What is and Why Concurrency?

• What is a concurrent program?
  – One with more than one active execution context (thread of control)
• Why concurrency?
  – Capture the logical structure of certain applications
  – Cope with independent physical devices
  – Performance through use of multiple processors

Programming Models

• Standard models of parallelism
  – shared memory (Pthreads)
  – message passing (MPI)
  – data parallel (Fortran 90 and HPF)
  – shared memory + data parallel (OpenMP)
  – Remote procedure call

Shared Memory

Thread Creation Syntax

• Properly nested (can share context)
  – Co-Begin (Algol 68, Occam, SR)
  – Parallel loops (HPF, Occam, Fortran90, SR)
  – Launch-at-Elaboration (Ada, SR)
• Fork/Join (pthreads, Ada, Modula-3, Java, SR)
• Implicit Receipt (RPC systems, SR)
• Early Reply (SR)

Loops

• For – sequential
• Forall – each statement executed completely and in parallel
• Dopar – each iteration executed in parallel
• Dosingle – each variable assigned once, new value always used

Sequential SOR

```c
for (k = 0; k < 100; k++) {
    for (j = 1; j < M-1; j++)
        for (i = 1; i < M-1; i++)
            a[j][i] = (b[j][i-1] + b[j][i+1] + b[j-1][i] + b[j+1][i])/4;
    for (j = 1; j < M-1; j++)
        for (i = 1; i < M-1; i++)
            b[j][i] = a[j][i];
} ```
Shared Memory Version

• for (k = 0; k < 100; k++) {
  • for (j = begin; j < end; j++)
    • for (i = 1; i < M-1; i++)
      • a[j][i] = (b[j][i-1] + b[j][i+1] + b[j-1][i] + b[j+1][i])/4;
      • barrier();
  •}
  • for (j = begin; j < end; j++)
  • for (i = 1; i < M-1; i++)
  • b[j][i] = a[j][i];
  • barrier();
•}

Data Parallel Version of SOR (Power C)

• for (k = 0; k < 100; k++) {
  •#pragma parallel shared(a, b) local(i,j)
    •#pragma pfor
      • for (j = 1; j < M-1; j++)
        • for (i = 1; i < M-1; i++)
          • a[j][i] = (b[j][i-1] + b[j][i+1] + b[j-1][i] + b[j+1][i])/4;
  •#pragma parallel shared(a, b) local(i, j)
    •#pragma pfor
      • for (j = 1; j < M-1; j++)
        • for (i = 1; i < M-1; i++)
          • b[j][i] = a[j][i];
•}

SOR in HPF (Fortran D)

real a(1000, 1000), b(1000, 1000)
C    decomposition d(1000, 1000)
C    align a, b with d
C    distribute d(:, block)
do k = 1, 1000
  do j = 2, 999
    do i = 2, 999
      a(i,j) = F(b(i-1,j),b(i+1,j),b(i,j-1),b(i,j+1))
    enddo
  enddo
second loop (b(i,j) = a(i,j))
enddo

Programming Models

• Standard models of parallelism
  – shared memory (Pthreads)
  – message passing (MPI)
  – data parallelism (Fortran 90 and HPF)
  – shared memory + data parallelism (OpenMP)

Message Passing Systems

• Provide process creation and destruction
• Provide message passing facilities (send and receive, in various flavors) to distribute and communicate data
• Provide additional synchronization facilities

Distributed Memory - Message Passing

mem1  mem2  mem3  memN
proc1  proc2  proc3  procN

network
What does the user have to do?

- This is what we said for shared memory:
  - Decide how to decompose the computation into parallel parts.
  - Create (and destroy) processes to support that decomposition.
  - Add synchronization to make sure dependences are covered.
- Is the same true for message passing?

Another Look at SOR Example

```c
for some number of timesteps/iterations {
    for (i=0; i<n; i++)
        for( j=0; j<n; j++ )
            temp[i][j] = 0.25 *
                (grid[i-1][j] + grid[i+1][j]
                 grid[i][j-1] + grid[i][j+1]);
    for( i=0; i<n; i++)
        for( j=0; j<n; j++ )
            grid[i][j] = temp[i][j];
}
```

Shared Memory

Data Distribution (only middle processes)

Data Communication (only middle processes)

Index Translation

- Reduces space declaration
  ```c
  for( i=1; i<n/p; i++)
      for( j=1; j<n; j++ )
          temp[i][j] = 0.25*( grid[i-1][j] + grid[i+1][j]
                          grid[i][j-1] + grid[i][j+1]);
  ```

Remember, all variables are local.
Index Translation is Optional

- Allocate the full arrays on each processor
- Leave indices alone
- Higher memory use
- Sometimes necessary (see later)

What does the user need to do?

- Divide up the program into parallel parts
- Create and destroy processes to do above
- Partition and distribute the data
- Communicate data at the right time
- (Sometimes) perform index translation
- Still need to perform synchronization?
  - Sometimes, but many times goes hand in hand with data communication

Message Passing Systems

- Provide process creation and destruction
- Provide message passing facilities (send and receive, in various flavors) to distribute and communicate data
- Provide additional synchronization facilities

MPI (Message Passing Interface)

- Is the de facto message passing standard
- Available on virtually all platforms, including public domain versions (MPICH)
- Grew out of an earlier message passing system, PVM, still actively used, but now outdated

MPI Process Creation/Destruction

```c
MPI_Init( int argc, char **argv )
    Initializes the MPI execution environment
MPI_Finalize()
    Terminates MPI execution environment (all processes must call this routine before exiting)
```

MPI Group Communication

- Communicators: provide a scope for all communication
- Groups: define an ordered collection of participant processes, each with a rank

  E.g., `MPI_COMM_WORLD`
  - Predefined communicator consisting of the group of all processes initiated for a parallel program
MPI Process Identification

MPI_Comm_size( comm, &size )
Determines the number of processes

MPI_Comm_rank( comm, &pid )
Pid is the process identifier of the caller

MPI Basic Send (Blocking)

MPI_Send(buf, count, datatype, dest, tag, comm)
buf: address of send buffer
count: number of elements
datatype: data type of send buffer elements
dest: process id of destination process
tag: message tag (ignore for now)
comm: communicator (ignore for now)

MPI Basic Receive (Blocking)

MPI_Recv(buf, count, datatype, source, tag, comm, & status)
buf: address of receive buffer
count: size of receive buffer in elements
datatype: data type of receive buffer elements
source: source process id or MPI_ANY_SOURCE
tag and comm: ignore for now
status: status object

MPI Matrix Multiply (w/o Index Translation)

main(int argc, char * argv[])
{
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
for( i=0; i<p; i++) {
    from[i] = (i * n)/p;
    to[i] = ((i+1) * n)/p;
}
/* Data distribution */ ...
/* Computation */ ...
/* Result gathering */ ...
MPI_Finalize();
}

/* Data distribution */
if( myrank != 0 ) {
    MPI_Recv( &a[from[myrank]], n*n/p, MPI_INT, 0, tag,
    MPI_COMM_WORLD, &status );
    MPI_Recv( &b, n*n, MPI_INT, 0, tag, MPI_COMM_WORLD,
    &status );
} else {
    for( i=1; i<p; i++) {
        MPI_Send( &a[from[i]], n*n/p, MPI_INT, i, tag,
        MPI_COMM_WORLD );
        MPI_Send( &b, n*n, MPI_INT, i, tag, MPI_COMM_WORLD );
    }
}

/* Computation */
for ( i=from[myrank]; i<to[myrank]; i++) {
    for (j=0; j<n; j++) {
        C[i][j]=0;
        for (k=0; k<n; k++)
            C[i][j] += A[i][k]*B[k][j];
    }
}
MPI Matrix Multiply (w/o Index Translation)

/* Result gathering */
if (myrank!=0)
    MPI_Send( &c[from[myrank]], n*n/p, MPI_INT, 0,
        tag, MPI_COMM_WORLD);
else
    for( i=1; i<p; i++ )
        MPI_Recv( &c[from[i]], n*n/p, MPI_INT,
            i, tag, MPI_COMM_WORLD, &status);

MPI Matrix Multiply (with Index Translation)

main(int argc, char *argv[])
{
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    for( i=0; i<p; i++ )
        from[i] = (i * n)/p;
    /* Data distribution */ ...
    /* Computation */ ...
    /* Result gathering */ ...
    MPI_Finalize();
}

MPI Matrix Multiply (with Index Translation)

/* Data distribution */
if( myrank != 0 ) {
    MPI_Recv( &a, n*n/p, MPI_INT, 0, tag, MPI_COMM_WORLD,
        &status );
    MPI_Recv( &b, n*n, MPI_INT, 0, tag, MPI_COMM_WORLD,
        &status );
} else {
    for( i=1; i<p; i++ )
        MPI_Send( &a[from[i]], n*n/p, MPI_INT, i, tag,
            MPI_COMM_WORLD );
        MPI_Send( &b, n*n, MPI_INT, i, tag,
            MPI_COMM_WORLD );
}

MPI Matrix Multiply (with Index Translation)

/* Computation */
for ( i=0; i<to[i]-from[i]; i++)
    for (j=0; j<n; j++)
        c[i][j]=0;
    for (k=0; k<n; k++)
        c[i][j] += a[i][k]*b[k][j];

MPI Matrix Multiply (with Index Translation)

/* Result gathering */
if (myrank!=0)
    MPI_Send( &c, n*(to[i]-from[i]), MPI_INT, 0,
        tag, MPI_COMM_WORLD);
else
    for( i=1; i<p; i++ )
        MPI_Recv( &c[from[i]], n*(to[i]-from[i]),
            MPI_INT, i, tag,
            MPI_COMM_WORLD, &status);

Types of Communication

• Blocking – if return from the procedure indicates
  the user is allowed to use (or re-use) resources
  (such as buffers) specified in the call
• Non-blocking – if the procedure may return before
  the operation completes, thereby not allowing the
  user to re-use resources
• Collective – if all processes in a process group
  need to invoke the procedure
Running an MPI Program
• `mpirun <program_name> <arguments>`
• Causes a Unix process to be run on each of the hosts

Global Operations (1 of 2)
• So far, we have only looked at point-to-point or one-to-one message passing facilities
• Often, it is useful to have one-to-many or many-to-one message communication
• This is what MPI’s global operations do

Global Operations (2 of 2)
• `MPI_Barrier`
• `MPI_Bcast`
• `MPI_Gather`
• `MPI_Scatter`
• `MPI_Reduce`
• `MPI_Allreduce`

Barrier
`MPI_Barrier(comm)`
Global barrier synchronization, as before: all processes wait until all have arrived.

Broadcast
`MPI_Bcast(inbuf, incnt, intype, root, comm)`

- `inbuf`: address of input buffer (on root); address of output buffer (elsewhere)
- `incnt`: number of elements
- `intype`: type of elements
- `root`: process id of root process

Before Broadcast
```
    proc0 | proc1 | proc2 | proc3
       root

inbuf
```
After Broadcast

MPI_Scatter(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)
inbuf: address of input buffer
incnt: number of elements sent to each process
intype: type of input elements
outbuf: address of output buffer
outcnt: number of output elements
outtype: type of output elements
root: process id of root process

Before Scatter

Gather

MPI_Gather(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)
inbuf: address of input buffer
incnt: number of input elements
intype: type of input elements
outbuf: address of output buffer
outcnt: number of output elements
outtype: type of output elements
root: process id of root process
After Gather

Broadcast/Scatter/Gather

- These three primitives are sends and receives at the same time
- Perhaps un-intended consequence: requires global agreement on layout of data

MPI Matrix Multiply Revised (1 of 2)

//main function
main(int argc, char *argv[])
{
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    from = (myrank * n)/p;
    to = ((myrank+1) * n)/p;
    MPI_Scatter(a, n*n/p, MPI_INT, a, n*n/p, MPI_INT, 0,
                MPI_COMM_WORLD);
    MPI_Bcast(b,n*n, MPI_INT, 0, MPI_COMM_WORLD);
    ...
}

MPI Matrix Multiply Revised (2 of 2)

//MPI gather
for (i=from; i<to; i++)
    for (j=0; j<n; j++)
    { C[i][j]=0;
      for (k=0; k<n; k++)
        C[i][j] += A[i][k]*B[k][j];
    }
    MPI_Gather (C[from], n*n/p, MPI_INT, c[from], n*n/p ,
                MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Finalize();

SOR Sequential Code

for some number of timesteps/iterations {
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
        { temp[i][j] = 0.25 *
          ( grid[i-1][j] + grid[i+1][j] +
            grid[i][j-1] + grid[i][j+1] );
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                grid[i][j] = temp[i][j];
    }

MPI SOR

- Allocate grid and temp arrays.
- Use MPI_Scatter to distribute initial values, if any (requires non-local allocation).
- Use MPI_Gather to return the results to process 0 (requires non-local allocation).
- Focusing only on communication within the computational part ...
Data Communication (only middle processes)

PROC2

PROC3

MPI SOR

for some number of timesteps/iterations {
    for (i=from; i<to; i++)
        for (j=0; j<n, j++)
            temp[i][j] = 0.25 *
                (grid[i-1][j] + grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
    for (i=from; i<to; i++)
        for (j=0; j<n; j++)
            grid[i][j] = temp[i][j];
/* here comes communication */
}

MPI SOR Communication

if (myrank != 0) {
    MPI_Send (grid[from], n, MPI_DOUBLE, myrank-1, tag, MPI_COMM_WORLD);
    MPI_Recv (grid[from-1], n, MPI_DOUBLE, myrank-1, tag, MPI_COMM_WORLD, &status);
}
if (myrank != p-1) {
    MPI_Send (grid[to-1], n, MPI_DOUBLE, myrank+1, tag, MPI_COMM_WORLD);
    MPI_Recv (grid[to], n, MPI_DOUBLE, myrank+1, tag, MPI_COMM_WORLD, &status);
}

No Barrier Between Loop Nests?

• Not necessary
• Anti-dependences do not need to be covered in message passing
• Memory is private, so overwrite does not matter

SOR: Terminating Condition

• Real versions of SOR do not run for some fixed number of iterations
• Instead, they test for convergence
• Possible convergence criterion: difference between two successive iterations is less than some delta

SOR Sequential Code with Convergence

for( ; diff > delta; ) {
    for (i=0; i<n; i++)
        for (j=0; j<n, j++)
            diff = max(diff, fabs(grid[i][j] - temp[i][j]));
    grid[i][j] = temp[i][j];
}
Reduction

MPI_Reduce(inbuf, outbuf, count, type, op, root, comm)

inbuf: address of input buffer
outbuf: address of output buffer
count: number of elements in input buffer
type: datatype of input buffer elements
op: operation (MPI_MIN, MPI_MAX, etc.)
root: process id of root process

Global Reduction

MPI_Allreduce(inbuf, outbuf, count, type, op, comm)

inbuf: address of input buffer
outbuf: address of output buffer
count: number of elements in input buffer
type: datatype of input buffer elements
op: operation (MPI_MIN, MPI_MAX, etc.)
no root process

MPI SOR Code with Convergence

for( ; diff > delta; ) {
  for (i=from; i<to; i++)
    for( j=0; j<n, j++) {
        ...
        mydiff = 0.0;
        for (i=from; i<to; i++)
          for (j=0; j<n; j++) {
            mydiff=max(mydiff,fabs(grid[i][j]-temp[i][j]));
            grid[i][j] = temp[i][j];
          }
        MPI_Allreduce (&mydiff, &diff, 1, MPI_DOUBLE,
                      MPI_MAX, MPI_COMM_WORLD);
    ...
}

Molecular Dynamics

for some number of timesteps {
  for( i=0; i<num_mol; i++ )
    for( j=0; j<count[i]; j++ )
      force[i] += f(loc[i],loc[index[i][j]]);
  for( i=0; i<num_mol; i++ )
    loc[i] = g( loc[i], force[i] );
}

Molecular Dynamics (continued)

• 1st i-loop: no loop-carried dependences.
• 2nd i-loop: no loop-carried dependences.
• Anti-dependence between 1st and 2nd loop.
• True dependence between 2nd and 1st loop.
• Let’s assume block distribution in i
  (may have load balance problems).

Shared Memory Molecular Dynamics

for some number of timesteps {
  for( i=from; i<to; i++ )
    for( j=0; j<count[i]; j++ )
      force[i] += f(loc[i],loc[index[i][j]]);
  barrier();
  for( i=from; i<to; i++ )
    loc[i] = g( loc[i], force[i] );
  barrier();
}
Message Passing Molecular Dynamics

• No need for synchronization between loops.
• What to send at the end of an outer loop iteration?
  – Send our part of loc to all processes (single broadcast per process, but perhaps inefficient).
  – Figure out who needs what in separate phase.
  – What if count/index change?
  – What if more complicated work distribution?

Sequential vs. Parallel Execution

• Sequential
  (Color -- picture; horizontal line -- processor).

• Parallel

PIPE: Sequential Program

```c
for( i=0; i<num_pic, read(in_pic); i++ ) {
  int_pic_1[i] = trans1( in_pic );
  int_pic_2[i] = trans2( int_pic_1[i] );
  int_pic_3[i] = trans3( int_pic_2[i] );
  out_pic[i] = trans4( int_pic_3[i] );
}
```

PIPE: Parallel Program

```c
P0:for( i=0; i<num_pics, read(in_pic); i++ ) {
  int_pic_1[i] = trans1( in_pic );
  signal( event_1_2[i] );
}
```

```c
P1: for( i=0; i<num_pics; i++ ) {
  wait( event_1_2[i] );
  int_pic_2[i] = trans2( int_pic_1[i] );
  signal( event_2_3[i] );
}
```

PIPE: MPI Parallel Program

```c
P0:for( i=0; i<num_pics, read(in_pic); i++ ) {
  int_pic_1[i] = trans1( in_pic );
  MPI_Send( int_pic_1[i], n, MPI_INT, 1, tag, comm );
}
```

```c
P1: for( i=0; i<num_pics; i++ ) {
  MPI_Recv(int_pic_1[i], n, MPI_INT, tag, comm, &stat);
  int_pic_2[i] = trans2( int_pic_1[i] );
  MPI_Send( int_pic_2[i], n, MPI_INT, 2, tag, comm );
}
```

PIPE: MPI Better Parallel Program

```c
P0:for( i=0; i<num_pics, read(in_pic); i++ ) {
  int_pic_1 = trans1( in_pic );
  MPI_Send( int_pic_1, n, MPI_INT, 1, tag, comm );
  MPI_Send( int_pic_1, n, MPI_INT, 1, tag, comm );
}
```

```c
P1: for( i=0; i<num_pics; i++ ) {
  MPI_Recv(int_pic_1, n, MPI_INT, tag, comm, &stat);
  int_pic_2 = trans2( int_pic_1 );
  MPI_Send( int_pic_2, n, MPI_INT, 2, tag, comm );
  MPI_Send( int_pic_2, n, MPI_INT, 2, tag, comm );
}
```
Why This Change?

- Anti-dependences on int_pic_1 between P0 and P1, etc., prevent parallelization.
- Remember: anti-dependences do not matter in message passing programs.
- Reason: the processes do not share memory, thus no worry that P1 overwrites what P0 still has to read.

Caveat

- The memory usage is not necessarily decreased in the program
- The buffers now appear inside the message passing library, rather than in the program

TSP: Sequential Program

init_q(); init_best();
while( (p=de_queue()) != NULL ) {
    for each expansion by one city {
        q = add_city(p);
        if( complete(q) ) { update_best(q) ;
            else { en_queue(q) ;
        }
    }
}

Parallel TSP: Possibilities

- Have each process do one expansion.
- Have each process do expansion of one partial path.
- Have each process do expansion of multiple partial paths.
- Issue of granularity/performance, not an issue of correctness.

TSP: MPI Program Structure

- Have a coordinator/worker scheme:
  - Coordinator maintains shared data structures (priority queue and current best path).
  - Workers expand one partial path at a time.
- Sometimes also called client/server:
  - Workers/clients issue requests.
  - Coordinator/server responds to requests.

TSP: MPI Main Program

main() {
    MPI_Init();
    MPI_Comm_rank( comm, &myrank );
    MPI_Comm_size ( comm, &p);
    /* Read input and distribute */
    if( myrank == 0 )
        Coordinator();
    else
        Worker();
    MPI_Finalize();
}
TSP: MPI Communication

- From worker to coordinator:
  - request for new partial path to explore
  - insertion of new partial path in queue
  - update coordinator if new best path is found
- From coordinator to worker:
  - new partial path to expand
  - end of computation (no more partial paths)
  - new best path length, if any

How to Distinguish Messages?

Use the tag field in the messages

\[
\text{MPI\_Send(buf, count, datatype, dest, tag, comm)}
\]

\[
\text{MPI\_Recv(buf, count, datatype, source, tag, comm, \&status)}
\]

Define a set of message tags:

\text{PUT\_PATH, GET\_PATH, BEST\_PATH, DONE, ...}
Define a corresponding set of message records

More on status in \text{MPI\_Recv()}

- Status is a record with fields
  - MPI\_SOURCE
  - MPI\_TAG
- Thus, on a \text{MPI\_Recv()}, when you specify
  MPI\_ANY\_TAG, MPI\_ANY\_SOURCE,
  you can find out the tag and the source from
  the status field.

TSP: MPI Communication

- From worker to coordinator:
  - request for new partial path to explore
  - insertion of new partial path in queue
  - update coordinator if new best path is found
- From coordinator to worker:
  - new partial path to expand
  - end of computation (no more partial paths)
  - new best path length, if any

TSP Worker (1 of 3)

\[
\text{MPI\_Send( NULL, 0, MPI\_INT, GET\_PATH, comm)};
\text{for( ;; ) { }
  \text{MPI\_Recv( \&msg, MSGSIZE, MPI\_INT, 0,
    MPI\_ANY\_TAG, comm, \&status );}
  \text{switch( status.MPI\_TAG ) { }
    \text{case NEW\_PATH: NewPath();}
    \text{case WORKER\_BEST\_PATH: WorkerBestPath();}
    \text{case DONE: exit(0);}
  }
}\]

TSP Worker (2 of 3)

\[
\text{WorkerBestPath()}\]
\[
\text{update bestlength;}
\]
TSP Worker (3 of 3)

```c
NewPath(p)
{
  for (each city not in path p)
  {
    q = expand by that city;
    if (q->length < bestlength)
      if (complete(q))
        MPI_Send(&msg, MSGSIZE, MPI_INT, 0, BEST_PATH, comm);
      else
        MPI_Send(&msg, MSGSIZE, MPI_INT, 0, PUT_PATH, comm);
  }
  MPI_Send(NULL, 0, MPI_INT, GET_PATH, comm);
}
```

TSP: MPI Communication

- From worker to coordinator:
  - request for new partial path to explore
  - insertion of new partial path in queue
  - update coordinator if new best path is found
- From coordinator to worker:
  - new partial path to expand
  - end of computation (no more partial paths)
  - new best path length, if any

TSP: MPI Coordinator (1 of 4)

```c
for (;;)
{
  MPI_Recv(&msg, MSGSIZE, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
  switch (status.MPI_TAG)
  {
    case GET_PATH: GetPath();
    case PUT_PATH: PutPath();
    case BEST_PATH: BestPath();
  }
}
```

TSP: MPI Coordinator (2 of 4)

```c
BestPath()
{
  if (msg.length < bestlength) {
    update bestlength;
    for (i=1; i<p; i++)
      MPI_Send(&msg, MSGSIZE, MPI_INT, WORKER_BEST_PATH, comm);
  }
}
```

TSP: MPI Coordinator (3 of 4)

```c
GetPath()
{
  if (not empty) {
    construct msg;
    MPI_Send(&msg, MSGSIZE, MPI_INT, status.MPI_SOURCE, NEW_PATH, comm);
  } else {
    waiting[w++] = status.MPI_SOURCE;
    if (w == p-1)
      for (i=1; i<p; i++)
        MPI_Send(NULL, 0, MPI_INT, DONE, comm);
  }
}
```

TSP: MPI Coordinator (4 of 4)

```c
PutPath()
{
  if (w>0) {
    MPI_Send(&msg, MSGSIZE, MPI_INT, waiting[--w], NEW_PATH, comm);
  } else {
    insert in q;
  }
}
```
A Problem with This Solution

• The coordinator does nothing else than maintaining shared state (updating it, and responding to queries about it).
• It is possible to have the coordinator perform computation through the MPI asynchronous communication facility.

MPI Asynchronous Communication

MPI_Iprobe(source, tag, comm, &flag, &status)
source: process id of source
tag: tag of message
comm: communicator (ignore)
flag: true if message is available, false otherwise
status: return status record (source and tag)
Checks for the presence of a message without blocking.

TSP: Revised Coordinator

for( ;; ) {
    flag = true;
    for( ;flag; ) {
        MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &flag, &status);
        if( flag ) {
            MPI_Recv(&msg, MSGSIZE, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
            switch( status.MPI_TAG ) { … }
        }
    }
    remove next partial path from queue as in worker
}

Remarks about This Solution

• Not guaranteed to be an improvement.
• If coordinator was mostly idle, this structure will improve performance.
• If coordinator was the bottleneck, this structure will make performance worse.
• Asynchronous communication tends to make programs complicated.

More Comments on Solution

• Solution requires lots of messages
• Number of messages is often primary overhead factor
• Message aggregation: combining multiple messages in one

A Bit of Perspective

• Which one is easier, shared memory or message passing?
  – This has been the subject of a raging debate for the last ten years or so