CS 267: Distributed Memory Programming (MPI) and Tree-Based Algorithms

Kathy Yelick

www.cs.berkeley.edu/~yelick/cs267_sp07
Recap of Last Lecture

• Distributed memory multiprocessors
  • Several possible network topologies
  • On current systems, illusion that all nodes are directly connected to all others (performance may vary)
  • Key performance parameters:
    • Latency ($\alpha$) bandwidth ($1/\beta$), LogP for overlap details

• Message passing programming
  • Single Program Multiple Data model (SPMD)
  • Communication explicit send/receive
  • Collective communication
  • Synchronization with barriers

Continued today
Outline

Tree-Base Algorithms (after MPI wrap-up)
• A log n lower bound to compute any function in parallel
• Reduction and broadcast in O(log n) time
• Parallel prefix (scan) in O(log n) time
• Adding two n-bit integers in O(log n) time
• Multiplying n-by-n matrices in O(log n) time
• Inverting n-by-n triangular matrices in O(log n) time
• Evaluating arbitrary expressions in O(log n) time
• Evaluating recurrences in O(log n) time
• Inverting n-by-n dense matrices in O(log n) time
• Solving n-by-n tridiagonal matrices in O(log n) time
• Traversing linked lists
• Computing minimal spanning trees
• Computing convex hulls of point sets

• There are online html lecture notes for this material from the 1996 course taught by Jim Demmel
MPI Basic (Blocking) Send

MPI_SEND(start, count, datatype, dest, tag, comm)
- The message buffer is described by \((\text{start}, \text{count}, \text{datatype})\).
- The target process is specified by \(\text{dest}\) (rank within \text{comm})
- When this function returns, the buffer (A) can be reused, but the message may not have been received by the target process.

MPI_RECV(start, count, datatype, source, tag, comm, status)
- Waits until a matching \((\text{source} \text{ and tag})\) message is received
- \text{source} is rank in communicator specified by \text{comm}, or \text{MPI\_ANY\_SOURCE}
- \text{tag} is a tag to be matched on or \text{MPI\_ANY\_TAG}
- Receiving fewer than \text{count} is OK, but receiving more is an error
- \text{status} contains further information (e.g. size of message)
A Simple MPI Program

```c
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
    int rank, buf;
    MPI_Status status;
    MPI_Init(&argv, &argc);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* Process 0 sends and Process 1 receives */
    if (rank == 0) {
        buf = 123456;
        MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else if (rank == 1) {
        MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
        printf( "Received %d\n", buf );
    }

    MPI_Finalize();
    return 0;
}
```

Note: Fortran and C++ versions are in online lecture notes

Slide source: Bill Gropp, ANL
A Simple MPI Program (Fortran)

```fortran
program main
  include 'mpif.h'
  integer rank, buf, ierr, status(MPI_STATUS_SIZE)

  call MPI_Init(ierr)
  call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
  C Process 0 sends and Process 1 receives
  if (rank .eq. 0) then
    buf = 123456
    call MPI_Send( buf, 1, MPI_INTEGER, 1, 0,
                   MPI_COMM_WORLD, ierr )
  else if (rank .eq. 1) then
    call MPI_Recv( buf, 1, MPI_INTEGER, 0, 0,
                   MPI_COMM_WORLD, status, ierr )
    print *, "Received ", buf
  endif
  call MPI_Finalize(ierr)
end
```

Slide source: Bill Gropp, ANL
A Simple MPI Program (C++)

```cpp
#include "mpi.h"
#include <iostream>
int main( int argc, char *argv[])
{
    int rank, buf;
    MPI::Init(argv, argc);
    rank = MPI::COMM_WORLD.Get_rank();

    // Process 0 sends and Process 1 receives
    if (rank == 0) {
        buf = 123456;
        MPI::COMM_WORLD.Send( &buf, 1, MPI::INT, 1, 0 );
    } else if (rank == 1) {
        MPI::COMM_WORLD.Recv( &buf, 1, MPI::INT, 0, 0 );
        std::cout << "Received " << buf << "\n";
    }

    MPI::Finalize();
    return 0;
}
```
Retrieving Further Information

• **Status** is a data structure allocated in the user’s program.

• In C:

```c
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

• In Fortran:

```fortran
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..
               status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```
Retrieving Further Information

- **Status** is a data structure allocated in the user’s program.
- In C++:
  ```c++
  int recvd_tag, recvd_from, recvd_count;
  MPI::Status status;
  Comm.Recv(..., MPI::ANY_SOURCE, MPI::ANY_TAG, ..., status )
  recvd_tag   = status.Get_tag();
  recvd_from  = status.Get_source();
  recvd_count = status.Get_count( datatype );
  ```
Collective Operations in MPI

- *Collective* operations are called by all processes in a communicator
- `MPI_BCAST` distributes data from one process (the root) to all others in a communicator
- `MPI_REDUCE` combines data from all processes in communicator and returns it to one process
  - Operators include: `MPI_MAX`, `MPI_MIN`, `MPI_PROD`, `MPI_SUM`,…
- In many numerical algorithms, `SEND/RECEIVE` can be replaced by `BCAST/REDUCE`, improving both simplicity and efficiency
  - Can use a more efficient algorithm than you might choose for simplicity (e.g., P-1 send/receive pairs for broadcast or reduce)
  - May use special hardware support on some systems
Example: PI in C - 1

```c
#include "mpi.h"
#include <math.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done) {
        if (myid == 0) {
            printf("Enter the # of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
```
Example: PI in C - 2

```c
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
            MPI_COMM_WORLD);
if (myid == 0)
    printf("pi is approximately %.16f, Error is .16f\n",
            pi, fabs(pi - PI25DT));
}
MPI_Finalize();
return 0;
```
Example: PI in Fortran - 1

```fortran
program main
  include 'mpif.h'
  integer done, n, myid, numprocs, i, rc
  double pi25dt, mypi, pi, h, sum, x, z
  data done/.false./
  data PI25DT/3.141592653589793238462643/
  call MPI_Init(ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,numprocs, ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,myid, ierr)
  do while (.not. done)
    if (myid .eq. 0) then
      print *,"Enter the number of intervals: (0 quits)"
      read *, n
    endif
    call MPI_Bcast(n, 1, MPI_INTEGER, 0,
      *                MPI_COMM_WORLD, ierr)
    if (n .eq. 0) goto 10
    print *,"Calculating PI..."
    h = PI25DT / numprocs
    do i = 1, n
      x = i*h
      z = 1 / (1 - x**2)
      sum = sum + z
    enddo
    mypi = 2 * sum
    if (myid .eq. 0) then
      print *,myid, " PI = ", mypi
    endif
  enddo
```

Slide source: Bill Gropp, ANL
Example: PI in Fortran - 2

```fortran
h   = 1.0 / n
sum = 0.0
    do i=myid+1,n,numprocs
        x = h * (i - 0.5)
        sum += 4.0 / (1.0 + x*x)
    enddo
mypi = h * sum
    call MPI_Reduce(mypi, pi, 1, MPI_DOUBLE_PRECISION,
            MPI_SUM, 0, MPI_COMM_WORLD, ierr )
    if (myid .eq. 0) then
        print *, "pi is approximately ", pi,
        * "", Error is ", abs(pi - PI25DT)
    enddo
10 continue
    call MPI_Finalize( ierr )
end
```

Example: PI in C++ - 1

```cpp
#include "mpi.h"
#include <math.h>
#include <iostream>

int main(int argc, char *argv[]) {
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI::Init(argc, argv);
    numprocs = MPI::COMM_WORLD.Get_size();
    myid = MPI::COMM_WORLD.Get_rank();
    while (!done) {
        if (myid == 0) {
            std::cout << "Enter the # of intervals: (0 quits) ";
            std::cin >> n;;
        }
        MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0);
        if (n == 0) break;
    }
}
```
Example: PI in C++ - 2

```cpp
h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;
MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE,
                        MPI::SUM, 0);

if (myid == 0)
    std::cout << "pi is approximately " << pi << ", Error is " << fabs(pi - PI25DT) << "\n";

MPI::Finalize();
return 0;
```

2/9/2007
MPI Collective Routines

• Many Routines: Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv

• All versions deliver results to all participating processes.

• V versions allow the hunks to have different sizes.

• Allreduce, Reduce, Reduce_scatter, and Scan take both built-in and user-defined combiner functions.

• MPI-2 adds Alltoallw, Exscan, intercommunicator versions of most routines
Buffers

- Message passing has a small set of primitives, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance

- When you send data, where does it go? One possibility is:

```
Process 0
User data --> Local buffer --> the network --> Local buffer --> User data
```

Derived from: Bill Gropp, ANL
Avoiding Buffering

• It is better to avoid copies:

This requires that \texttt{MPI\_Send} wait on delivery, or that \texttt{MPI\_Send} return before transfer is complete, and we wait later.
Sources of Deadlocks

• Send a large message from process 0 to process 1
  • If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

• What happens with this code?

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (1)</td>
<td>Send (0)</td>
</tr>
<tr>
<td>Recv (1)</td>
<td>Recv (0)</td>
</tr>
</tbody>
</table>

• This is called “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received
Some Solutions to the “unsafe” Problem

• Order the operations more carefully:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

• Supply receive buffer at same time as send:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sendrecv(1)</td>
<td>Sendrecv(0)</td>
</tr>
</tbody>
</table>
More Solutions to the “unsafe” Problem

• Supply own space as buffer for send

Process 0                             Process 1

\texttt{Bsend(1)}                     \texttt{Bsend(0)}
\texttt{Recv(1)}                      \texttt{Recv(0)}

• Use non-blocking operations:

Process 0                             Process 1

\texttt{Isend(1)}                     \texttt{Isend(0)}
\texttt{Irecv(1)}                     \texttt{Irecv(0)}
\texttt{Waitall}                      \texttt{Waitall}
MPI’s Non-blocking Operations

• Non-blocking operations return (immediately) “request handles” that can be tested and waited on:

```c
MPI_Request request;
MPI_Status status;
MPI_Isend(start, count, datatype,
    dest, tag, comm, &request);
MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request);
MPI_Wait(&request, &status);
```
(each request must be Waited on)

• One can also test without waiting:

```c
MPI_Test(&request, &flag, &status);
```
MPI’s Non-blocking Operations (Fortran)

• Non-blocking operations return (immediately) “request handles” that can be tested and waited on:

```fortran
integer request
integer status(MPI_STATUS_SIZE)
call MPI_Isend(start, count, datatype, 
                  dest, tag, comm, request,ierr)
call MPI_Irecv(start, count, datatype, 
                  dest, tag, comm, request, ierr)
call MPI_Wait(request, status, ierr)
(Each request must be waited on)
```

• One can also test without waiting:

```fortran
call MPI_Test(request, flag, status, ierr)
```
MPI’s Non-blocking Operations (C++)

• Non-blocking operations return (immediately) “request handles” that can be tested and waited on:

```cpp
MPI::Request request;
MPI::Status status;

request = comm.Isend(start, count,
                       datatype, dest, tag);
request = comm.Irecv(start, count,
                       datatype, dest, tag);
request.Wait(status);
```

(each request must be Waited on)

• One can also test without waiting:

```cpp
flag = request.Test( status );
```
Other MPI Point-to-Point Features

• It is sometimes desirable to wait on multiple requests:
  \[
  \text{MPI\_Waitall(count, array\_of\_requests, array\_of\_statuses)}
  \]

• Also MPI\_Waitany, MPI\_Waitsome, and test versions

• MPI provides multiple *modes* for sending messages:
  • Synchronous mode (**MPI\_Ssend**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  • Buffered mode (**MPI\_Bsend**): user supplies a buffer to the system for its use. (User allocates enough memory to avoid deadlock.)
  • Ready mode (**MPI\_Rsend**): user guarantees that a matching receive has been posted. (Allows access to fast protocols; undefined behavior if matching receive not posted.)
Synchronization

- Global synchronization is available in MPI
  - C: `MPI_Barrier( comm )`
  - Fortran: `MPI_Barrier( comm, ierr )`
  - C++: `comm.Barrier();`

- Blocks until all processes in the group of the communicator `comm` call it.

- Almost never required to make a message passing program correct
  - Useful in measuring performance and load balancing
Tree-Based Computation

• The broadcast and reduction operations in MPI are a good example of tree-based algorithms
• For reductions: take n inputs and produce 1 output
• For broadcast: take 1 input and produce n outputs
• What can we say about such computations in general?
A log n lower bound to compute any function of n variables

- Assume we can only use binary operations, one per time unit
- After 1 time unit, an output can only depend on two inputs
- Use induction to show that after k time units, an output can only depend on $2^k$ inputs
  - After $\log_2 n$ time units, output depends on at most n inputs
- A binary tree performs such a computation
Broadcasts and Reductions on Trees
Parallel Prefix, or Scan

- If “+” is an associative operator, and x[0],…,x[p-1] are input data then parallel prefix operation computes
  \[ y[j] = x[0] + x[1] + \ldots + x[j] \quad \text{for } j=0,1,\ldots,p-1 \]
- Notation: \[ j:k \text{ mean } x[j]+x[j+1]+\ldots+x[k], \text{ blue is final value } \]
Mapping Parallel Prefix onto a Tree - Details

- **Up-the-tree phase (from leaves to root)**
  1) Get values L and R from left and right children
  2) Save L in a local register Lsave
  3) Pass sum L+R to parent

  By induction, Lsave = sum of all leaves in left subtree

- **Down the tree phase (from root to leaves)**
  1) Get value S from parent (the root gets 0)
  2) Send S to the left child
  3) Send S + Lsave to the right child

  By induction, S = sum of all leaves to left of subtree rooted at the parent
E.g., Fibonacci via Matrix Multiply Prefix

• Consider computing of the Fibonacci numbers:

\[
F_{n+1} = F_n + F_{n-1}
\]

• Each step can be viewed as a matrix multiplication:

\[
\begin{pmatrix}
F_{n+1} \\
F_n
\end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}
\]

Can compute all \( F_n \) by matmul_prefix on

\[
\left[ \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \right]
\]

then select the upper left entry

Derived from: Alan Edelman, MIT
Adding two n-bit integers in $O(\log n)$ time

• Let $a = a[n-1]a[n-2]...a[0]$ and $b = b[n-1]b[n-2]...b[0]$ be two n-bit binary numbers

• We want their sum $s = a+b = s[n]s[n-1]...s[0]$

  $c[-1] = 0$ \quad … rightmost carry bit
  for $i = 0$ to $n-1$
    $c[i] = ( (a[i] \text{ xor } b[i]) \text{ and } c[i-1] ) \text{ or } ( a[i] \text{ and } b[i] )$ \quad … next carry bit
    $s[i] = ( a[i] \text{ xor } b[i] ) \text{ xor } c[i-1]$

• **Challenge: compute all $c[i]$ in $O(\log n)$ time via parallel prefix**

  for all $(0 \leq i \leq n-1)$ $p[i] = a[i] \text{ xor } b[i]$ \quad … propagate bit
  for all $(0 \leq i \leq n-1)$ $g[i] = a[i] \text{ and } b[i]$ \quad … generate bit

  $c[i] = \begin{pmatrix} p[i] \text{ and } c[i-1] \end{pmatrix} \text{ or } g[i] = \begin{pmatrix} p[i] & g[i] \end{pmatrix} \times \begin{pmatrix} c[i-1] \\ 1 \end{pmatrix} = C[i] \times \begin{pmatrix} c[i-1] \\ 1 \end{pmatrix}$

  … 2-by-2 Boolean matrix multiplication (associative)
  $= C[i] \times C[i-1] \times \ldots \times C[0] \times \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

  … evaluate each $P[i] = C[i] \times C[i-1] \times \ldots \times C[0]$ by parallel prefix

• **Used in all computers to implement addition - ** Carry look-ahead

2/9/2007
Other applications of scans

• There are several applications of scans, some more obvious than others
  • add multi-precision numbers (represented as array of numbers)
  • evaluate recurrences, expressions
  • solve tridiagonal systems (numerically unstable!)
  • implement bucket sort and radix sort
  • to dynamically allocate processors
  • to search for regular expression (e.g., grep)
• Names: +\ (APL), cumsum (Matlab), MPI_SCAN
• Note: 2n operations used when only n-1 needed
Evaluating arbitrary expressions

- Let E be an arbitrary expression formed from +, -, *, /, parentheses, and n variables, where each appearance of each variable is counted separately
- Can think of E as arbitrary expression tree with n leaves (the variables) and internal nodes labeled by +, -, * and /
- Theorem (Brent): E can be evaluated in $O(\log n)$ time, if we reorganize it using laws of commutativity, associativity and distributivity
- Sketch of (modern) proof: evaluate expression tree E greedily by
  - collapsing all leaves into their parents at each time step
  - evaluating all “chains” in E with parallel prefix
Multiplying n-by-n matrices in $O(\log n)$ time

- For all $(1 \leq i,j,k \leq n)$, $P(i,j,k) = A(i,k) \times B(k,j)$
  - cost = 1 time unit, using $n^3$ processors

- For all $(1 \leq i,j \leq n)$, $C(i,j) = \sum_{k=1}^{n} P(i,j,k)$
  - cost = $O(\log n)$ time, using a tree with $n^3 / 2$ processors
Evaluating recurrences

- Let $x_i = f_i(x_{i-1})$, $f_i$ a rational function, $x_0$ given
- How fast can we compute $x_n$?
- Theorem (Kung): Suppose $\text{degree}(f_i) = d$ for all $i$
  - If $d=1$, $x_n$ can be evaluated in $O(\log n)$ using parallel prefix
  - If $d>1$, evaluating $x_n$ takes $\Omega(n)$ time, i.e. no speedup is possible
- Sketch of proof when $d=1$

\[
x_i = f_i(x_{i-1}) = \frac{a_i \cdot x_{i-1} + b_i}{c_i \cdot x_{i-1} + d_i}
\]

\[
x_i = \frac{\text{num}_i}{\text{den}_i} = \frac{(a_i \cdot \text{num}_{i-1} + b_i \cdot \text{den}_{i-1})/(c_i \cdot \text{num}_{i-1} + d_i \cdot \text{den}_{i-1})}{(\text{num}_i)/(\text{den}_i)}
\]

\[
\begin{pmatrix}
\text{num}_i \\
\text{den}_i
\end{pmatrix} = \begin{pmatrix}
a_i & b_i \\
c_i & d_i
\end{pmatrix} \begin{pmatrix}
\text{num}_{i-1} \\
\text{den}_{i-1}
\end{pmatrix} = M_i \begin{pmatrix}
\text{num}_{i-1} \\
\text{den}_{i-1}
\end{pmatrix} = M_i \cdot M_{i-1} \cdot \ldots \cdot M_1 \cdot \begin{pmatrix}
\text{num}_0 \\
\text{den}_0
\end{pmatrix}
\]

Can use parallel prefix with 2-by-2 matrix multiplication

- Sketch of proof when $d>1$
  - $\text{degree}(x_i)$ as a function of $x_0$ is $d^i$
  - After $k$ parallel steps, $\text{degree}(\text{anything}) \leq 2^k$
  - Computing $x_i$ take $\Omega(i)$ steps
Summary

• Message passing programming
  • Maps well to large-scale parallel hardware (clusters)
  • Most popular programming model for these machines
  • A few primitives are enough to get started
    • send/receive or broadcast/reduce plus initialization
  • More subtle semantics to manage message buffers to avoid copying and speed up communication

• Tree-based algorithms
  • Elegant model that is a key piece of data-parallel programming
  • Most common are broadcast/reduce
  • Parallel prefix (aka scan) has produces partial answers and can be used for many surprising applications
    • Some of these or more theoretical than practical interest
Extra Slides
Inverting triangular matrices in $O(\log^2 n)$ time

- Fact:

\[
\begin{pmatrix}
A & 0 \\
C & B
\end{pmatrix}^{-1} = \begin{pmatrix}
A^{-1} & 0 \\
-B^{-1}CA^{-1} & B^{-1}
\end{pmatrix}
\]

- Function $\text{Tri}_\text{Inv}(T)$ … assume $n = \text{dim}(T) = 2^m$ for simplicity

If $T$ is 1-by-1
returns $\frac{1}{T}$
else

... Write $T = \begin{pmatrix} A & 0 \\ C & B \end{pmatrix}$

In parallel do {
  invA = $\text{Tri}_\text{Inv}(A)$
  invB = $\text{Tri}_\text{Inv}(B)$
} ... implicitly uses a tree

newC = $-\text{invB} \cdot C \cdot \text{invA}$

Return $\begin{pmatrix}
\text{invA} & 0 \\
\text{newC} & \text{invB}
\end{pmatrix}$

- $\text{time}(\text{Tri}_\text{Inv}(n)) = \text{time}(\text{Tri}_\text{Inv}(n/2)) + O(\log(n))$
  - Change variable to $m = \log n$ to get $\text{time}(\text{Tri}_\text{Inv}(n)) = O(\log^2 n)$
Inverting Dense n-by-n matrices in $O(\log^2 n)$ time

• Lemma 1: Cayley-Hamilton Theorem
  • expression for $A^{-1}$ via characteristic polynomial in $A$

• Lemma 2: Newton’s Identities
  • Triangular system of equations for coefficients of characteristic polynomial, matrix entries = $s_k$

• Lemma 3: $s_k = \text{trace}(A^k) = \sum_{i=1}^{n} A^k[i,i] = \sum_{i=1}^{n} [\lambda_i(A)]^k$

• Csanky’s Algorithm (1976)
  1) Compute the powers $A^2, A^3, ..., A^{n-1}$ by parallel prefix
     cost = $O(\log^2 n)$
  2) Compute the traces $s_k = \text{trace}(A^k)$
     cost = $O(\log n)$
  3) Solve Newton identities for coefficients of characteristic polynomial
     cost = $O(\log^2 n)$
  4) Evaluate $A^{-1}$ using Cayley-Hamilton Theorem
     cost = $O(\log n)$

• Completely numerically unstable
Summary of tree algorithms

• Lots of problems can be done quickly - in theory - using trees
• Some algorithms are widely used
  • broadcasts, reductions, parallel prefix
  • carry look ahead addition
• Some are of theoretical interest only
  • Csanky’s method for matrix inversion
  • Solving general tridiagonals (without pivoting)
  • Both numerically unstable
  • Csanky needs too many processors
• Embedded in various systems
  • CM-5 hardware control network
  • MPI, Split-C, Titanium, NESL, other languages