Introduction to MPI

Communication

Plays an important role in parallel programming models

- Sockets
  - standard model for network communication
  - slow interface, no standard interface for common programming tasks
- Shared memory
  - implicit communication via common memory blocks *e.g.,* pthreads, OpenMP
- Message Passing
  - explicit communication between processes

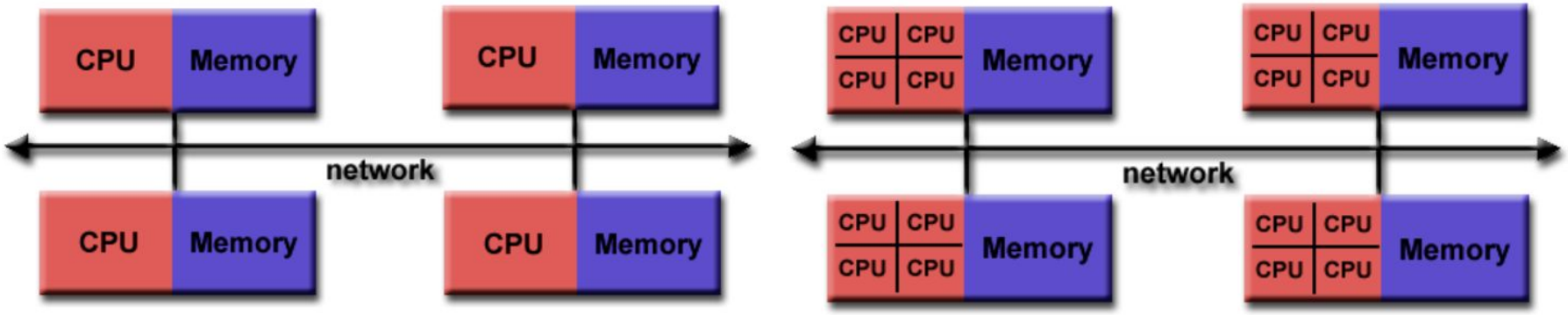
MPI (Message Passing Interface)

- Addresses the message-passing parallel programming model
  - moving data from the address space of one process to that of another process through cooperative operations on each process

- It is a specification on providing message passing support for parallel/distributed applications

- It is standardized, has good scalability, portable and flexible

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.



- The programming model is a distributed memory model. However, regardless of the underlying physical architecture of the machine today, MPI runs on virtually any hardware platform:
  - distributed memory
  - shared memory
  - hybrid

- Relatively more complex than shared memory programming model, has communication overhead.

Implementations and Compilers

- Although the MPI programming interface has been standardized, actual library implementations differ

MPI Library	Where?	Compilers
MVAPICH	Linux clusters	TOSS 2: GNU, Intel, PGI TOSS 3: GNU, Intel, PGI, Clang
Open MPI	Linux clusters	TOSS 2: GNU, Intel, PGI TOSS 3: GNU, Intel, PGI, Clang
Intel MPI	Linux clusters	Intel, GNU
IBM BG/Q MPI	BG/Q clusters	IBM, GNU
IBM Spectrum MPI	Coral Early Access and Sierra clusters	IBM, GNU, PGI, Clang

Threads and Processes

- Threads and processes are both independent sequences of execution.

- Threads (of a given process) run in a shared memory space. They share the virtual address space and system resources of a process. In addition, each thread maintains a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled.

- Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

Core Functions

- int MPI_Init(int* argc, char*** argv)
  - initialize MPI
- int MPI_Finalize()
  - ending an MPI application, close all resources
- int MPI_Comm_size(MPI_Comm comm, int* size)
  - number of processes in the group of comm
- int MPI_Comm_rank(MPI_Comm comm, int* rank)
  - rank of the calling process in group of comm
- int MPI_Comm_abort(MPI_Comm comm)
  - terminate MPI communication connection with an error flag

Core Functions

- int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - sends data to another process
- int MPI_Receive(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - receives data from another process

MPI_Init

int MPI_Init(int* argc, char* argv)**

- argc and argv can be used to pass all the command line arguments to all processes
- must be called before any other MPI functions
- initializes the MPI runtime

MPI_Finalize

int MPI_Finalize()

- must be called after all MPI functions
- frees all MPI communication resources allocated to the application

MPI_Comm_size

int MPI_Comm_size(MPI_Comm comm, int* size)

- returns the total number of MPI processes running in the communicator.

- the communicator MPI_Comm gives the scope of the request. MPI_COMM_WORLD is the predefined communicator and its scope is the all processes of the application.

MPI_Comm_rank

int MPI_Comm_rank(MPI_Comm comm, int* rank)

- assigns a unique id to each process in a communicator.

MPI_Comm_abort

int MPI_Comm_abort(MPI_Comm comm)

- terminates the communication connections and returns with an error flag of value 1

MPI_Send

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- buf: starting address of send buffer (pointer)
- count: Number of elements in send buffer (nonnegative integer)
- datatype: Datatype of each send buffer element (MPI_Datatype)
- dest: Rank of destination (integer)
- tag: Message tag (integer)
- comm: Communicator (handle)

MPI_Receive

int MPI_Receive(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)

- buf: starting address of receive buffer (pointer)
- count: Maximum number of elements in receive buffer (integer)
- datatype: Datatype of each receive buffer element (MPI_Datatype)
- source: Rank of source (integer)
- tag: Message tag (integer)
- comm: Communicator (handle)
- status: Status object (Status)

Blocking vs. Non-blocking calls

MPI_Send() and MPI_Recv() are blocking calls:

- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task.

- A blocking receive only "returns" after the data has arrived and is ready for use by the program.

Blocking vs. Non-blocking calls

- Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.

- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

Predefined MPI Types

Data types for C language bindings	
MPI_CHAR	8-bit character
MPI_SIGNED_CHAR	8-bit signed character
MPI_UNSIGNED_CHAR	8-bit unsigned character
MPI_WCHAR	16-bit unsigned character
MPI_SHORT	16-bit integer
MPI_INT	32-bit integer
MPI_LONG	32-bit integer
MPI_LONG_LONG	64-bit integer
MPI_LONG_LONG_INT	64-bit integer
MPI_UNSIGNED	32-bit unsigned integer
MPI_UNSIGNED_LONG	32-bit unsigned integer
MPI_UNSIGNED_LONG_LONG	64-bit unsigned integer
MPI_UNSIGNED_SHORT	16-bit unsigned integer
MPI_FLOAT	32-bit floating point
MPI_DOUBLE	64-bit floating point
MPI_LONG_DOUBLE	64-bit floating point

```cpp
#include <iostream>
#include "mpi.h"

using namespace std;

// module load mpi/openmpi_2.0.3_gcc
// compile with: mpicxx ./helloworld.cpp -o helloworld
// Run on multiple compute nodes on CCV: srun -N2 -n16 --mpi=pmi2 ./helloworld
// Submit a batch job on CCV using shell script: sbatch ./mpiBatch.sh
// Run multiple processes on local machine (single node): mpiexec -n 16 ./helloworld

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

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

  char myName[MPI_MAX_PROCESSOR_NAME];
  int resultLength;
  MPI_Get_processor_name(myName, &resultLength);

  cout << "Hello World! Process: " << rank << " out of " << size << " running on: " << myName << endl;

  MPI_Finalize();

  return 0;

}
```

Program Output:

```
Hello World! Process: 15 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 5 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 9 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 13 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 1 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 2 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 10 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 11 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 14 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 3 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 7 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 12 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 8 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 0 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 4 out of 16 running on: login004.oscar.ccv.brown.edu
Hello World! Process: 6 out of 16 running on: login004.oscar.ccv.brown.edu
```

```
if (rank == 0) {

  vector<double> p0_list;
  p0_list.push_back(1.0);
  p0_list.push_back(1.5);
  p0_list.push_back(2.0);
  p0_list.push_back(5.5);
  p0_list.push_back(11.2);

  /*
  * int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  * buf = reference to data to be sent
  * count = number of items in the message
  * datatype = implies size of items in the message
  * dest = rank of process to receive the message
  * tag = programmer specified identifier (used to organize messages)
  */
  MPI_Send(&p0_list[0], 5, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);

  cout << "Message sent by process " << rank << endl;

  p0_list.resize(5);
  MPI_Recv(&p0_list[0], 5, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);

  cout << "Process " << rank << " received: ";
  for(int i = 0; i < p0_list.size(); ++i){
    cout << p0_list[i] << " ";
  }
  cout << endl;
```

```
} else {

  if (rank == 1){

    vector<double> p1_list;
    p1_list.resize(5);

    /*
    * int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag,
    * MPI_Comm comm, MPI_Status *status)
    * buf = reference to data to be sent
    * count = number of items in the message
    * datatype = implies size of items in the message
    * source = rank of process from which to receive the message
    * tag = programmer specified identifier (used to organize messages)
    * status = pointer to structure storing misc. information about the message:
    *   status.MPI_SOURCE (rank of sender)
    *   status.MPI_TAG (tag of message)
    *   status.MPI_ERROR (error code)
    */
    MPI_Recv(&p1_list[0], 5, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

    cout << "Process " << rank << " received: ";
    for(int i = 0; i < p1_list.size(); ++i){
      cout << p1_list[i] << " ";
      p1_list[i] = p1_list[i] * 2.0;
    }
    cout << endl;

    MPI_Send(&p1_list[0], 5, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

    cout << "Message sent by process " << rank << endl;
```

```
    } else {

      int sleeptime = 2;
      sleep(sleeptime);
      cout << "Process " << rank << " slept for " << sleeptime << " secs and exited." << endl;

    }

  }

  MPI_Finalize();

  return 0;
}
```

Program Output:

```
Message sent by process 0
Process 0 received: 2 3 4 11 22.4
Process 1 received: 1 1.5 2 5.5 11.2
Message sent by process 1
Process 2 slept for 2 secs and exited.
Process 3 slept for 2 secs and exited.
```

Safe vs. Unsafe MPI

- MPI_Send and MPI_Recv are blocking calls, i.e., the MPI standard implies that:
  - a process will block on MPI_Send until a remote MPI_Recv copies data from the send buffer, i.e., until the send buffer is clear
  - a process will block on MPI_Recv until a remote MPI_Send supplies the data, i.e., until the receive buffer is full

- Some MPI implementations provide additional buffering of data communications, so MPI_Send might return even without matching MPI_Recv, if its buffer is copied into another buffer

Safe vs. Unsafe MPI

- Clearly, MPI_Recv still needs data, so it will always block without a matching MPI_Send

- A safe MPI program is one that does not rely on a buffered underlying implementation in order to function correctly

```
if (rank == 0)
{
    MPI_Recv(from 1);
    MPI_Send(to 1);
}
else if (rank == 1)
{
    MPI_Recv(from 0);
    MPI_Send(to 0);
}
```

deadlock
receives executed on both processes before matching send; buffering irrelevant

```
if (rank == 0)
{
    MPI_Send(to 1);
    MPI_Recv(from 1);
}
else if (rank == 1)
{
    MPI_Send(to 0);
    MPI_Recv(from 0);
}
```

unsafe
may work if buffers are larger than messages; if unbuffered, or messages are sufficiently large this code will fail

```
if (rank == 0)
{
    MPI_Send(to 1);
    MPI_Recv(from 1);
}
else if (rank == 1)
{
    MPI_Recv(from 0);
    MPI_Send(to 0);
}
```

safe
sends are paired with corresponding receives between processes; no assumptions made about buffering

Ensuring a program is safe

- MPI_Ssend is synchronous mode send. It sends a message and blocks until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.
  - your code is still not safe against deadlock

- Strategies for avoiding deadlock
  - pay attention to order of send/receive in communication operations
  - use MPI_Sendrecv
  - use non-blocking communication: the calls will not block, but it’s easy to introduce bugs if not careful

MPI_Sendrecv

int MPI_Sendrecv (void* sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void* recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status* status)

- Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.

MPI_Sendrecv

- Combined send and receive operation
  - allows MPI to deal with order of calls to reduce potential for deadlock
  - does not imply pairwise send/receive

- Can be satisfied by regular sends/receives on other processes
  - does not remove all potential for deadlock

```
...
  if (rank == 0)
    {
      int in, out = 5;

      MPI_Sendrecv(&out, 1, MPI_INT, 1, 0,
                   &in, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);

      printf("P0␣Received:␣%d\n",in);
    }
  else
    {
      int in, out = 25;

      MPI_Recv(&in, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
      MPI_Send(&out, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

      printf("P1␣Received:␣%d\n", in);
    }
...
```

```
...
  if (rank == 0)
    {
      int in, out = 5;

      MPI_Sendrecv(&out, 1, MPI_INT, 1, 0,
                   &in, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);

      printf("P0 Received: %d\n",in);
    }
  else
    {
      int in, out = 25;

      MPI_Recv(&in, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
      MPI_Send(&out, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

      printf("P1 Received: %d\n", in);
    }
...
```

What happens if np > 2?
Deadlock?

Non-blocking communication

- Standard MPI sends and receives don’t return until their arguments can be safely modified by the calling process
  - sending: message envelope created, data sent/buffered
  - receiving: message received, data copied to provided buffer
- If this activity can take place concurrently with computation we aren’t fully utilizing available resources
  - assuming separate computing resource for communication

Non-blocking communication

- Non-blocking communication avoids deadlock and allows computation to be interleaved with these activities
  - call to send/receive “posts” the operation
  - must subsequently explicitly complete the operation
  - NOTE: non-blocking send/recv can be matched by standard recv/send

MPI_Isend and MPI_Irecv

int MPI_Isend (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request* request)

Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the non-blocking send has completed.

MPI_Isend and MPI_Irecv

int MPI_Irecv (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request* request)

Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to MPI_Wait or MPI_Test to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

MPI_Isend and MPI_Irecv

- Processing continues immediately without waiting for the message to be copied out/into the application buffer
- The program must use calls to MPI_Wait to verify that the non-blocking send/receive has completed

MPI_Wait

int MPI_Wait (MPI_Request* request, MPI_Status* status)

- MPI_Wait blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify any, all or some completions.

- request = a handle for your operation provided by the system, identifies the operation when you subsequently complete it

```
...
  if (rank == 0)
    {
      MPI_Isend(&val1, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &req1);
      MPI_Irecv(&val2, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &req2);
      /* do some computational work here */
      MPI_Wait(&req1, &status);
      MPI_Wait(&req2, &status);
      val1 = tmp;
    }
  else
    {
      MPI_Irecv(&val2, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &req2);
      MPI_Isend(&val1, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &req1);
      /* do some computational work here */
      MPI_Wait(&req2, &status);
      MPI_Wait(&req1, &status);
      val1 = tmp;
    }
...
```

Communication patterns

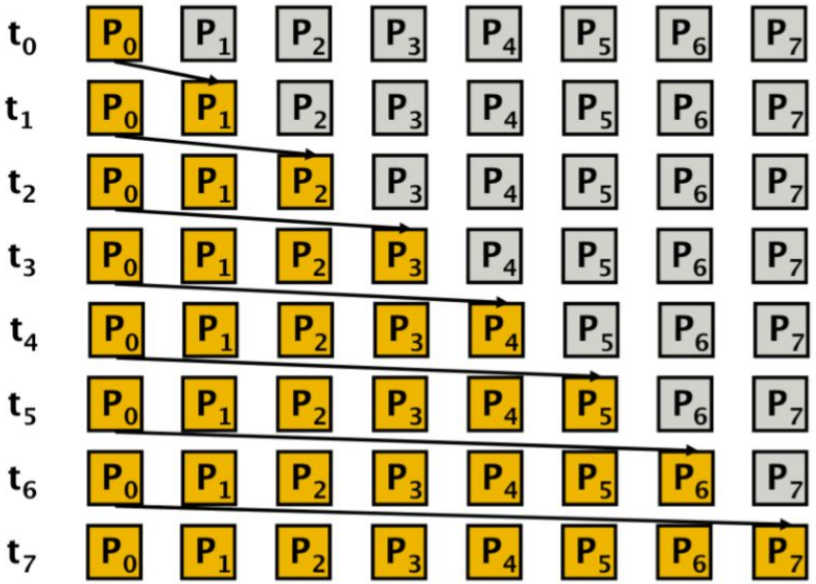
- One sends to one (usual MPI_Send and MPI_Recv)
- One sends to all (broadcast)
- All send to one (reduction combined with predefined operation)
- All send to all (all broadcast/reduction)
- Subset sends to subset
- Special: modularized by data and/or task

Broadcasting

- A process has a value that needs to be sent to all other processes

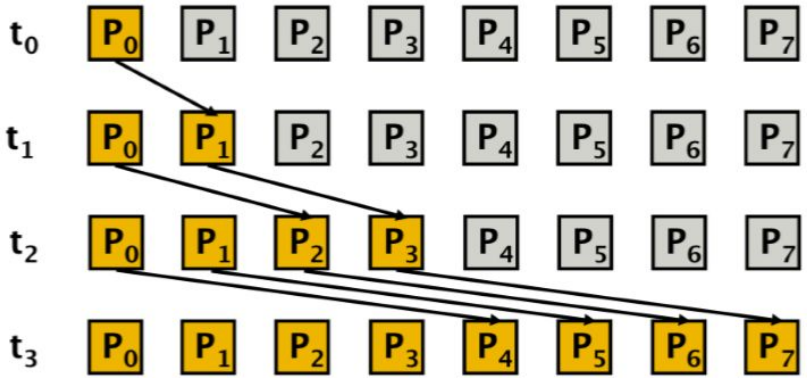
- Serial method O(N)
  - run a loop that sends value to each other process in turn
  - N−1 cycles to send value to all other processes

do i := 1, N−1
  send data from P0 to P



Parallel broadcast

- Broadcasting can take advantage of available parallelism
  - step 1: P0 sends to P1
  - step 2: P0 sends to P2, P1 sends to P3
  - step 3: P0 to P4, P1 to P5, P2 to P6, P3 to P7, and so on
  - note: there is no canonical order for communication
- log2 N steps
- This is a common parallel technique called recursive doubling: progress toward completion of process doubled at each step by doubling number of processes involved at each step
- Must consider issues of knowing sender/receiver in broadcast implementation



Reduction

- Partial results held by multiple processes need to be combined and sent to a single process

- Serial reduction O(N)

```
do i = N−2, 0, −1
    result_i := combine( result_i , result_i +1)
```

Common reduction operations

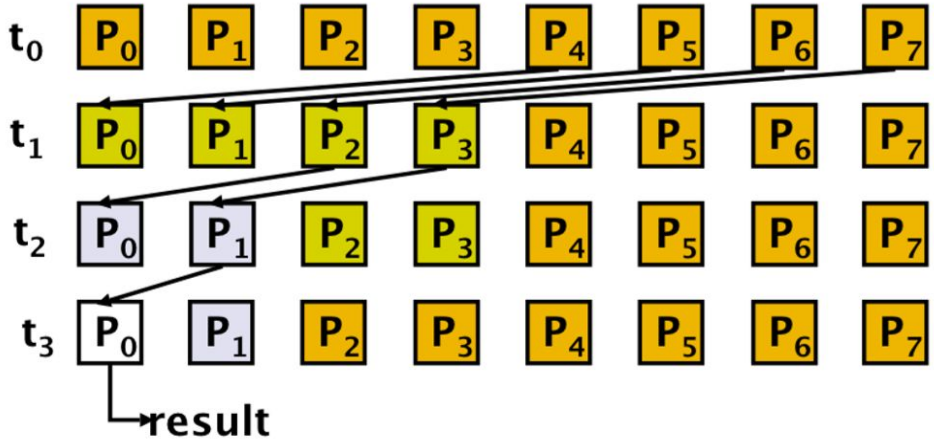
Defined for

- simple arithmetic operations: sum, difference, product, quotient (division)
- properties: maximum/minimum value, location of max/min value
- logical operations: AND, OR, XOR
- bitwise operations: AND, OR, XOR
- sorting

MPI Reduction Operation		C Data Types
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double
MPI_MINLOC	min value and location	float, double and long double

Parallel reduction

- Assuming there is no order for reduction arithmetic
- Parallel reduction is essentially the reverse of the tree-based broadcast with added operations at each step to combine partial results where necessary
- Inverted binary tree O(log2 N)



Messages

How does MPI distinguish messages from one another?

- who it is from
- who it is for
- a user defined label/tag used to mark messages
- the group of processes in which communication is taking place
- other information (attributes)

The **message envelope** contains at least these pieces of information (possibly more depending on implementation)

- sender rank
- receiver rank
- tag
- communicator

Tags

- Analogous to “marking” the messages in order to classify them
  - a tag is assigned to the message when it is sent (user provided)
  - the tag becomes part of the message envelope and can be used to organize the receipt of the message (implicitly or explicitly)

- A receive message expecting a particular tag can be programmed to complete successfully only in response to a message with that tag

- There is nothing special about tags: they are user defined constructs for the purpose of organizing communication

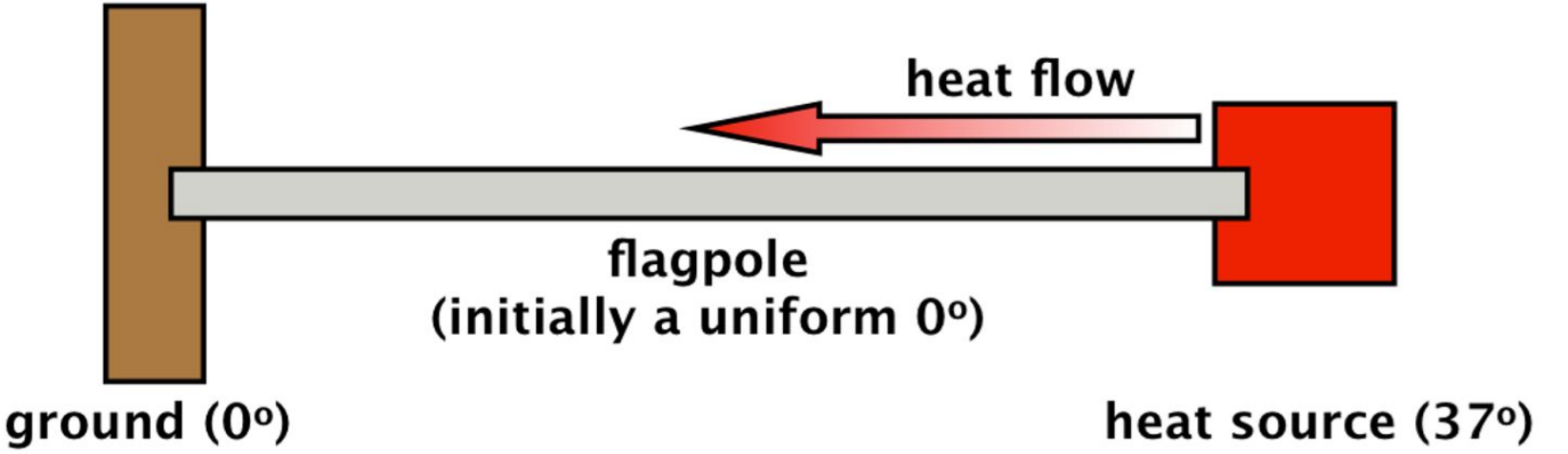
MPI bottlenecks

- There is an overhead with each MPI message⇒strongly suggest to group small messages together, e.g. send rows/columns instead of individual elements
  - there shouldn’t be millions of messages flying around (very inefficient)
- There is a limitation on the size of the send/receive buffer which is implementation- / platform- / computer-dependent
  - dictated by the available local memory and interconnect bandwidth
  - must be aware of the buffer limit when sending a lot of data

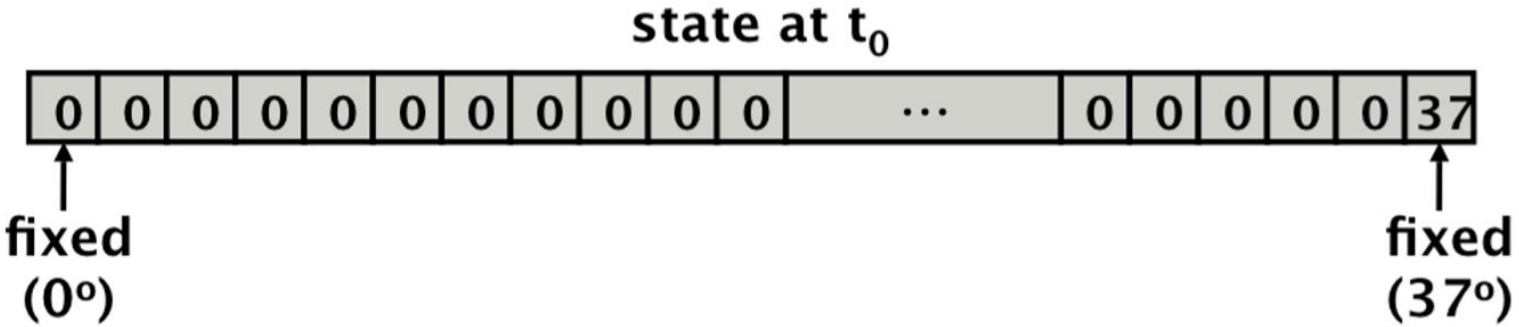
Example: heat flow

- This example is adapted from one in "High Performance Computing" by Kevin Dowd & Charles Severance, O'Reilly & Associates, 1998
- Heat flow is a classic problem in scalable parallel processing
  - in a single dimension:
    - ★ a rod at a constant temperature
    - ★ one end of the rod is exposed to a heat source
    - ★ simulate the flow of heat from one end to the other and determine its steady-state
  - in two dimensions:
    - ★ a metal sheet at a constant temperature
    - ★ heat source(s) placed beneath the sheet
    - ★ simulate flow of heat out from heat sources
  - etc.

- In our example, we will consider the following set-up:
  - we wish to find the steady state of heat distribution in a flagpole
  - assume that initially the flagpole and the ground achieved a uniform temperature of 0 degrees Celsius (we will also ignore any ambient air effects)
  - the top of the flag pole is embedded in an object that has a temperature of 37 degrees Celsius



- Note: data parallelism
- We'll treat the system in a discrete fashion
  - divide pole into *N* discrete zones (cells) with uniform temperature throughout each
  - fix temperatures of leftmost/rightmost cell to 0 and 37 respectively
  - divide time into discrete intervals at which to compute temperature
- Compute current temperature as mean of surrounding temperatures

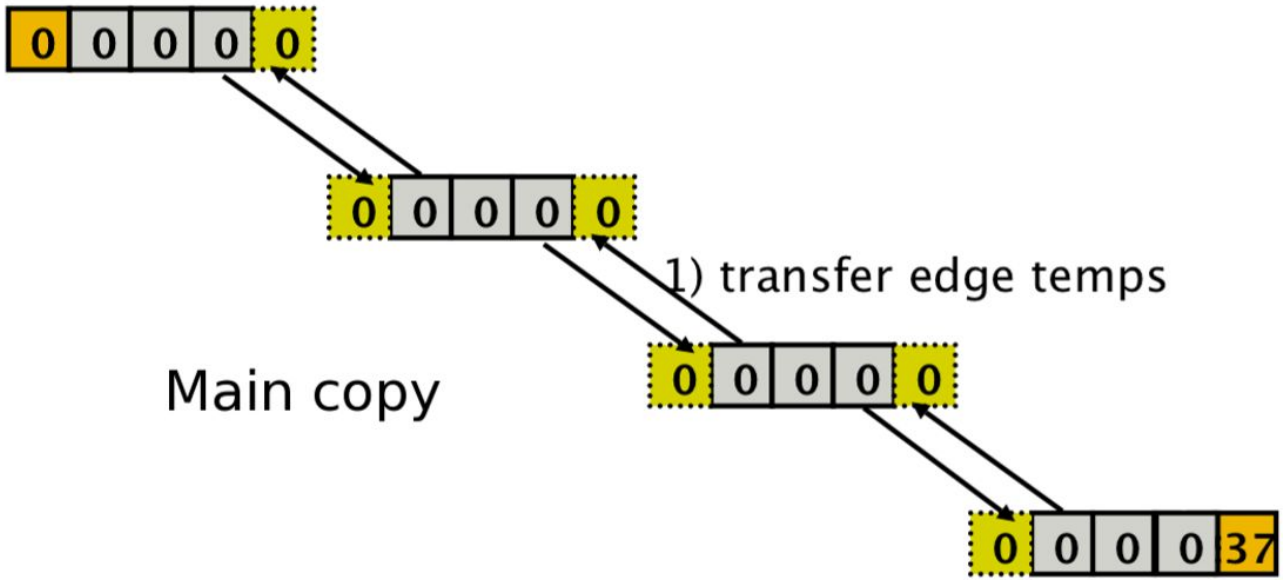


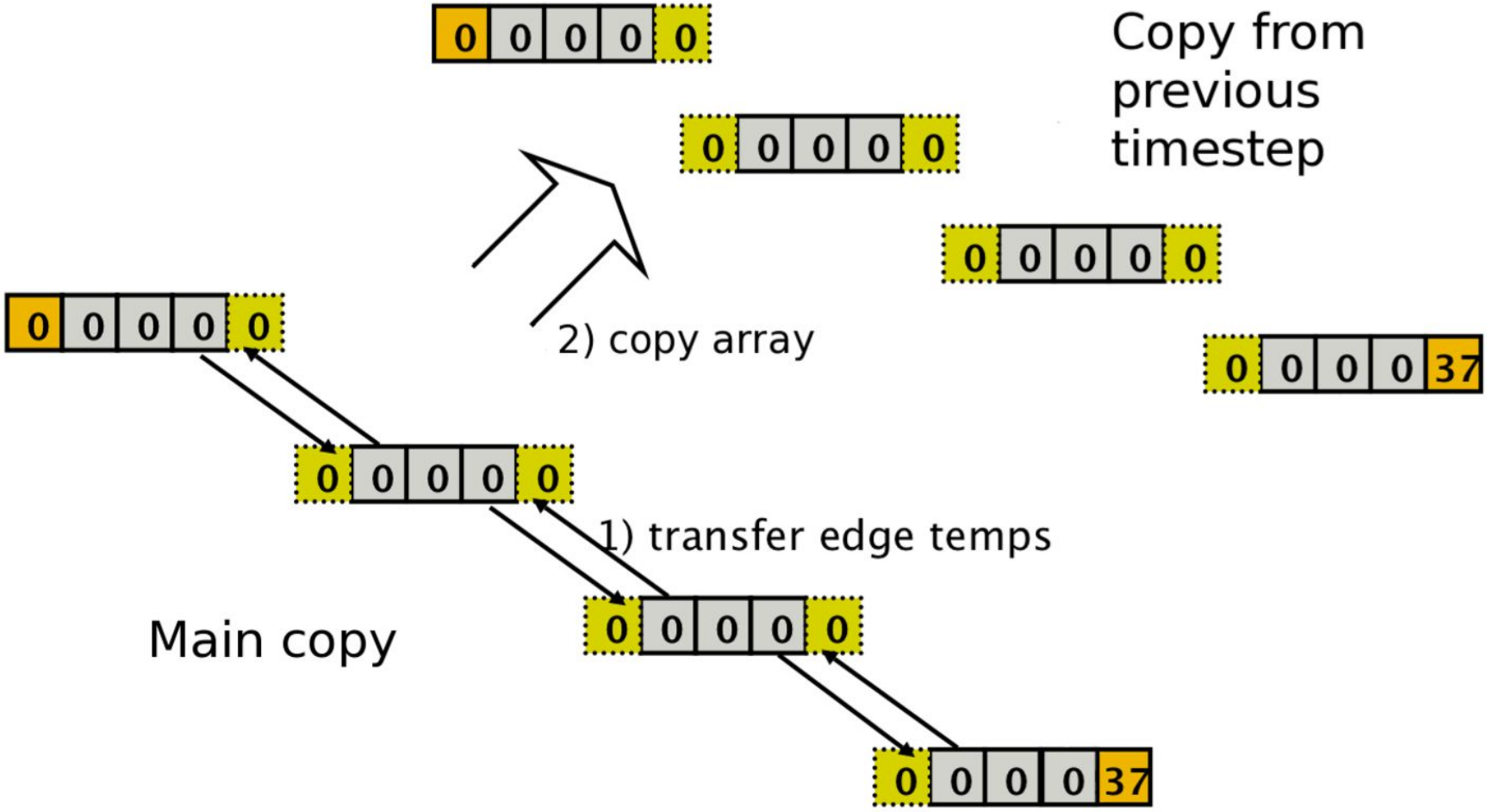
- Given K processors, divide the problem into K blocks, with $m = N/K$ zones in each
- Distribute all blocks to individual processors – **domain decomposition**
- Each processor computes update for its block at each time step:
  $T_s(t) = [T_{s-1}(t-1) + T_{s+1}(t-1)]/2, \quad \text{where } s = 1, \ldots, m$
- To compute T_s, a processor needs T_{s-1} and T_{s+1}
  - have an extra zone at each end fixed to the desired temperature $T_0 = 0$, $T_{m+1} = 37$
  - only compute T_s for $s = 1, \ldots, m$

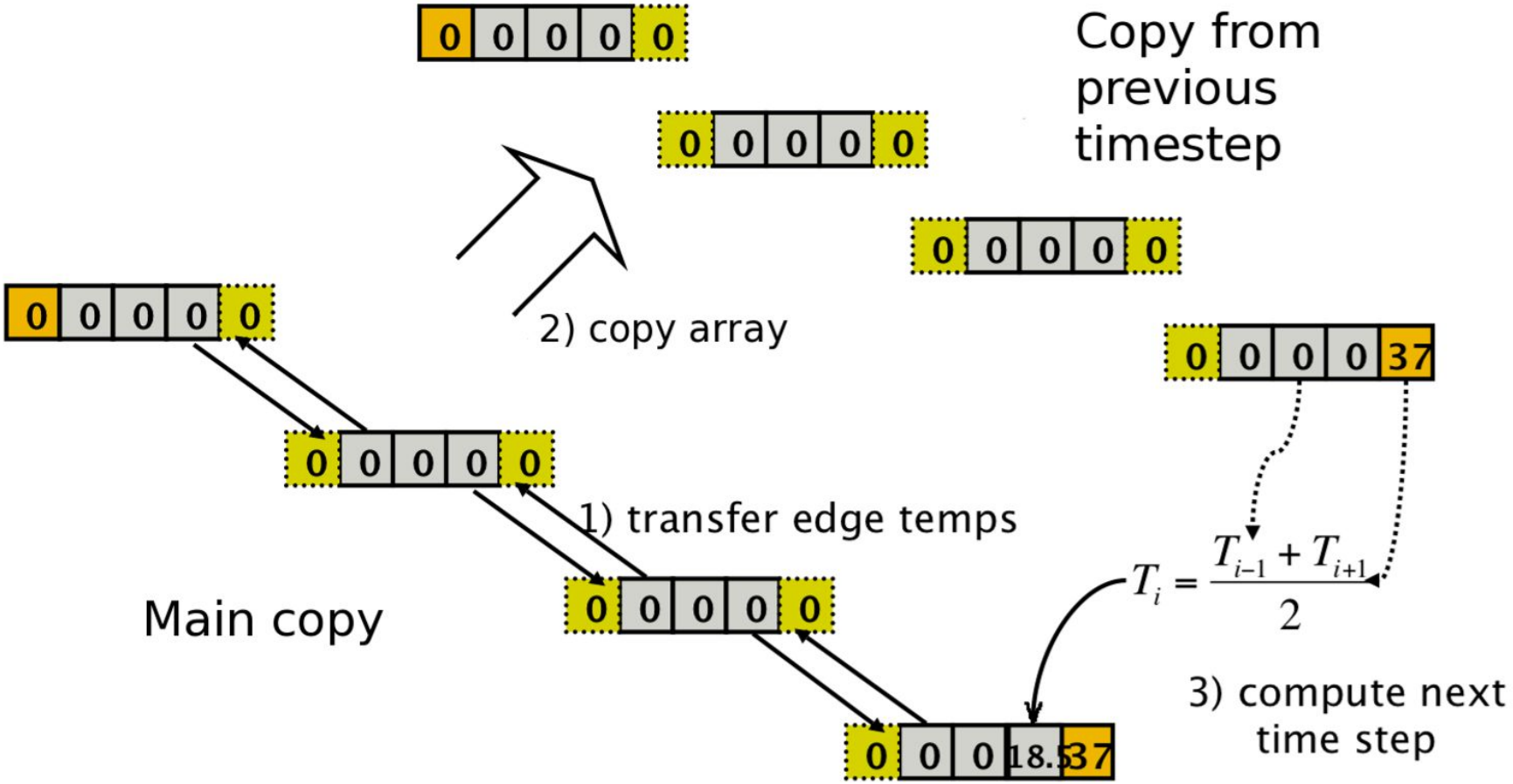
- Issue
  - consider the sequential procedure:
    ```
    for (i = 1; i <= m; i++)
        T[i] = (T[i-1] + T[i+1]) / 2;
    ```
  - note that T[i-1] will actually be the T at the next time step rather than this one (as T[i-1] was computed before T[i])
- Solution?
  - use two arrays: one as the source of values from the previous time step, while writing computed values to the other
  - how do we account for this in our parallel program (or is it even relevant)?

- Issue: consider communication needs
  - only adjacent zones on different processors need to communicate their values
  - note that there is communication flowing both left→right and left←right
  - must have two adjacent values before computing the temperature at the current time step
- Solution?
  - have two ghost zones on each processor (one on each end — array runs $0, \ldots, m+1$)
  - receive the adjacent values into the ghost zones and local loop on only values $1, \ldots, m$
  - is order of communication important?
    - adopt some convention, e.g. receive from left, send to right, or receive from right, send to left – is this safe? won't you run into a deadlock? are there other options?

Main copy


0 0 0 0 0
0 0 0 0 0
1) transfer edge temps
Main copy
0 0 0 0 0
0 0 0 0 37


0 0 0 0 0
Copy from previous timestep
0 0 0 0 0
0 0 0 0 0
2) copy array
0 0 0 0 0
0 0 0 0 37
0 0 0 0 0
1) transfer edge temps
Main copy
0 0 0 0 0
0 0 0 0 37


0 0 0 0 0
Copy from previous timestep
0 0 0 0 0
0 0 0 0 0
2) copy array
0 0 0 0 0
0 0 0 0 37
0 0 0 0 0
1) transfer edge temps
$T_i = \frac{T_{i-1} + T_{i+1}}{2}$
Main copy
0 0 0 0 0
0 0 0 18.5 37
3) compute next time step

References

- About Processes and Threads. Retrieved November 30, 2018, from https://docs.microsoft.com/en-us/windows/desktop/ProcThread/about-processes-and-threads
- Message Passing Interface. Retrieved November 30, 2018, from https://computing.llnl.gov/tutorials/mpi/
- Sharcnet HPC course slides by Alex Razoumov.