s_i/p).
- The total computation can therefore be done on p processors in time

  sum(i = 1 to t, ceiling(s_i/p))

Proof of Brent's Lemma (3)

sum(i = 1 to t, ceiling(s_i/p))
is bounded by
sum(i = 1 to t, (s_i+p-1)/p)

= sum(i = 1 to t, s_i/p)
+ sum(i = 1 to t, p/p)
- sum(i = 1 to t, 1/p)

= m/p + t - t/p
= t + (m-t)/p, as advertised.

Illustration of Brent

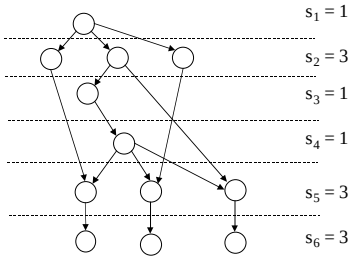
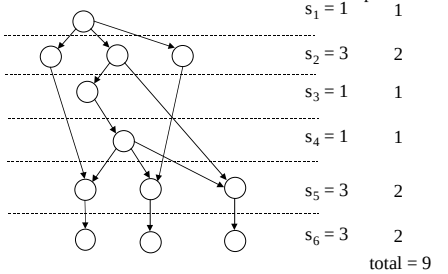


Illustration of Brent for 2 processors

Brent's bound predicts: t + (m-t)/p = 6 + (12-6)/2 = 9



Graph Exercise

- By the prefix sum problem, we mean that of computing from an array

  $x_0, x_1, x_2, \ldots, x_{n-1}$

  the array

  $(x_0), (x_0+x_1), (x_0+x_1+x_2), \ldots,(x_0+x_1+x_2+x_{n-1})$
- Can this problem be sped up using parallelism?
- Is there a cost optimal version?