Advanced Topics in Numerical Analysis: High Performance Computing

Distributed memory algorithms (MPI)

Georg Stadler, Dhairya Malhotra
Courant Institute, NYU

Spring 2019, Monday, 5:10–7:00PM, WWH #1302

April 15, 2019

Outline

Organization issues

Programming models overview

Sources of parallelism and locality

MPI Intro (point-to-point)

Distributed Memory Performance Model

Organization

Scheduling:

- Homework assignment #4 due today
- One short homework assignment posted tomorrow

Topics today:

- The distributed memory computing model
- Sources of parallelism
- Send and recv communication in MPI

Final projects

- Everybody should know topic and teams members by now.
- On the next homework assignment, we will ask you to detail your plans and list the steps you're planning and when you will be working on what.

Outline

Organization issues

Programming models overview

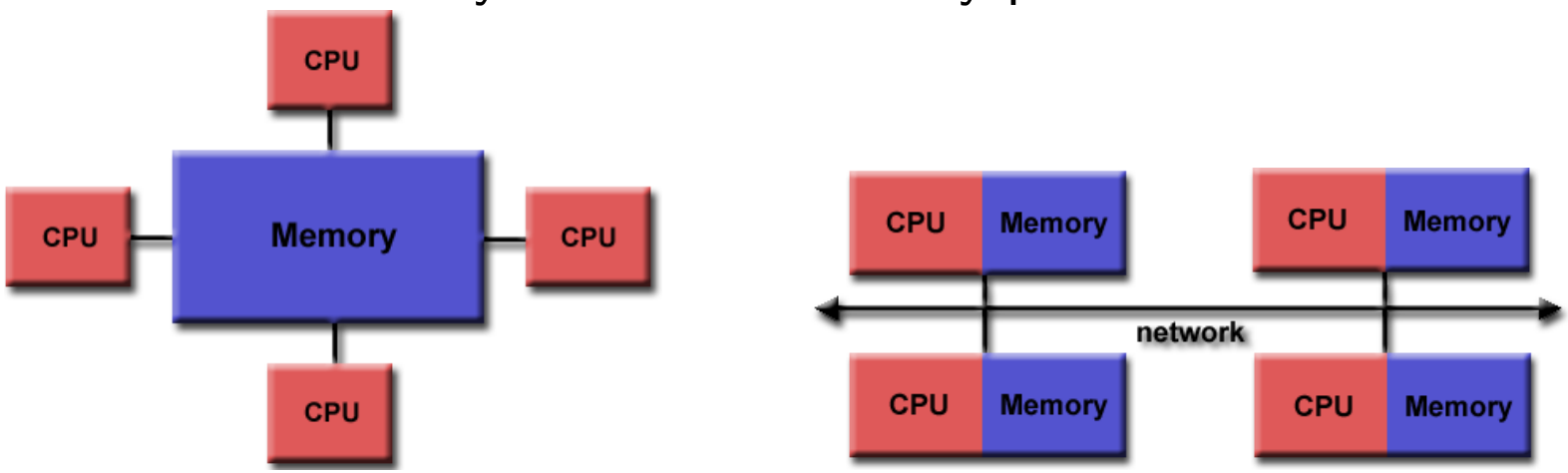
Sources of parallelism and locality

MPI Intro (point-to-point)

Distributed Memory Performance Model

Programming models

- Flynn's taxonomy:
  - Single instruction–single data (SISD)
  - Single instruction–multiple data (SIMD)
  - Multiple instruction–multiple data (MIMD)
- Distributed memory vs. shared memory parallelism



- Programming models: OpenMP vs. Message passing interface (MPI); and combinations thereof

Process vs. thread

- A process is an independent execution unit, which contains their own state information (pointers to instruction and stack). One process can contain several threads.
- Threads within a process share the same address space, and communicate directly using shared variables. Seperate stack but shared heap memory.

Outline

Organization issues

Programming models overview

Sources of parallelism and locality

MPI Intro (point-to-point)

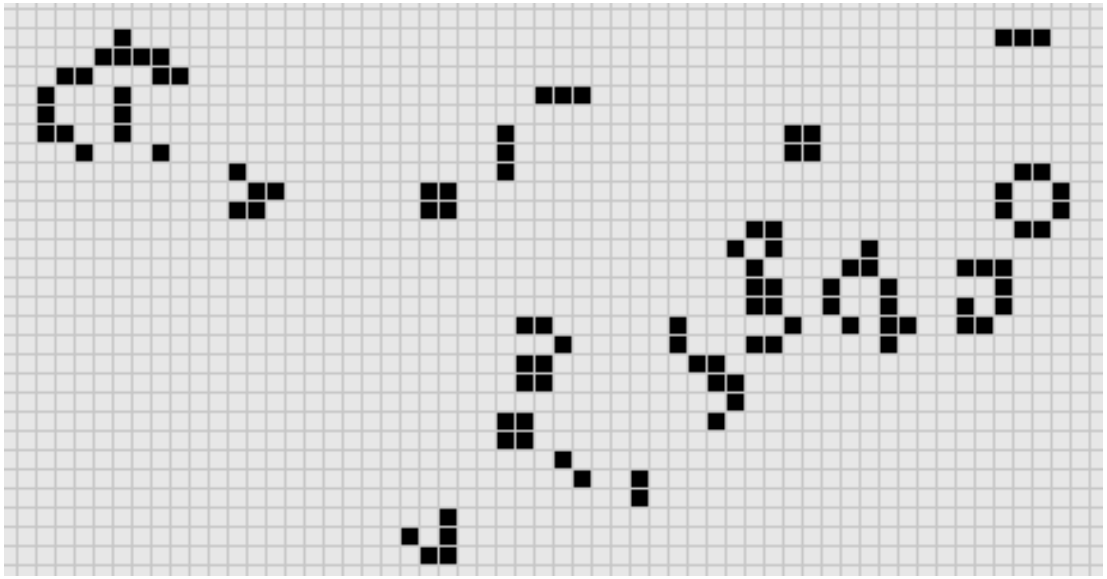
Distributed Memory Performance Model

Parallelism and locality

- Moving data (through network or memory hierarchy) is slow
- Real world problems often have parallelism and locality, e.g.,
  - objects move independently from each other ("embarrassingly parallel")
  - objects mostly influence other objects nearby
  - dependence on distant objects can be simplified
  - Partial differential equations have locality properties
- Applications often exhibit parallelism at multiple levels

Example I: Conway's game of life

https://www.youtube.com/watch?v=C2vgICfQawE



- Played on a board of "cells"; simple rules decide on if a cell is alive or dead in the next generation
- Is an example of a cellular automaton
- Amounts to checking the 8 neighbor cells in every generation
- How to parallelize? Decompose domain into subdomains...

Example II: Particle systems

A particle system has a finite number of particles which move according to Newton's law ($F = ma$); particles can be stars subject to gravity, atoms in a molecule, swimming fish, ...
Force on each particle:

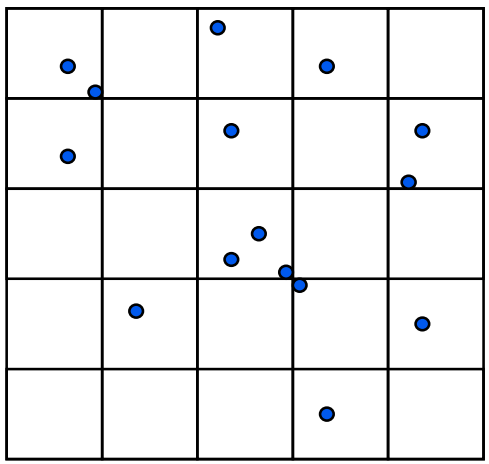
$$F_{\text{overall}} = F_{\text{external}} + F_{\text{nearby}} + F_{\text{far}}$$

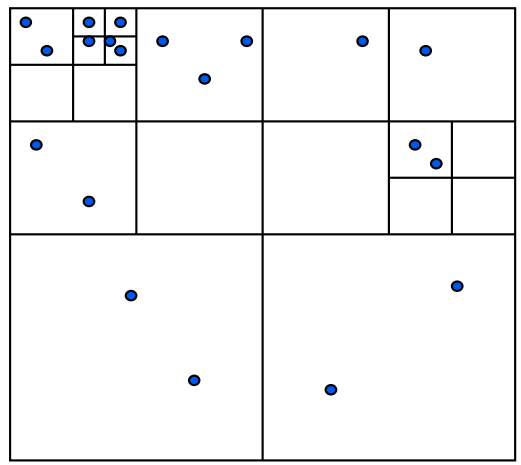
- external: background flow/ocean current/external electric field
- nearby attraction; collision force, Van der Waals forces
- far field: gravity, electrostatics

Example II: External and nearby forces

External force: independent, "embarrassingly parallel": evenly distribute particles amongst processors.
Nearby force: requires neighbor communication; assume, for instance collisions; need to check in "ghost layer" for particles on neighboring processes

- interaction of particles near processor boundary
- load imbalance if particles cluster; must be adjusted





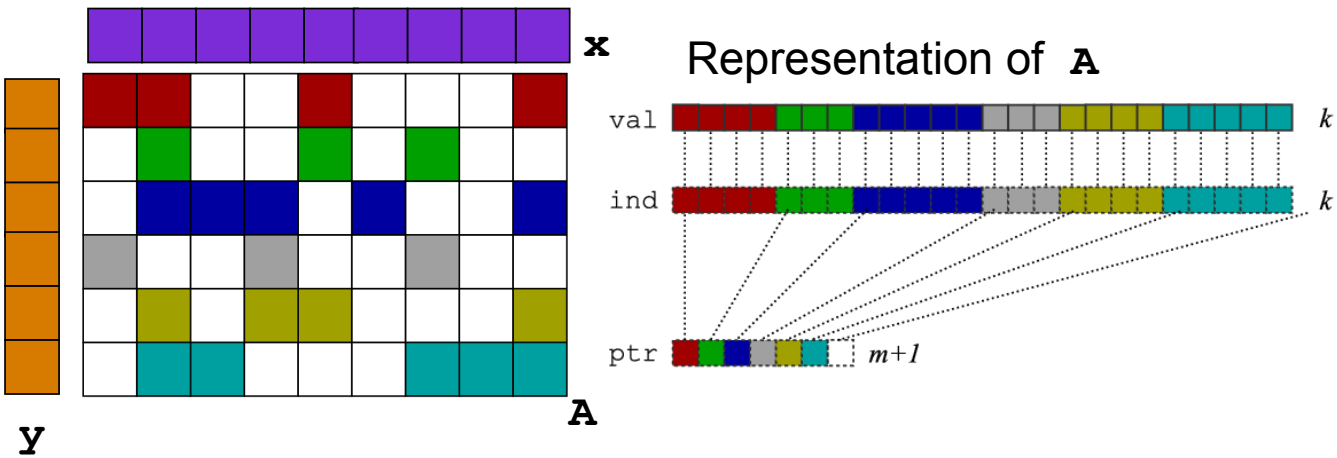
Example II: Far field forces

Far field forces involve all-to-all communication
Simple algorithm: $\mathcal{O}(n^2)$, where n is the number of particles.
More clever algorithms:

- Particle-mesh methods: interpolate particle force to nearest grid point; solve far field PDE (e.g., FFT); interpolate force back to particles
- Use tree construction; each node contains an approximation of descendants: Fast multipole method (FMM)

Example III: Sparse matrix-vector multiplication

Compressed sparse row (CSR) format:



Matrix multiplication kernel: $\boldsymbol{y} = \boldsymbol{y} + A\boldsymbol{x}$:

for each row i

for $k = \text{ptr}[i]$ to $\text{ptr}[i+1] - 1$ do

$\boldsymbol{y}[i] = A_{\text{val}}[k]\boldsymbol{x}[\text{ind}[k]]$

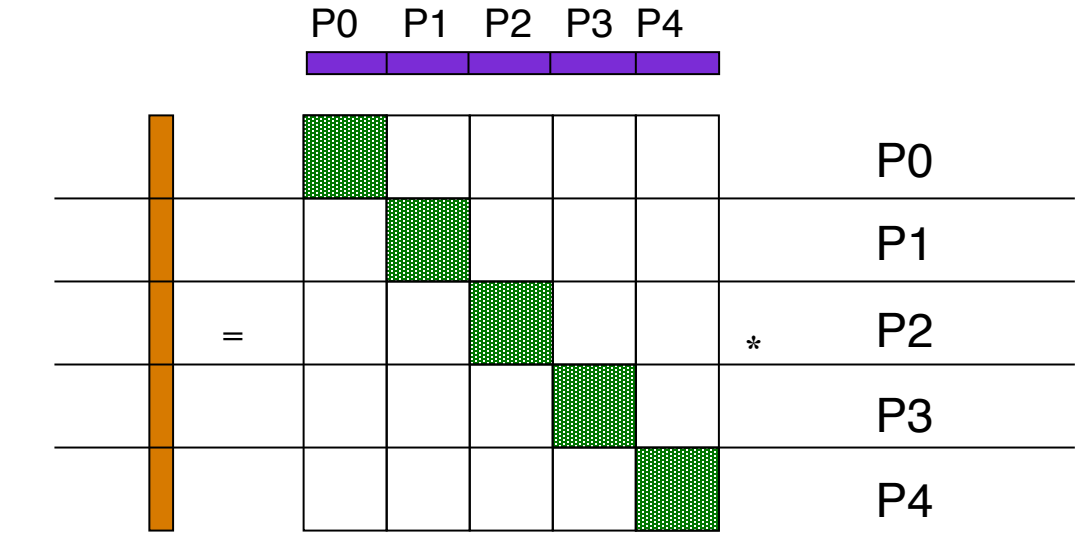
Example III: Sparse matrix-vector multiplication

How parallelize? Which processes compute/store which part of A, $\boldsymbol{x}$, $\boldsymbol{y}$?

Partition into index sets, and distribute to different processes. Requires communication if $\boldsymbol{x}$ is distributed as well.

Example III: Sparse matrix-vector multiplication

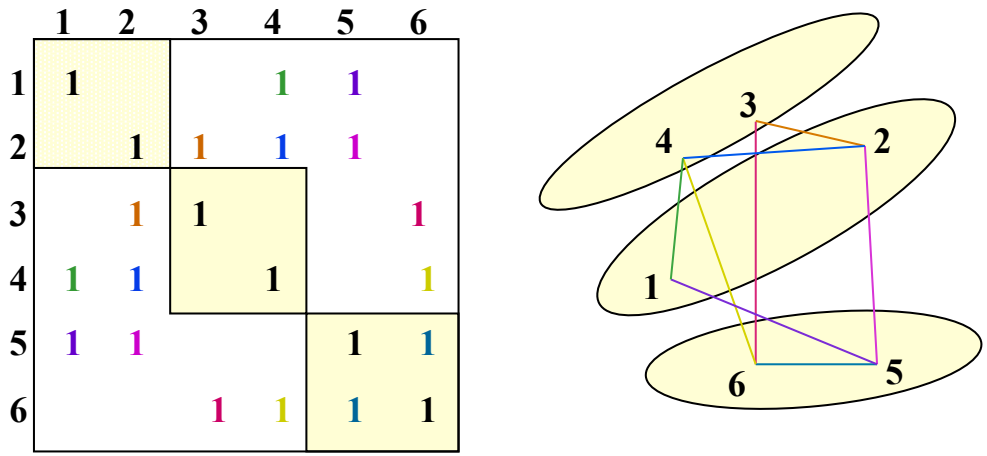
How parallelize? Which processes compute/store which part of A, $\boldsymbol{x}$, $\boldsymbol{y}$?



Communication can be reduced with proper ordering of rows/columns of A.

Example III: Sparse matrix-vector multiplication

How parallelize? Which processes compute/store which part of A, $\boldsymbol{x}$, $\boldsymbol{y}$?



Reordering and Graph Partitioning: Edges in graph correspond to nonzeros in matrix. Graph partitioning $\leftrightarrow$ minimizing communication in parallel matrix-vector multiplication.

Example IV: Partial differential equations

Types of PDEs influence parallelism

- Elliptic PDE (gravitation,elasticity,...): Steady-state, global dependence in space
- Hyperbolic PDE: (acoustic/electromagnetic waves,...): Time-dependent, local dependence in space
- Parabolic PDE (heat flow, diffusion,...): Time-dependent, global space dependence

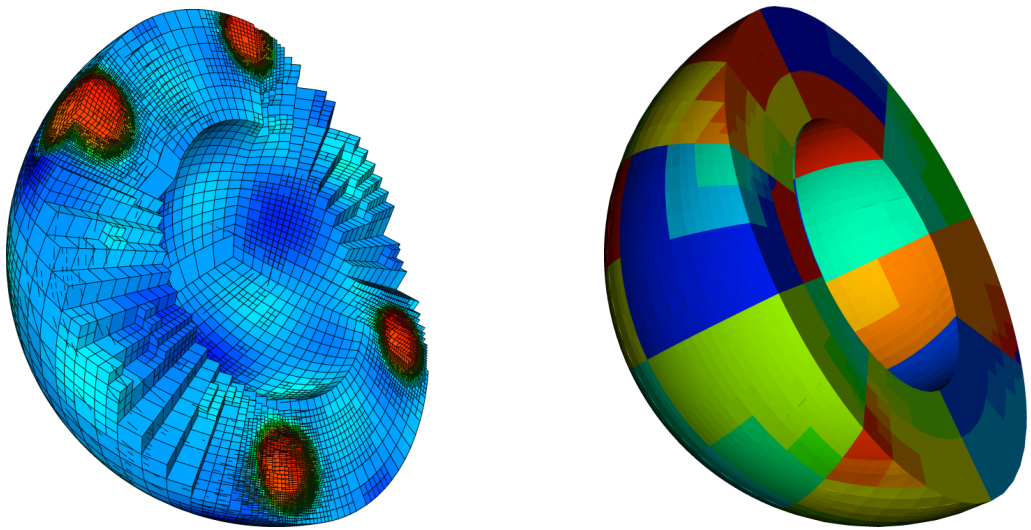
Many PDEs (e.g., Navier-Stokes equation) combine properties of these basic types.

Example IV: Partial differential equations: elliptic

$$-\Delta u = f \text{ on } \Omega$$

$$+ \text{ bdry cond.}$$

After discretization, this is becomes a system with a positive definite, symmetric matrix. Efficient solvers include geometric or algebraic multigrid or FFT (requires proper boundary conditions and mesh). Parallel Gauss elimination allow limited parallelism.



Field governing mesh refinement (left). Mesh partitioning, each color illustrates mesh portion owned by a different processor (right).

Example IV: Partial differential equations: hyperbolic

$$u_{tt} - \Delta u = f \text{ on } \Omega$$
$$+ \text{ bdry cond.}$$
$$+ \text{ initial cond.}$$

Often, method of choice is explicit time stepping, which requires a matrix-vector multiplication in each time step:

$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \delta t A \boldsymbol{u}^k$$

Explicit time stepping is commonly used. CFL stability does not restrict the size of the time step δt significantly.
Parallelization based on decomposition of mesh (leads to a similar decomposition as for sparse matrices).

Example IV: Partial differential equations: parabolic

$$u_t - \Delta u = f \text{ on } \Omega$$
$$+ \text{ bdry cond.}$$
$$+ \text{ initial cond.}$$

Stability is a problem for explicit time stepping (requires very small time step!). Thus, one usually uses implicit time stepping:

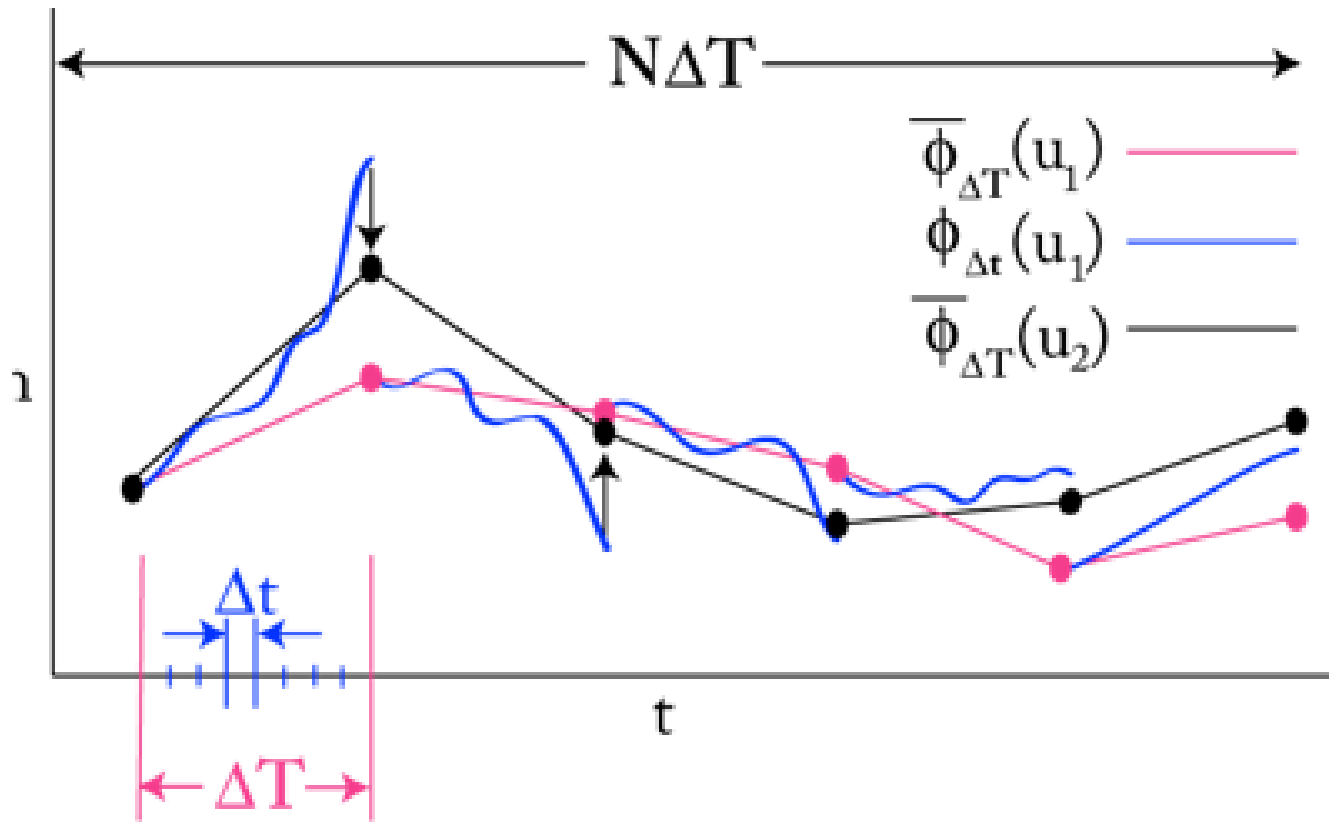
$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \delta t A \boldsymbol{u}^{k+1},$$

which requires to solve systems in every time step. These are similar as in the case of an elliptic PDE (solvers: multigrid, FMM,...) Parallelization based on decomposition of mesh.

Example IV: Partial differential equations: parallel-in-time

Parallelization-in-time is an active field of research. Basic idea:

- Use a fast and inaccurate serial time integration method $\bar{\Phi}$ as starting guess
- Iterative local-in-time parallel correction with more accurate time integration Φ



Outline

Organization issues

Programming models overview

Sources of parallelism and locality

MPI Intro (point-to-point)

Distributed Memory Performance Model

Introduction to MPI

Use B. Gropp's PPT slides

https://github.com/NYU-HPC19/lecture10

Outline

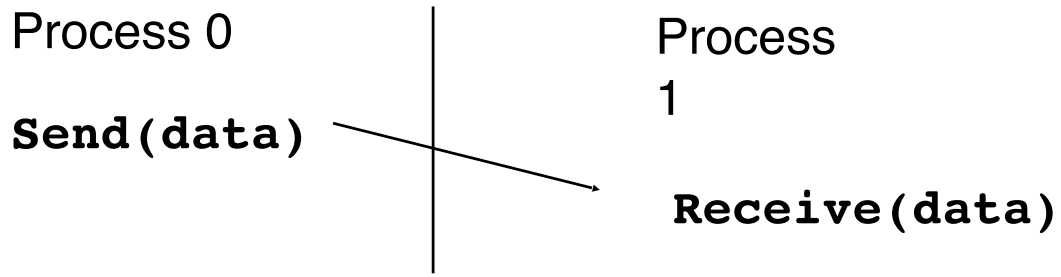
- Background
  - The message-passing model
  - Origins of MPI and current status
  - Sources of further MPI information
- Basics of MPI message passing
  - Hello, World!
  - Fundamental concepts
  - Simple examples in Fortran and C
- Extended point-to-point operations
  - non-blocking communication
  - modes

The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.

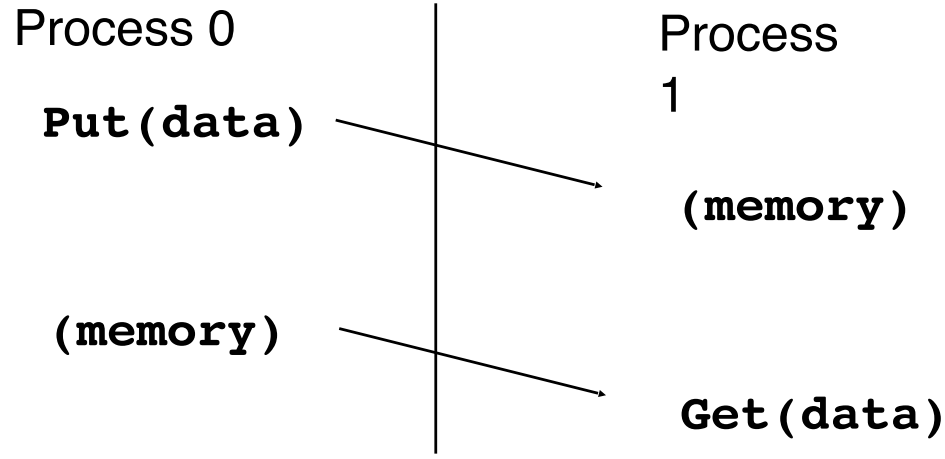
Cooperative Operations for Communication

- The message-passing approach makes the exchange of data *cooperative*.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.



One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
- One-sided operations are part of MPI-2.



What is MPI?

- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

Why Use MPI?

- MPI provides a powerful, efficient, and *portable* way to express parallel programs
- MPI was explicitly designed to enable libraries…
- … which may eliminate the need for many users to learn (much of) MPI

A Minimal MPI Program (C)

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

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

Running MPI Programs

- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- **`mpiexec -np 3 <args>`** `or`
- **`mpirun -np 3 <args>`** `(not specified in MPI standard)`

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

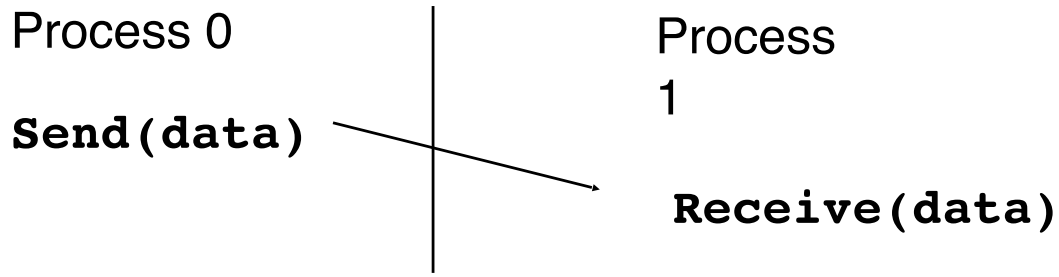
Better Hello (C)

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

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 );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

MPI Basic Send/Receive

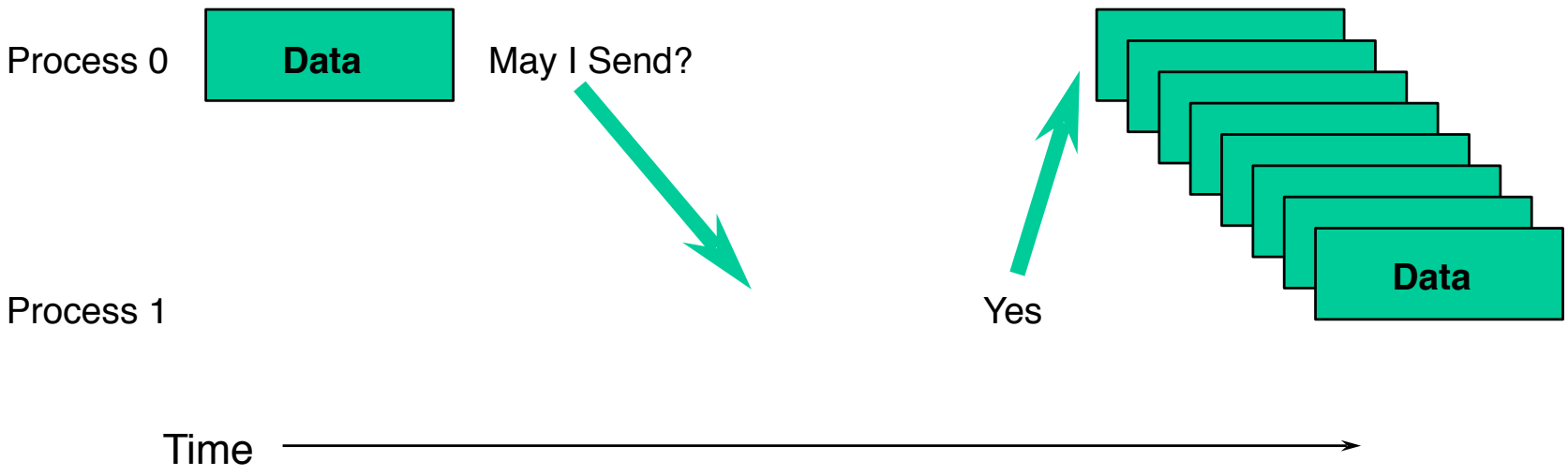
- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

What is message passing?

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.
- Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with datatypes.

MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- 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(start, count, datatype, source, tag, comm, status)

- Waits until a matching (on **`source`** and **`tag`**) message is received from the system, and the buffer can be used.
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**.
- **`status`** contains further information
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

Why Datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication).
- Specifying application-oriented layout of data in memory
  - reduces memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available

Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of “wild card” tags.
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application
- Use MPI_Comm_split to create new communicators

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**
- Point-to-point (send/recv) isn't the only way...

Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This is called “unsafe” because it depends on the availability of system buffers

Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	**`Recv(0)`**
`Recv(1)`	**`Send(0)`**

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	**`Isend(0)`**
`Irecv(1)`	**`Irecv(0)`**
`Waitall`	**`Waitall`**

Outline

Organization issues

Programming models overview

Sources of parallelism and locality

MPI Intro (point-to-point)

Distributed Memory Performance Model

Distributed Memory Performance Models

Postal model:

- Hockney, Jesshope: Parallel Computers 2: architecture, programming and algorithms (1988)
- Simplest model for distributed memory point-to-point communication.
- Parameters: latency (t_s), per-word-transfer time (t_w)

$$t_{comm} = t_s + t_w\ m$$

where, m is the message size in bytes.

Other models:

- **LogP model:** latency (L), overhead (o), per-word-transfer time (g), number of processors (P)