Lecture 14

MPI, the Message Passing Interface
Application design
Announcements

• TA Evaluation next Tuesday
• CAPE!
Parallel Image Segmentation
Parallel image segmentation

• After dividing the image into strips, we implement a new algorithm with 2 steps
• Step 1: Local labeling
• Use the serial algorithm to segment each strip independently of the others
• Step 2: Merge the clusters so that we use a global labeling scheme
Serial Algorithm

• We maintain a table that allows us to link pixels that are far apart, taking $\lg(N)$ steps instead of $N$
## A simple example

<table>
<thead>
<tr>
<th>4 x 4 image</th>
<th>After phase 0</th>
<th>After phase 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>12 12 12 12</td>
<td>13 13 13 13</td>
</tr>
<tr>
<td>0 0 0 8</td>
<td>0 0 0 12</td>
<td>0 0 0 13</td>
</tr>
<tr>
<td>9 10 11 12</td>
<td>13 13 12 12</td>
<td>13 13 13 13</td>
</tr>
<tr>
<td>13 0 0 0</td>
<td>13 0 0 0</td>
<td>13 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>After phase 2</th>
<th>After phase 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 13 13 13</td>
<td>13 13 13 13</td>
</tr>
<tr>
<td>0 0 0 13</td>
<td>0 0 0 13</td>
</tr>
<tr>
<td>13 13 13 13</td>
<td>13 13 13 13</td>
</tr>
<tr>
<td>13 0 0 0</td>
<td>13 0 0 0</td>
</tr>
</tbody>
</table>
A glimpse at the code

```c
for p = 0:lg(N)+1
    for i=N-1:-1, j=N-1:-1, ij=i*N+j, where local[idx]
        lsave = local[ij]; // save previous label
        local[ij] = MAX(local[ij], all 8 neighbors);
    
    If label assigned to this pixel during the
    “follow the pointers” step is worse than some neighbor’s label,
    we’re converging to a local maximum;

    We correct this, to enable us to converge to the global max,
    by replacing our root pixel’s label with a better newly found one

    if (lsave< local[ij])
        if (local[lsave - 1] < local[ij])
            local[lsave - 1] = local[ij];
```
Local and global labeling

• Global labels are needed to properly handle clusters that cross thread boundaries
Resolving labels

• We put 2 pixels in the same cluster only if their corresponding pixels are within the threshold

• When we do this, we update the root of one pixel to point at the root of the other
Multiple steps required

- How many steps do we require, as a function of $P$?
Parallel control flow

- In each phase we double the size of the region we’ve labeled
- The amount of parallelism drops by half in each phase
- Only 1 thread updates the region of its “buddy”
MPI
The Message Passing Interface
MPI

• The API is delivered as a library called MPI “Message Passing Interface”
  ‣ MPI-1 has 125 routines
  ‣ Callable from C, C++, Fortran, etc.
  ‣ All major vendors support MPI
  ‣ Reference material: see http://www-cse.ucsd.edu/users/baden/Doc/mpi.html

• 7 minimal routines needed by nearly every MPI program
  ‣ start, end, and query MPI execution state (4)
  ‣ non-blocking point-to-point message passing (3)
Functionality we’ll cover today

- Point-to-point communication
- Communicators
- Data types
- Tags
- Non-blocking communication
- Message Filtering
A first MPI program : “hello world”

#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv);
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
Query functions

```c
main(int argc, char **argv ){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("I am process %d of %d.\n", rank, size);
    MPI_Finalize();
}
```
Message passing functionality

• MPI provides a rich collection of routines to move data between address spaces
  ‣ A single pair of communicating processes use \textit{point-to-point} communication
  ‣ With \textit{collective communication}, all the processors communicate together

• In point-to-point message passing we can filter messages in various ways

• This allows us to organize message passing activity conveniently
Point to point messages

• To send a message we need
  ‣ A destination
  ‣ A tag
  ‣ A message body (can be empty)
  ‣ A context (called a “communicator” in MPI)

• To receive a message we need similar information, including a receptacle to hold the incoming data
Communicators

- A communicator is a name-space (or a context) describing a set of processes that may communicate
- MPI defines a default communicator `MPI_COMM_WORLD` containing all processes
- MPI provides the means of generating uniquely named subsets (advanced MPI)
const int Tag=99;
int msg[2] = { rank, rank * rank};
if (rank == 0) {
    MPI_Status status;
    MPI_Recv(msg, 2, MPI_INT, 1, Tag, MPI_COMM_WORLD, &status);
}
else  MPI_Send(msg, 2, MPI_INT, 0, Tag, MPI_COMM_WORLD);
MPI Tags

• Tags enable processes to organize or screen messages

• Each sent message is accompanied by a user-defined integer tag:
  ‣ Receiving process can use this information to organize or filter messages
  ‣ MPI_ANY_TAG inhibits screening.

![Diagram](attachment:image.png)
MPI Datatypes

• MPI messages have a specified length
• The unit depends on the type of the data
• The length in bytes is sizeof(type) × # elements
• We don’t use the # bytes as the length
  ‣ Heterogeneous machines with different storage representations
  ‣ Performance
MPI Datatypes

• Because MPI is a library, we specify the type (and hence length) of an element.

• To this end MPI specifies a set of built-in types, corresponding to the primitive types of the language from which MPI is called.

• In C: MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, MPI_LONG, MPI_UNSIGNED, MPI_BYTE,…

• Also defined types, e.g. structs,
Message status

• An MPI_Status variable is a struct that contains the sending processor and the message tag
• This information is useful when we haven’t filtered the messages
• We may also access the length of the received message (may be shorter than the message buffer)

```c
MPI_Recv( message, count,
          TYPE, MPI_ANY_SOURCE,
          MPI_ANY_TAG, COMMUNICATOR,
          &status);

MPI_Get_count(&status, TYPE, &recv_count);

status.MPI_SOURCE      status.MPI_TAG
```
Asynchronous, non-blocking communication in MPI

- An extra request argument is required

  ```c
  MPI_Request request;
  MPI_Irecv(buf, count, type, source, tag, comm, &request)
  ```

- We use the request variable to specify which message we are synchronizing in `MPI_Wait()`

  ```c
  MPI_Wait(&request, &status)
  ```

- Making above 3 calls in succession is equivalent to

  ```c
  MPI_Recv(buf, count, type, source, tag, comm, &status)
  ```
Correctness and fairness

• When there are multiple outstanding iRecvs, MPI doesn’t say how incoming messages are matched…
• Or even if the process is fair

MPI_Request req1, req2;
MPI_Status status;
MPI_Irecv(buff, len, CHAR, ANY_NODE, TYPE, WORLD,&req1);
MPI_Irecv(buff2,len, CHAR, ANY_NODE, TYPE, WORLD,&req2);
MPI_Send(buff, len, CHAR, nextnode, TYPE, WORLD);
MPI_Send(buff, len, CHAR, prevnode, TYPE, WORLD);
MPI_Wait(&req1, &status);
MPI_Wait(&req2, &status);
Buffering

- If there is not a pending receive, then an incoming message is placed in an anonymous system buffer.
- When the receive gets posted, the message is moved into the user specified buffer.
- Double copying reduces communication performance.
- Non-blocking communication can help avoid this problem.
Rendezvous

• When a long message is to be sent, MPI first checks if the recipient has sufficient storage to receive the message

• If so, then it sends the message. This is called a *rendezvous* implementation. What are the advantages and disadvantages?
Eager limits

• In an *eager* implementation, we just send the message
• In practice, MPI implementations switch between the two modes
• The *eager limit* is the longest message that can be sent in eager mode
• Maximum value on IBM SP systems is 256K
• See M. Banikazemi et al., IEEE TPDS, 2001, “MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems”
• What about longer messages?
Sends that block

- Consider the following example of an “unsafe” program
- It may deadlock if there isn’t enough storage to receive the incoming message(s)

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(x, 1)</td>
<td>Send(x, 0)</td>
</tr>
<tr>
<td>Recv(y)</td>
<td>Recv(y)</td>
</tr>
</tbody>
</table>
Avoiding an unsafe program

• The system has pre-allocated storage for the incoming messages so there’s no possibility of running out of storage

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRecv(x)</td>
<td>IRecv(x)</td>
</tr>
<tr>
<td>Send(y, 1)</td>
<td>Send(y, 0)</td>
</tr>
</tbody>
</table>