Special Topics I — Vector and Parallel computing

1 Parallelism in the PC: pipelining

Goal: getting maximum throughput at a fixed clock speed.

1. Divide instruction into stages
2. Execute stages simultaneously on multiple instructions
3. Potential speedup = (clocks/instruction)/(pipeline stages).

Example instructions:

- Instruction fetch
- Instruction Decode/Register fetch
- Execute (this might be several stages for F.P. instruction)
- Memory access (if needed)
- Write back

Pipeline Hazards: these also cause trouble for vector and parallel machines.

- Structure hazards

  Not enough functional units to handle all concurrent operations.

- Data hazards

  Results from the WB stage of an earlier instruction are needed by a later instruction.

- Control hazards

  Branch invalidates later instructions in the pipeline.

2 Amdahl's law

$$\text{Speedup} = \frac{1}{1 - f_{Enhanced} + f_{Enhanced}/\text{Speedup}_{Enhanced}}$$

Lesson: significant fraction of code must be enhanced to gain significant speedups.

3 Vector processors

Why Vector?

- Allows deep pipelining without data hazards.
- Slows Instruction Fetch and Decode rate.
- Well known memory access pattern.
- Avoids control hazards.

4 Vector programming considerations

- Avoid short loops.
- Avoid conditionally executed statements in loops.
- Avoid data dependencies in loops.

  Example: sparse matrices.

  ```
    DO 100 I = 1,N
  100  A(K(I)) = A(K(I)) + C(M(I))
  ```

  Scatter gather operations can be used to make this efficient.

5 Parallel Architectures

- SIMD: single instruction multiple data.
- MIMD: multiple instructions, multiple data.
  - Centralized Shared Memory (Uniform Memory Access.)
    - single threaded processor
    - multi-threaded processor
  - Distributed Memory
    - Shared address space (Non-Uniform Memory Access).
      Easy(ish) to program; hard to implement correctly.
    - Message passing.
      Hard to program; easy to implement.

6 Programming strategies

- fine-grain parallelism
  - Similar to vectorization
  - Requires low-latency, shared address space.
- course-grain parallelism
  - Minimize data dependencies.
  - Hide latencies when data dependencies exist.

7 MPI tutorial

Reference: Bill Gropp, Rusty Lusk, and Anthony Skjellum, *Using MPI.*
Web: http://WWW.ERC.MsState.Edu/mpi/

Hello world:

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv )
{
  char message[20];
  int i,rank, size, type=99;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if(rank == 0) {
    strcpy(message, "Hello, world");
    for (i=1; i<size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, type,
               MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, type,
             MPI_COMM_WORLD, &status);

  printf( "Message from process =%d : %.13s\n",
          rank,message);
  MPI_Finalize();
}
```

Compiling and running:

```
setenv PATH /usr/local/lam62b/bin/mpicc:$PATH
```

```
mpicc hello.c -o hello
lamboot bhost
mpirun -c 3 [-c2c] hello
```

The bhost file:

```
a
e
```

8 Types of messages

- Synchronous: `MPI_Ssend()` returns only when a receive has completed.
- Asynchronous
  - Blocking: `MPI_Send(buf, ...)` returns when the contents of `buf` have been copied.
  - Nonblocking: `MPI_Isend(buf, ...)` returns immediately.
- Ready: `MPI_Rsend()` assumes a receive is already posted.

9 Avoiding deadlocks

```
#include <stdio.h>
#include "mpi.h"
#define MSGLEN 2048         /* length of message in elements */
#define TAG_A 100
#define TAG_B 200

main( int argc, char **argv )
{
  float message1 [MSGLEN], message2 [MSGLEN];
  int rank, i;
  MPI_Status status;         /* status of communication */

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  printf ( " Task %d initialized\n", rank );

  for ( i=0; i<MSGLEN; i++ )  {
    message1[i] = 100; message2[i] = -100;
  }

  if ( rank == 0 )  {
      MPI_Send ( message1, MSGLEN, MPI_FLOAT, 1, TAG_A,
                 MPI_COMM_WORLD );
      MPI_Recv ( message2, MSGLEN, MPI_FLOAT, 1, TAG_B,
                 MPI_COMM_WORLD, &status );
  }
  else if ( rank == 1)  {
      MPI_Send ( message1, MSGLEN, MPI_FLOAT, 0, TAG_B,
```

```
                MPI_COMM_WORLD );
      MPI_Recv ( message2, MSGLEN, MPI_FLOAT, 0, TAG_A,
                 MPI_COMM_WORLD, &status );
  }

  MPI_Finalize();
  return 0;
}
```

Solution:

```
  MPI_Isend ( message1, MSGLEN, MPI_FLOAT, dest, send_tag,
              MPI_COMM_WORLD, &request);
  MPI_Recv ( message2, MSGLEN, MPI_FLOAT, source, recv_tag,
             MPI_COMM_WORLD, &status );
  MPI_Wait ( &request, &status);
```

Other solutions:

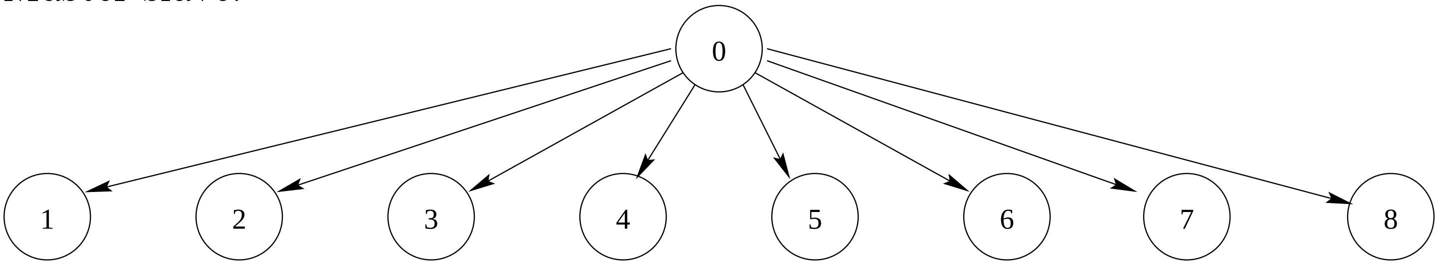
- Ordered send and receive
- Combined send and receive: `MPI_Sendrecv()`.
- Buffered sends: `MPI_Bsend()`.

10 Collective communication

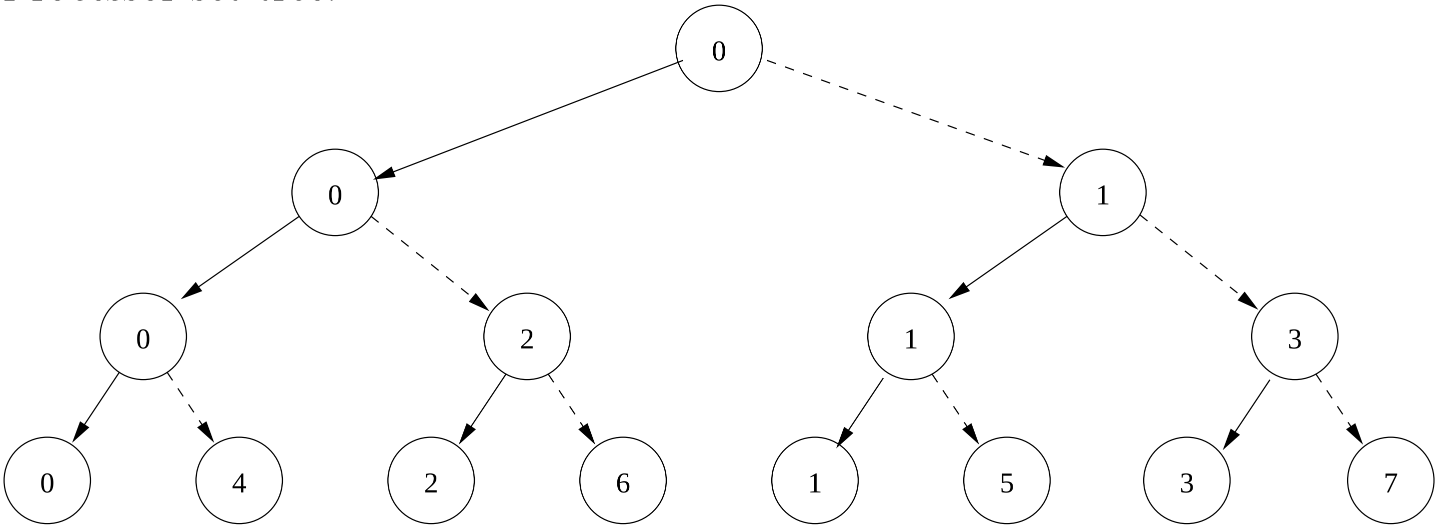
`MPI_Barrier(MPI_Comm comm );`
`MPI_Bcast(buffer, count, datatype, root, comm );`
Broadcasts data in `root`'s `buffer` to all processes.
`MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm);`
Performs reduction of data in `sendbuf` with `op` and stores result in `root`'s `recvbuf`.
There are also all-to-all versions of these.

11 Process management

Master-slave:



Processor set tree:



12 Multi-threading issues

- Modifing a global data item is non-trivial.
- Locks (`mutex`s) are needed to change shared data reliably.