Parallel and Distributed Computing: MPI part 2

Erik Saule
esaule@uncc.edu

Parallel and Distributed Computing

Learning Outcomes

After this lecture you will be able to

- use point to point communication primitives
- explain the difference between blocking and non-blocking communications
- identify deadlocks and performance issues with point to point communications

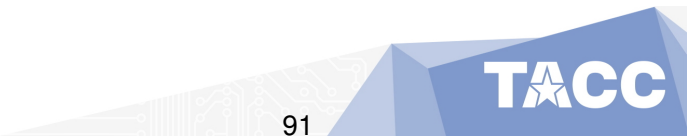
The assignment will make you

- Use point to point communication to write a dynamic scheduler
- use point to point communication to solve a problem with regular communication pattern

Outline

MPI point-to-point mechanism

- Two-sided communication
- Matched send and receive calls
- One process sends to a specific other process
- Other process does a specific receive.



Ping-pong

A sends to B, B sends back to A

Process A executes the code

```
MPI_Send( /* to: */ B ..... );
MPI_Recv( /* from: */ B ... );
```

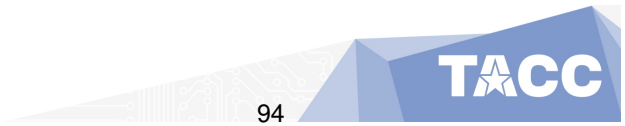
Process B executes

```
MPI_Recv( /* from: */ A ... );
MPI_Send( /* to: */ A ..... );
```

Ping-pong in MPI

Remember SPMD:

```
if ( /* I am process A */ ) {
  MPI_Send( /* to: */ B ..... );
  MPI_Recv( /* from: */ B ... );
} else if ( /* I am process B */ ) {
  MPI_Recv( /* from: */ A ... );
  MPI_Send( /* to: */ A ..... );
}
```

This completely breaks process symmetry :(

Data Communication

- Data communication in MPI is like email exchange
  - One process sends a copy of the data to another process (or a group of processes), and the other process receives it
- Communication requires the following information:
  - Sender has to know:
    - Whom to send the data to (receiver's process rank)
    - What kind of data to send (100 integers or 200 characters, etc)
    - A user-defined "tag" for the message (think of it as an email subject; allows the receiver to understand what type of data is being received)
  - Receiver "might" have to know:
    - Who is sending the data (OK if the receiver does not know; in this case sender rank will be **`MPI_ANY_SOURCE`**, meaning anyone can send)
    - What kind of data is being received (partial information is OK: I might receive *up to* 1000 integers)
    - What the user-defined "tag" of the message is (OK if the receiver does not know; in this case tag will be **`MPI_ANY_TAG`**)

MPI Basic (Blocking) Send

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
```

- The message buffer is described by (`buf`, `count`, `datatype`).
- The target process is specified by `dest` and `comm`.
  - `dest` is the rank of the target process in the communicator specified by `comm`.
- `tag` is a user-defined "type" for the message
- When this function returns, the data has been delivered to the system and the buffer can be reused.
  - The message may not have been received by the target process.

MPI Basic (Blocking) Receive

```
MPI_RECV(buf, count, datatype, source, tag, comm, status)
```

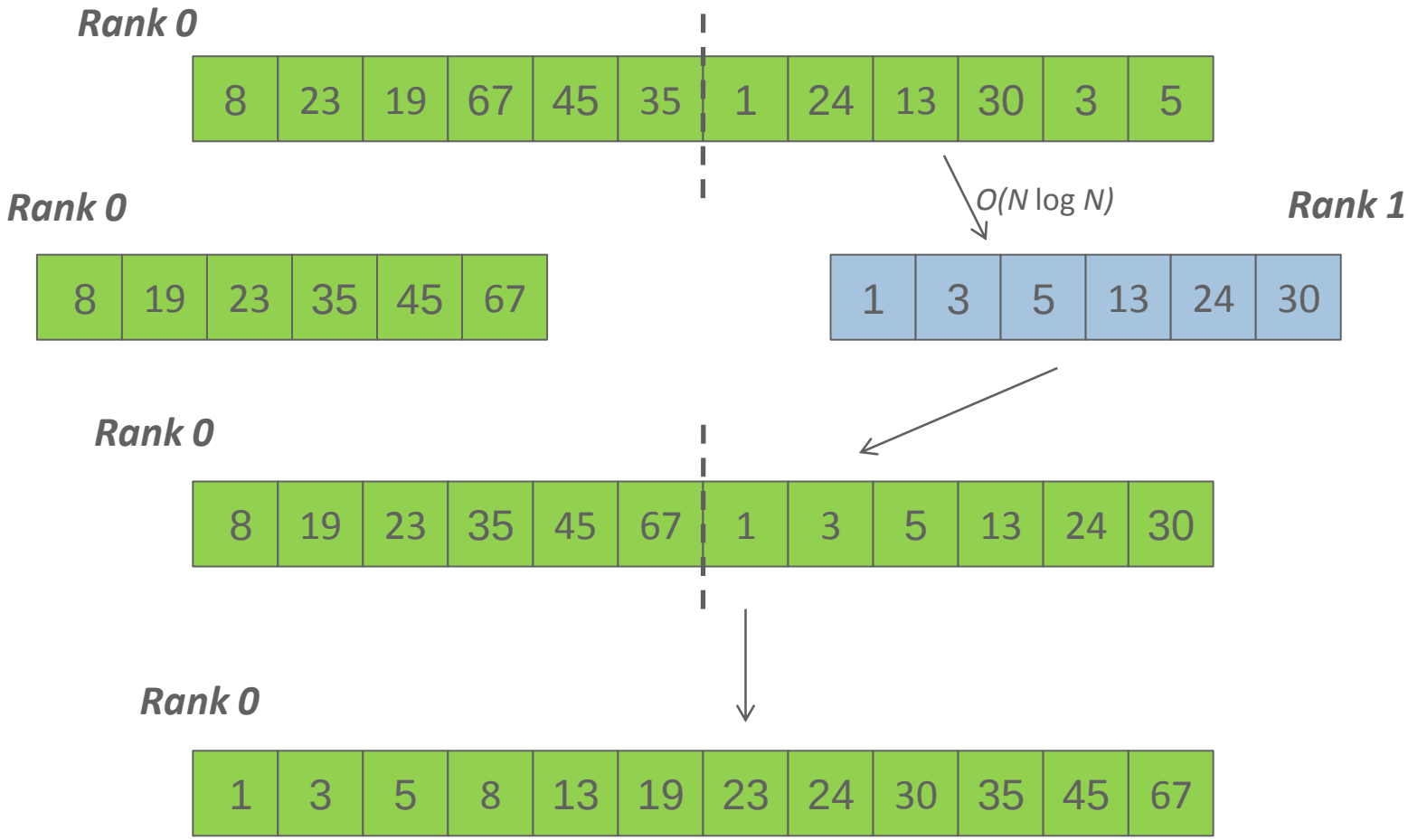
- Waits until a matching (on **source**, **tag**, **comm**) message is received from the system, and the buffer can be used.
- **source** is rank in communicator **comm**, or **MPI_ANY_SOURCE**.
- Receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.
- **status** contains further information:
  - Who sent the message (can be used if you used **MPI_ANY_SOURCE**)
  - How much data was actually received
  - What tag was used with the message (can be used if you used **MPI_ANY_TAG**)
  - **MPI_STATUS_IGNORE** can be used if we don't need any additional information

Simple Communication in MPI

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

int main(int argc, char ** argv)
{
    int rank, data[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
}
```

Pavan Balaji and Torsten Hoefler, PPoPP, Shenzhen, China (02/24/2013)

Parallel Sort using MPI Send/Recv



Parallel Sort using MPI Send/Recv (contd.)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
    int rank;
    int a[1000], b[500];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
        sort(a, 500);
        MPI_Recv(b, 500, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);

        /* Serial: Merge array b and sorted part of array a */
    }
    else if (rank == 1) {
        MPI_Recv(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        sort(b, 500);
        MPI_Send(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize(); return 0;
}
```

Because recv takes wildcards

Status Object

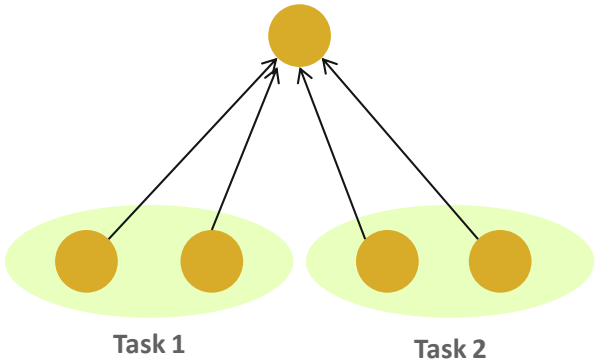
- The status object is used after completion of a receive to find the actual length, source, and tag of a message
- Status object is MPI-defined type and provides information about:
  - The source process for the message (`status.MPI_SOURCE`)
  - The message tag (`status.MPI_TAG`)
  - Error status (`status.MPI_ERROR`)
- The number of elements received is given by:

```
MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

`status`	return status of receive operation (status)
`datatype`	datatype of each receive buffer element (handle)
`count`	number of received elements (integer)(OUT)

Usage example of status

Using the “status” field



- Each “worker process” computes some task (maximum 100 elements) and sends it to the “master” process together with its group number: the “tag” field can be used to represent the task
  - Data count is not fixed (maximum 100 elements)
  - Order in which workers send output to master is not fixed (different workers = different src ranks, and different tasks = different tags)

Usage example of status

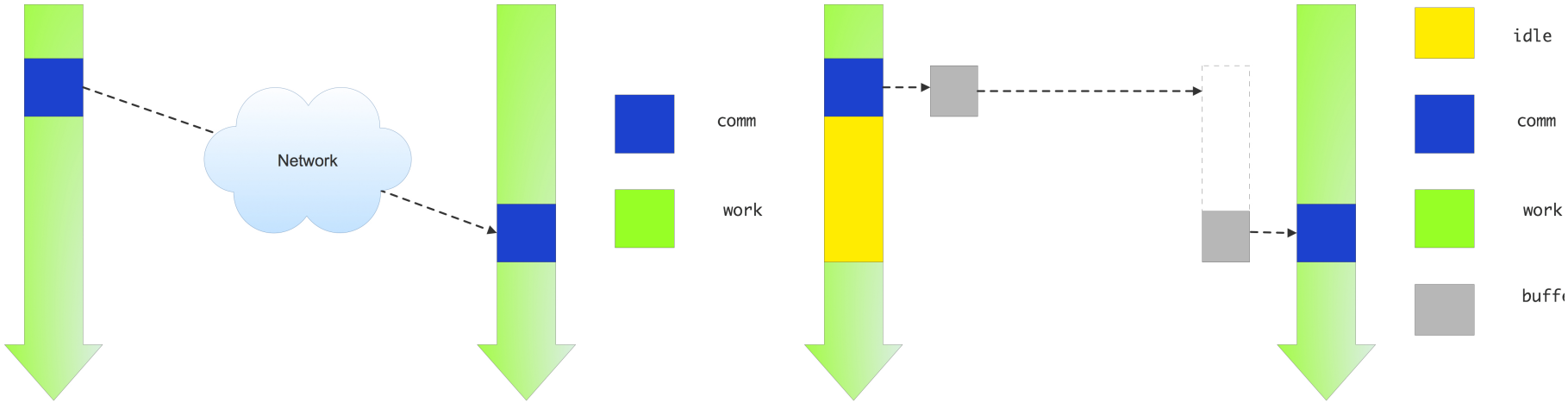
Using the "status" field (contd.)

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

int main(int argc, char ** argv)
{
    [...snip...]

    if (rank != 0)
        MPI_Send(data, rand() % 100, MPI_INT, 0, group_id,
                 MPI_COMM_WORLD);
    else {
        for (i = 0; i < size - 1 ; i++) {
            MPI_Recv(data, 100, MPI_INT, MPI_ANY_SOURCE,
                     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_INT, &count);
            printf("worker ID: %d; task ID: %d; count: %d\n",
                   status.source, status.tag, count);
        }
    }

    [...snip...]
}
```

Blocking send/recv

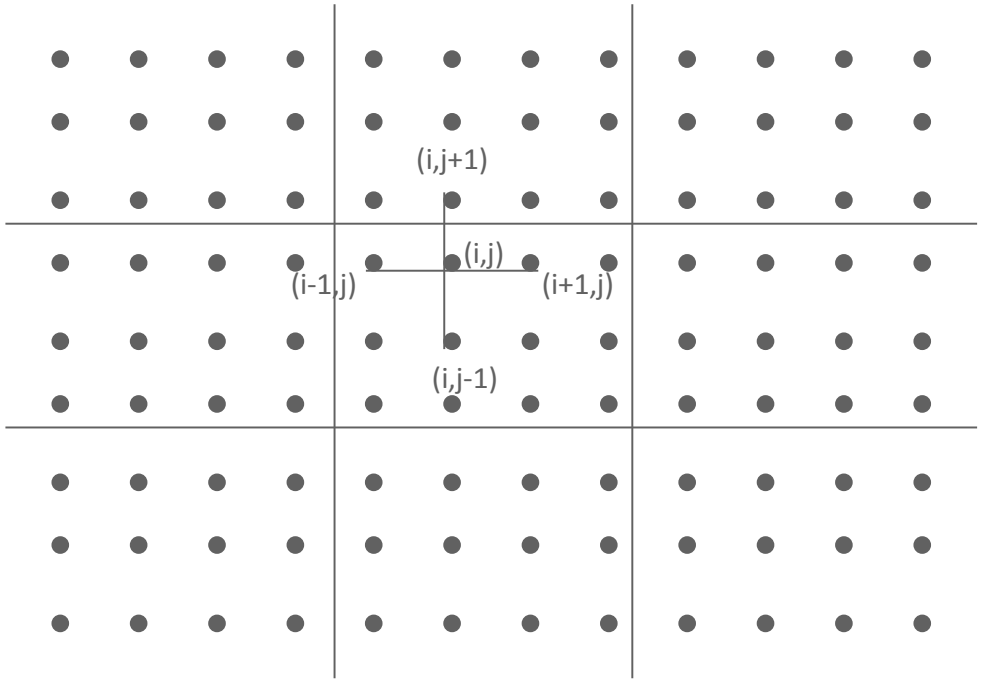
`MPI_Send` and `MPI_Recv` are *blocking* operations:

- The process waits ('blocks') until the operation is concluded.
- A send can not complete until the receive executes.



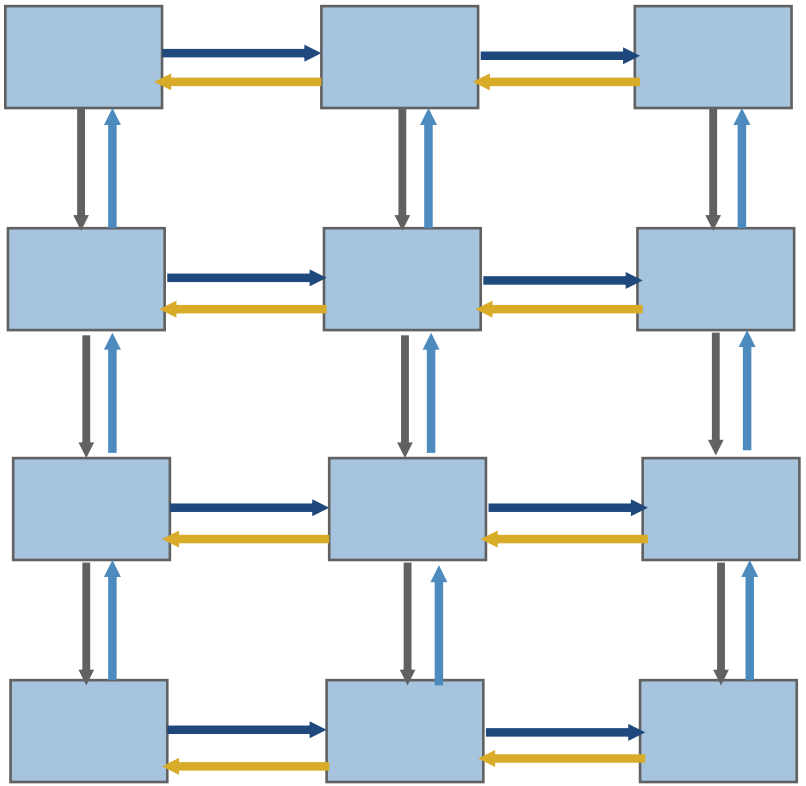
Ideal vs actual send/recv behaviour.

2D Poisson Problem



Mesh Exchange

- Exchange data on a mesh



Pavan Balaji and Torsten Hoefler, PPoPP, Shenzhen, China (02/24/2013)

Sample Code

```
Do i=1, n_neighbors
  Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
                                comm, ierr)
Enddo

Do i=1, n_neighbors
  Call MPI_Recv(edge, len, MPI_REAL, nbr(i), tag,
                               comm, status, ierr)
Enddo
```

- What is wrong with this code?

Deadlock

Every body sends, no one receives

A simple solution

- If I have a right neighbor, send data right
- If I have a left neighbor, get data from left

- If I have a left neighbor, send data left
- If I have a right neighbor, get data from right

- If I have a up neighbor, send data up
- If I have a down neighbor, get data from down

- If I have a down neighbor, send data down
- If I have a up neighbor, get data from up

What's wrong with that?

A simple solution

- If I have a right neighbor, send data right
- If I have a left neighbor, get data from left

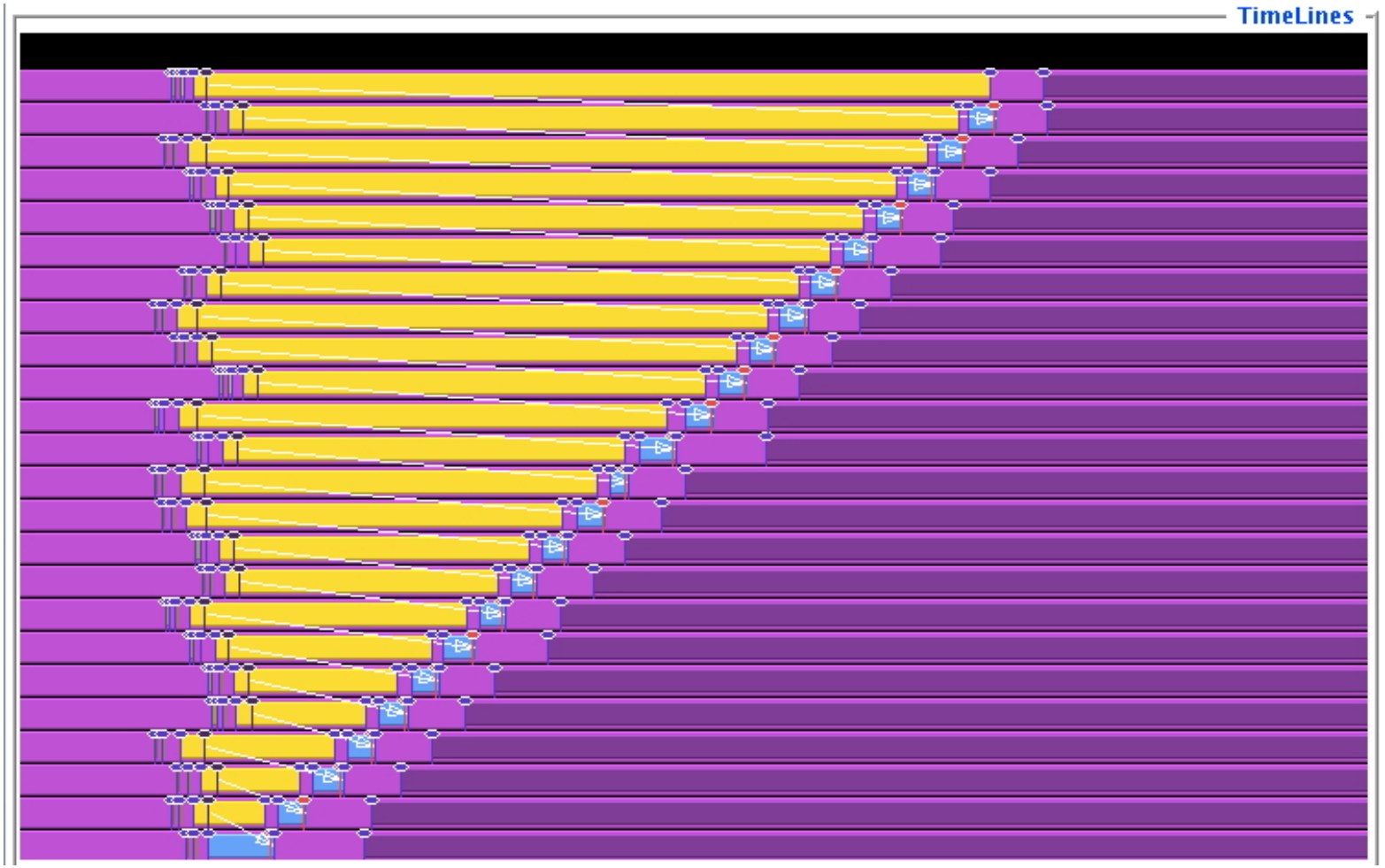
- If I have a left neighbor, send data left
- If I have a right neighbor, get data from right

- If I have a up neighbor, send data up
- If I have a down neighbor, get data from down

- If I have a down neighbor, send data down
- If I have a up neighbor, get data from up

What's wrong with that?
Sequentializes communications

TAU trace: serialization



The real problem

Communications in the previous format are not independent from the other.
The solution:
decompose all the communications in a set of non interacting communications.
The name:

The real problem

Communications in the previous format are not independent from the other.
The solution:
decompose all the communications in a set of non interacting communications.
The name: decomposing a graph in a set of matching. That is a classic graph theoretical problem.
Ex: odd/even communications

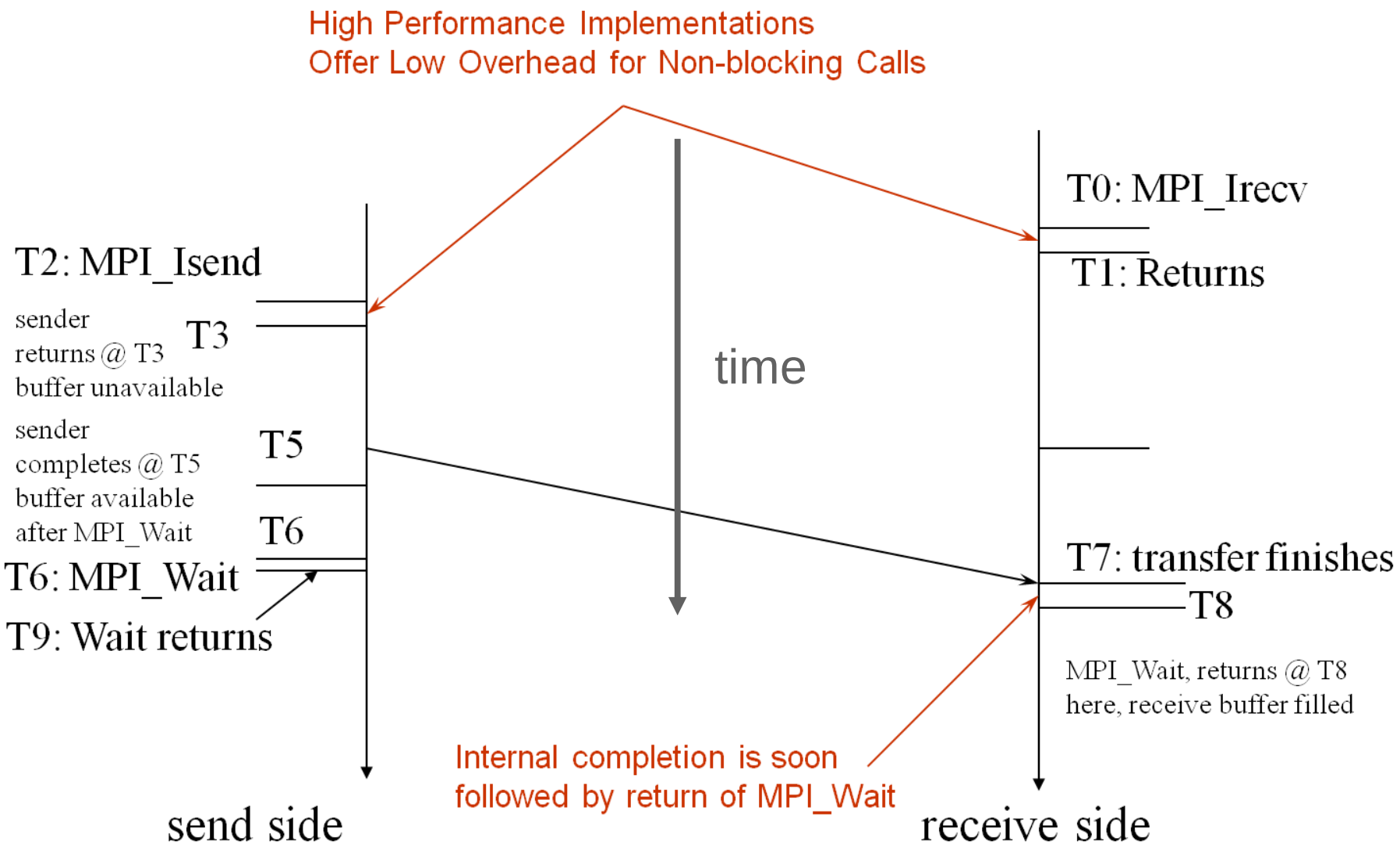
Outline

A Non-Blocking communication example

```
int main(int argc, char ** argv)
{
    [...snip...]
    if (rank == 0) {
        for (i=0; i< 100; i++) {
            /* Compute each data element and send it out */
            data[i] = compute(i);
            MPI_ISend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                      &request[i]);
        }
        MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
    }
    else {
        for (i = 0; i < 100; i++)
            MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
    }
    [...snip...]
}
```

Pavan Balaji and Torsten Hoefler, PPoPP, Shenzhen, China (02/24/2013)

Non-Blocking Send-Receive Diagram



Pavan Balaji and Torsten Hoefler, PPoPP, Shenzhen, China (02/24/2013)

Non-Blocking Communication

- Non-blocking (asynchronous) operations return (immediately) "request handles" that can be waited on and queried
  - `MPI_ISEND(start, count, datatype, dest, tag, comm, request)`
  - `MPI_IRECV(start, count, datatype, src, tag, comm, request)`
  - `MPI_WAIT(request, status)`
- Non-blocking operations allow overlapping computation and communication
- One can also test without waiting using **MPI_TEST**
  - **MPI_TEST(request, flag, status)**
- Anywhere you use **MPI_SEND** or **MPI_RECV**, you can use the pair of **MPI_ISEND/MPI_WAIT** or **MPI_IRECV/MPI_WAIT**
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead of barriers

Multiple Completions

- It is sometimes desirable to wait on multiple requests:
  - `MPI_Waitall(count, array_of_requests, array_of_statuses)`
  - `MPI_Waitany(count, array_of_requests, &index, &status)`
  - `MPI_Waitsome(count, array_of_requests, array_of_indices, array_of_statuses)`
- There are corresponding versions of `test` for each of these

A magic bullet?

Data from the sender can not be reused until the communication has been waited upon.
Some MPI implementation may not support an arbitrary number of pending asynchronous communications.
All communication must be explictely waited upon.
Usually a bit slower than blocking communications.

Outline

More sends and receive

- `MPI_Bsend`, `MPI_Ibsend`: buffered send
- `MPI_Ssend`, `MPI_Issend`: synchronous send
- `MPI_Rsend`, `MPI_Irsend`: ready send
- Persistent communication: repeated instance of same proc/data description.

too obscure to go into.

Other things

One sided communication. (Put and get operations)
MPI IO
MPI data types
MPI process spawning

Outline

Master-Worker

Dynamic scheduling of work using MPI. (on numerical integration)
One process, the Master, directs the other ones.
The other processes, the Workers, will receive some work request, do the work, and send an acknowledgement/result back.
The Master sends some work to all the Workers and awaits for a response, once it gets a response, it sends more work to the worker that replied.
Hint: use asynchronous communication.

Heat Equation

Implement a basic heat equation diffusion in 2D.
$Heat^{k+1}[i][j] = 1/5(Heat^{k+1}[i-1][j] + Heat^{k+1}[i+1][j] + Heat^{k+1}[i][j] + Heat^{k+1}[i][j-1] + Heat^{k+1}[i][j+1])$.
Do a block partitioning, like for matvec.
I suggest using asynchronous operation.

Outline

External

Books:

- Using MPI, 3rd edition. William Gropp, Ewing Lusk and Anthony Skjellum. MIT Press. Available through the library at https://librarylink.uncc.edu/login?url=http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6981847
- Using Advanced MPI. William Gropp, Torsten Hoefler, Rajeev Thakur and Ewing Lusk. MIT Press. Available through the library at https://librarylink.uncc.edu/login?url=http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6981848

MPI implementations:

- MPICH https://www.mpich.org
- OpenMPI https://www.open-mpi.org/

API Documentation:

- MPICH man pages https://www.mpich.org/static/docs/v3.2/www3/index.htm
- OpenMPI documentation https://www.open-mpi.org/doc/v3.0/

Slides I used:

- Tutorial on MPI programming. Victor Eijkhout. https://bitbucket.org/VictorEijkhout/parallel-computing-book/raw/e11748c8d8ae874ed645566ba0e82aa787ecf959/EijkhoutMPIlecture.pdf
- MPI for Dummies. Pavan Balaki, Torsten Hoefler. https://htor.inf.ethz.ch/teaching/mpi_tutorials/ppopp13/2013-02-24-ppopp-mpi-basic.pdf

Tutorial:

- part one of Parallel Programming in MPI and OpenMP. Victor Eijkhout. Draft at https://bitbucket.org/VictorEijkhout/parallel-computing-book/raw/e11748c8d8ae874ed645566ba0e82aa787ecf959/EijkhoutParComp.pdf