Message Passing Interface

Basic functions

Ending the last lecture...

Parallele Programmierung                            1

Parallele Programmierung                            2

Remembering: a simplified LogP model

- Let us assume now that:
  - A parallel program is run by \( p \) distinct and equal processors, each one with its own private memory;
  - The time to communicate \( n \) Bytes between 2 processors is modeled as:
    \[ T_{comm}(n) = L + n/g \]
  1. \( L \) is a latency (in sec.),
  2. \( 1/g \) (in B/sec) is the throughput (\( g \) is the “gap” between the transmission of 2 Bytes).
- This model is homogeneous, static and symmetric.
  - All the processors are supposed to be equal,
  - Their number does not change during the computation,
  - A communication does not privilege the sender or the receiver.

Parallele Programmierung                            3

Parallele Programmierung                            4

System of linear equations

- Coming back to the LU original (non D&C) factorization...
  - for \( k = 0; k <= n-2; k++ \) {
    - for \( i = k+1; i <= n-1; i++ \)
      - \( M[i][k] = M[i][k] / M[k][k] \)
      - if \( (r == rank(k)) \) then
        - for \( (j = k+1; j <= n-1; j++) \)
          - \( M[j][i] = M[j][i] + M[j][k]*M[k][i] \)

Parallele Programmierung                            5

Parallele Programmierung                            6

The parallel algorithm (1)

- Writing in a SPMD way (all the processors run this same code):
  - for \( r = my_proc_rank() \)
    - \( p = number_of_procs() \)
      - for \( (k = 0; k <= n-2; k++) \) {
        - for \( (i = k+1; i <= n-1; i++ \)
          - \( M[i][k] = M[i][k] / M[k][k] \)
          - if \( (r == rank(k)) \) then
            - for \( (j = k+1; j <= n-1; j++) \)
              - \( M[j][i] = M[j][i] + M[j][k]*M[k][i] \)

Parallele Programmierung                            1

Parallele Programmierung                            2

The parallel algorithm (1)

- Writing in a SPMD way (all the processors run this same code):
  - for \( r = my_proc_rank() \)
    - \( p = number_of_procs() \)
      - for \( (k = 0; k <= n-2; k++) \) {
        - for \( (i = k+1; i <= n-1; i++ \)
          - \( M[i][k] = M[i][k] / M[k][k] \)
          - if \( (r == rank(k)) \) then
            - for \( (j = k+1; j <= n-1; j++) \)
              - \( M[j][i] = M[j][i] + M[j][k]*M[k][i] \)

Parallele Programmierung                            3

Parallele Programmierung                            4

Nicolas Maillard, Marcus Ritt

Parallele Programmierung                            5

Parallele Programmierung                            6

Nicolas Maillard, Marcus Ritt
The parallel algorithm (2)

- Using a “local index” \( i (0...n/p-1) \) for the columns:
  \[ r = \text{my_proc_rank} \]
  \[ p = \text{number_of_procs} \]
  \[ i = 0 \]
  \[ \text{for } (k = 0 ; k < n-2 ; k++) \{ \]
      \[ \text{if (r == rank(k)) then} \]
      \[ \text{for } (i = k+1 ; i <= n-1 ; i++) \]
      \[ M[i][j] = M[i][j] / M[k][i]; \]
      \[ i = i + 1 \]
  \[ \} \]
  \[ \text{”each processor updates its r-l rightmost columns (r = n/p)*} \]
  \[ \text{for (i = k+1 ; i <= n-1 ; i++)} \]
  \[ M[i][j] = M[i][j] + M[i][k]*M[k][j]; \]

Two more “implementation” details

- You need to broadcast the elements \( M[i][j] \) in the middle of the algorithm.
  - This means sending \( n/p \) coefficients to all the p-1 other processors.
  - Takes time \( (p-1) L + n^2 gp \).
  - Note: this is a worst case scenario – a broadcast can (should) be implemented better.
  - It could take something like \( (L + n^2 gp) \log(p) \).

- Probably, this broadcast needs to access contiguous elements in the local memory.
  - This means that the \( M[i][j] \) coefs. probably should be stored in column-major order (Fortran order).
  - Else (in C), you have to use an intermediate buffer.
  - This is typical of MPI + C programming.

So what is the “\( \text{rank()} \)” function?

- \( \text{rank}(k) = \text{rank of the processor that owns the column } k. \)
- There are many options:
  - Block mapping: let \( B = n/p \), then \( \text{rank}(k) = k / B \)
    - / is the euclidean division.
  - With this formula, the last processor gets a little bit more elements than the other.
  - Very simple to implement, and maximizes locality.
  - Cyclic \((\text{round-robin})\) mapping: \( \text{rank}(k) = k \% p \)
    - Simple to implement, minimizes locality
    - Good for load balancing.
  - Block cyclic: given a block size \( n/p \geq 2 \)
  - The best of two worlds.

Parallel complexity of LU

- Each processor performs (roughly):
  - \( n^3/2p \) divisions in the first phase (pivot computation)
    - Actually, they are products.
  - Broadcast: \((L + n^2 gp) \log(p)\).
    - for \((k = 0 ; k < n-2 ; k++) \{ \]
    - \( n^2 gp \) products
  - in the update phase.
    - for \((i = k+1 ; i < n-1 ; i++) \]
    - \( M[i][k] = M[i][k] / M[k][k]; \]
    - \( \text{broadcast } \) \text{ “} \]
    - for \((j = k+1 ; j < n-1 ; j++) \]
    - \( \text{granularity: roughly } \log(p) \)
  - \( (n^2 gp + n^3/2p) T = (L + n^2 gp) \log(p). \)
  - This is not that bad (compare to the matrix products).
    - But the parallel runtime is far from ideal.

Outline

- Introduction to the Message Passing Interface
  - Parallel programming model of MPI
    - “MPI for Dummies” the 6 basic instructions.
  - How to run a MPI program.
  - More advanced MPI:
    - Collective communication
    - Non-blocking communication
The MPI paradigm

Each process is identified by a unique tag.

Two main open source distributions:
- MPICH (www-unix.mcs.anl.gov/mpich)
- OpenMPI (LAM-MPI): www.open-mpi.org

A few references on MPI

http://www.mpi-forum.org, for the norm MPI.

International conference: EuroPVM-MPI (LNCS, Springer)

Books:
- Gropp, William et al., Using MPI, MIT Press.
- Gropp, William et al., Using MPI-2, MIT Press.

November 1997 MPI 1.2

Late 1998 Partial implementation of MPI 2.0

– 2000 Most of MPI-2 available

• Each one of the p processes run the same binary program
  – Single program, Multiple Data paradigm.
  – In “basic” MPI you launch the p processes at the start of the program, and all the p processes must run until the end.
  – Each process is identified by a unique rank (a number between 0 and p-1).
  – Based on the rank, each process can:
    – Execute tests (if..then) to run those parts of the program that are relevant;
    – (Advanced use): nothing prevents the processes to launch threads...
    – Send/receive messages to/from any other given process.
    – There are many types of possible messages.

What is a MPI message?

• Look at the MPI_send function:
  int MPI_Send(void*, int, MPI_Datatype, int, MPI_Comm).

• Typical call:
  MPI_Send(&work, 1, MPI_INT, dest, WORKTAG, MPI_COMM_WORLD);

• It sends the content of a buffer from the current process to the receiver dest process.

• The buffer is defined by the 3 first arguments:
  – Work (void*): pointer to the memory area where the data are found.
  – 1, MPI_INT: number and basic type of the data (almost size(0))

• A message is identified by a tag (see WORKTAG).
  – The tag must be the same in the Recv and Send.
  – The type is irrelevant to the matching algorithm between sender and receiver.

November 1994 MPI 1.0

June 1995 MPI 1.1

November 1997 MPI 1.2

November 1999 Partial implementation of MPI 2.0

2000 All major MPI distributions include MPI 2.0

• November 1992 First draft of MPI 1

• November 1993 Second draft of MPI 1.0

The MPI paradigm

November 1997 MPI 1.2

Late 1998 Partial implementation of MPI 2.0

– 2000 Most of MPI-2 available

• Each one of the p processes run the same binary program
  – Single program, Multiple Data paradigm.
  – In “basic” MPI you launch the p processes at the start of the program, and all the p processes must run until the end.
  – Each process is identified by a unique rank (a number between 0 and p-1).
  – Based on the rank, each process can:
    – Execute tests (if..then) to run those parts of the program that are relevant;
    – (Advanced use): nothing prevents the processes to launch threads...
    – Send/receive messages to/from any other given process.
    – There are many types of possible messages.

What is a MPI message?

• Look at the MPI_send function:
  int MPI_Send(void*, int, MPI_Datatype, int, MPI_Comm).

• Typical call:
  MPI_Send(&work, 1, MPI_INT, dest, WORKTAG, MPI_COMM_WORLD);

• It sends the content of a buffer from the current process to the receiver dest process.

• The buffer is defined by the 3 first arguments:
  – Work (void*): pointer to the memory area where the data are found.
  – 1, MPI_INT: number and basic type of the data (almost size(0))

• A message is identified by a tag (see WORKTAG).
  – The tag must be the same in the Recv and Send.
  – The type is irrelevant to the matching algorithm between sender and receiver.

2000 Most of MPI-2 available

• Each one of the p processes run the same binary program
  – Single program, Multiple Data paradigm.
  – In “basic” MPI you launch the p processes at the start of the program, and all the p processes must run until the end.
  – Each process is identified by a unique rank (a number between 0 and p-1).
  – Based on the rank, each process can:
    – Execute tests (if..then) to run those parts of the program that are relevant;
    – (Advanced use): nothing prevents the processes to launch threads...
    – Send/receive messages to/from any other given process.
    – There are many types of possible messages.

What is a MPI message?

• Look at the MPI_send function:
  int MPI_Send(void*, int, MPI_Datatype, int, MPI_Comm).

• Typical call:
  MPI_Send(&work, 1, MPI_INT, dest, WORKTAG, MPI_COMM_WORLD);

• It sends the content of a buffer from the current process to the receiver dest process.

• The buffer is defined by the 3 first arguments:
  – Work (void*): pointer to the memory area where the data are found.
  – 1, MPI_INT: number and basic type of the data (almost size(0))

• A message is identified by a tag (see WORKTAG).
  – The tag must be the same in the Recv and Send.
  – The type is irrelevant to the matching algorithm between sender and receiver.
Parallelisierung
Nicolas Maillard, Marcus Ritt

MPI_recv

- Profile of the call:
  ```c
  int MPI_Recv(void*, int, MPI_Datatype, int, int, MPI_Comm, MPI_Status*)
  ```
  
- Typical use:
  ```c
  MPI_Status* s;
  int d, TAG = 103;
  MPI_Recv(&d, 1, MPI_INT, source, TAG, MPI_COMM_WORLD, &s);
  ```

  *source* is the rank of the sender process, *TAG* is the tag of the message.

  This call is blocking.
  - When the process executes the next instruction, d contains the data that was expected.

Whole example

```c
void main() {
  int p, r, tag = 103;
  MPI_Status stat;
  double val;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &r);
  if (r==0) {
    printf("Process 0 sends a message to 1\n");
    val = 3.14;
    MPI_Send(val, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
  } else {
    printf("Process 1 receives a message from 0\n");
    MPI_Status stat;
    MPI_Recv(&val, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &stat);
    printf("I received the value \"%f\",\n");
  }
}
```
Fertig!

- See you tomorrow for a practical session.

---

**Blocking vs. Non-blocking communication**

- MPI_Recv(&x,...) is blocking.
- MPI_Send(&x,...) is "non-blocking"
  - But x is copied into an internal buffer.
  - Send is non-blocking... Until the internal buffer gets full!

- **Non-blocking variants:** MPI_Irecv() and MPI_Isend()
  - Same args as Recv/Send, with one extra of type MPI_Request.
  - The MPI_Request enables the testing of the completion of the non-blocking communication.

- **Explicitly bufferized versions of Send/recv:**
  MPI_Bsend, MPI_Brecv().

---

**Test & Wait non-blocking comm.**

- **MPI_Test(MPI_Request* req, int* flag, MPI_Status* stat):**
  - Sets 'flag' to 0 or 1, depending of 'req'
  - You have to test 'flag' afterwards (if (flag) ...)

- **MPI_Wait(MPI_Request* req, MPI_Status* stat):**
  - Waits until the completion of the non-blocking comm.

- **Key for High-Performance:**
  - **computation/communication overlap.**
    - Launch a non-blocking communication (e.g. recv).
    - (in a loop) run all you can run, without having received the data, and test regularly for the reception.
    - If the loop ends up, then block with a MPI_Wait.

---

Nicolas Maillard, Marcus Ritt
20/10/2008