Introduction to parallel computing

Distributed Memory Programming with MPI (4)

Zhiao Shi (Additions by Will French)
Advanced Computing Center for Research & Education
Vanderbilt University
Last time – MPI Collective communications

• **All** the processes in the communicator must call the same collective function.
• Point-to-point communications are matched on the basis of tags and communicators.
• Collective communications don’t use tags.
• They’re matched solely on the basis of the communicator and the order in which they’re called.
MPI Derived data types

- Used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- The idea is that if a function that sends data knows this information about a collection of data items, it can collect the items from memory before they are sent.
- Similarly, a function that receives data can distribute the items into their correct destinations in memory when they’re received.
Derived datatypes

• Formally, consists of a sequence of basic MPI data types together with a displacement for each of the data types.

• Trapezoidal Rule example:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
</tr>
<tr>
<td>n</td>
<td>48</td>
</tr>
</tbody>
</table>

\{(\text{MPI\_DOUBLE, 0)}, (\text{MPI\_DOUBLE, 16}), (\text{MPI\_INT, 24})\}.
MPI_Type_create_struct

- Builds a derived datatype that consists of individual elements that have different basic types.

```c
int MPI_Type_create_struct(
    int count, /* in */,
    int array_of_blocklengths[], /* in */,
    MPI_Aint array_of_displacements[], /* in */,
    MPI_Datatype array_of_types[], /* in */,
    MPI_Datatype* new_type_p /* out */);
```
MPI_Type_commit

- Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

```c
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```
MPI_Type_free

- When we’re finished with our new type, this frees any additional storage used.

```c
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```
Examples

• Posted on the SC290 Github repo under:
  • mpi/derived_data_types/
• There are also helpful examples for different datatypes on the LLNL MPI page:
  • https://computing.llnl.gov/tutorials/mpi/
    #Derived_Data_Types
Performance evaluation
"Time" on a computer system

- Wall clock time
- User time
- System time

![Diagram showing relationships between time concepts](image)

<table>
<thead>
<tr>
<th>User Time</th>
<th>System Time</th>
<th>CPU Time</th>
</tr>
</thead>
</table>
Performance Measuring with Timings: Wall clock

- Wall clock time (real time, elapsed time)
  - contains everything that CPU time contains but it also includes waiting for I/O, communication, and other jobs.
Performance Measuring with Timings: CPU

• CPU time
  • **User** Time: actual CPU time used in executing the process in user mode
  • **System** time: CPU time spend in kernel mode, executing system calls

• What about a process with multiple threads running on multiple CPUs?
Measuring time in Unix

- Unix command `time`

```
%time ls
........
0.820u 0.300s 0:01.32 84.8%  0+0k 0+0io 4049pf+0w
```

- 0.82 seconds user time
- 0.30 seconds system time
- 1.32 seconds wall time
- 84.8% of total was used by CPU to run the command
  - \((0.82 + 0.3)/1.32 = 0.848\)
Elapsed serial time

gettimeofday() — C/C++
– Resolution up to microseconds

• Returns time elapsed from some point in the past.
Elapsed serial time

#define GET_TIME(now) { \
     struct timeval t; \ 
     gettimeofday(&t, NULL); \ 
     now = t.tv_sec + t.tv_usec/1000000.0; \ 
 }

double start, finish;

... 
GET_TIME(start);
/* Code to be timed */
... 
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
Elapsed parallel time

double MPI_Wtime(void);

- Returns the number of seconds that have elapsed since some time in the past. Resolution depends on hardware, usually at least 10^{-6}.

start = MPI_Wtime();
/* Code to be timed */
...
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n" my_rank, finish-start);
MPI_Barrier

- Ensures that no process will return from calling it until every process in the communicator has started calling it.

```c
int MPI_Barrier(MPI_Comm comm /* in */);
```
MPI_Barrier

double local_start, local_finish, local_elapsed, elapsed;
...
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
...
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);

if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed);
Speedup

\[ S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)} \]
Efficiency

\[ E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)} \]
Execution Time Components

• Given a problem of size $n$ on $p$ processors let
  • Inherently sequential computations $\sigma(n)$
  • Potentially parallel computations $\Phi(n)$
  • Communication operations $\kappa(n, p)$

• Then:
  $$S(p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p} + \kappa(n, p)} \equiv \frac{T_s}{T_p}$$
Efficiency

- The efficiency of a parallel computation is defined as a ratio between the speedup factor and the number of processing elements in a parallel system:

\[
E = \frac{\text{Exec. time using one processor}}{p \times \text{Exec. time using } p \text{ processors}} = \frac{T_s}{p \times T_p} = \frac{S(p)}{p}
\]

- Efficiency is a measure of the fraction of time for which a processing element is usefully employed in a computation.
Amidahl’s Law

\[ S(p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n) + \kappa(n, p)} \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \phi(n) / p} \]

since the communication time must be non-trivial.

• Let \( f \) represent the inherently sequential portion of the computation; then

\[ f = \frac{\sigma(n)}{\sigma(n) + \phi(n)} \]
Amdahl’s Law (cont.)

\[ S(p) = \frac{p}{1 + (p-1)f} \]
Next time

• Parallel Libraries, Parallel Python, Parallel Matlab