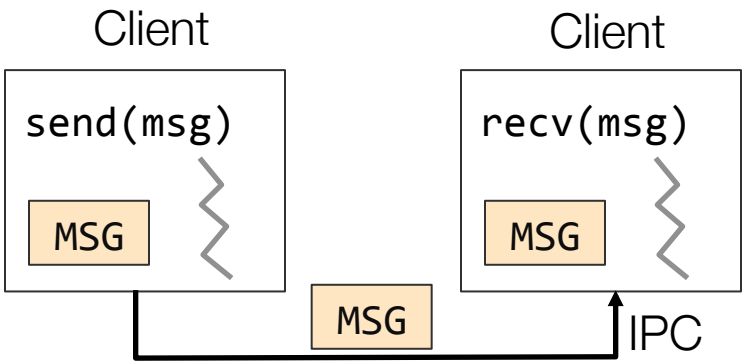
MPI and OpenMP (Lecture 25, cs262a)

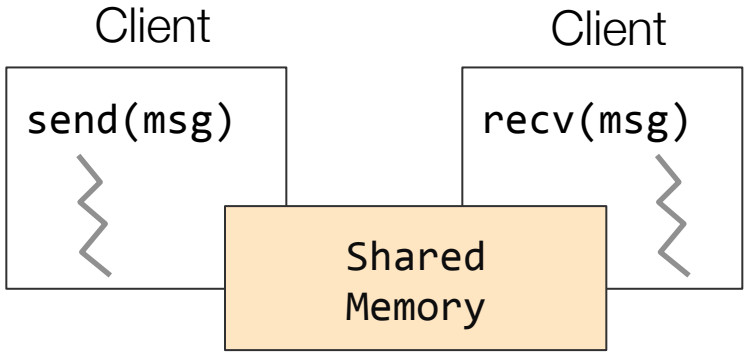
Ion Stoica,
UC Berkeley
November 19, 2016

Message passing vs. Shared memory



Message passing: exchange data explicitly via IPC

Application developers define protocol and exchanging format, number of participants, and each exchange



Shared memory: all multiple processes to share data via memory

Applications must locate and and map shared memory regions to exchange data

Architectures

Uniformed Shared
Memory (UMA)
Cray 2

Non-Uniformed Shared
Memory (NUMA)
SGI Altix 3700

Massively Parallel
DistrBluegene/L

Orthogonal to programming model

MPI

MPI - Message Passing Interface

- Library standard defined by a committee of vendors, implementers, and parallel programmers
- Used to create parallel programs based on message passing

Portable: one standard, many implementations

- Available on almost all parallel machines in C and Fortran
- De facto standard platform for the HPC community

Groups, Communicators, Contexts

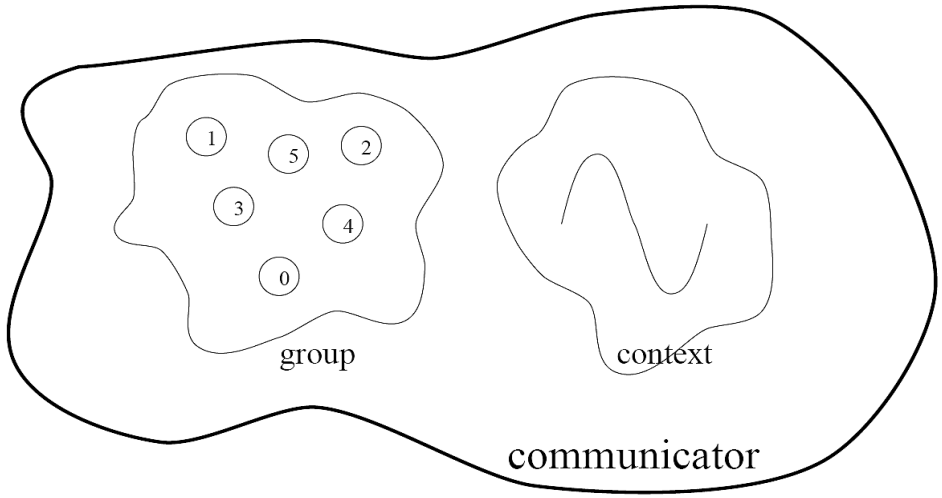
Group: a fixed ordered set of *k* processes, i.e., 0, 1, .., k-1

Communicator: specify scope of communication

- Between processes in a group
- Between two disjoint groups

Context: partition communication space

- A message sent in one context cannot be received in another context



This image is captured from:
"Writing Message Passing Parallel Programs with MPI", Course Notes, Edinburgh Parallel Computing Centre
The University of Edinburgh

Synchronous vs. Asynchronous Message Passing

A synchronous communication is not complete until the message has been received

An asynchronous communication completes before the message is received

Communication Modes

Synchronous: completes once ack is received by sender

Asynchronous: 3 modes

- Standard send: completes once the message has been sent, which may or may not imply that the message has arrived at its destination
- Buffered send: completes immediately, if receiver not ready, MPI buffers the message locally
- Ready send: completes immediately, if the receiver is ready for the message it will get it, otherwise the message is dropped silently

Blocking vs. Non-Blocking

Blocking, means the program will not continue until the communication is completed

- Synchronous communication
- Barriers: wait for every process in the group to reach a point in execution

Non-Blocking, means the program will continue, without waiting for the communication to be completed

MPI library

Huge (125 functions)

Basic (6 functions)

MPI Basic

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

Skeleton MPI Program (C)

```c
#include <mpi.h>

main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    /* main part of the program */

    /* Use MPI function call depend on your data
     * partitioning and the parallelization architecture
     */
    MPI_Finalize();
}
```

A minimal MPI program (C)

```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;
}
```

A minimal MPI program (C)

`#include` “`mpi.h`” provides basic MPI definitions and types.

MPI_Init starts MPI

MPI_Finalize exits MPI

Notes:

- Non-MPI routines are local; this “printf” run on each process
- MPI functions return error codes or MPI_SUCCESS

Error handling

By default, an error causes all processes to abort

The user can have his/her own error handling routines

Some custom error handlers are available for downloading from the net

Improved Hello (C)

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    /* rank of this process in the communicator */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    /* get the size of the group associates to the communicator */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

Improved Hello (C)

```
/* Find out rank, size */
int world_rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int number;
if (world_rank == 0)
  number = -1;
  MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
  MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("Process 1 received number %d from process 0\n", number);
}
```

Number of elements

Rank of destination

Tag to identify message

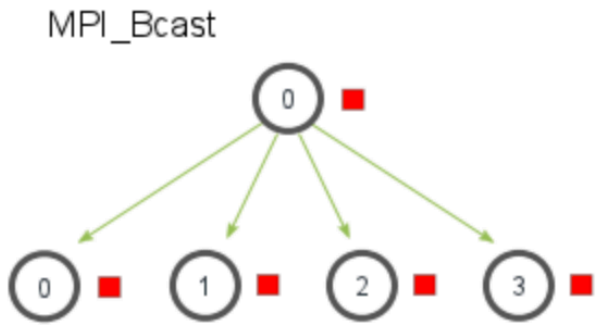
Default communicator

Rank of source

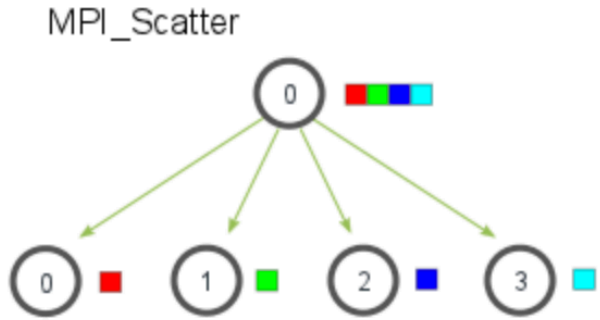
Status

Many other functions…

MPI_Bcast: send same piece of data to all processes in the group



MPI_Scatter: send different pieces of an array to different processes (i.e., partition an array across processes)

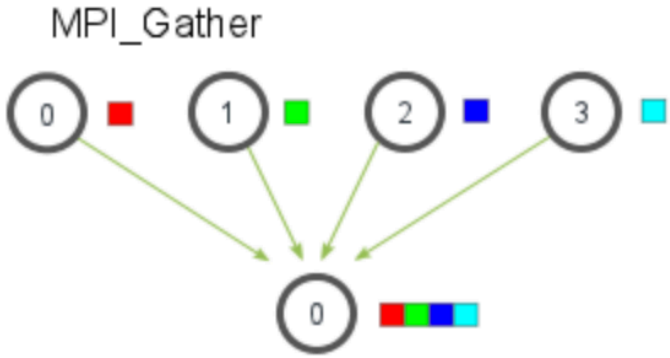


From: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Many other functions…

MPI_Gather: take elements from many processes and gathers them to one single process

- E.g., parallel sorting, searching

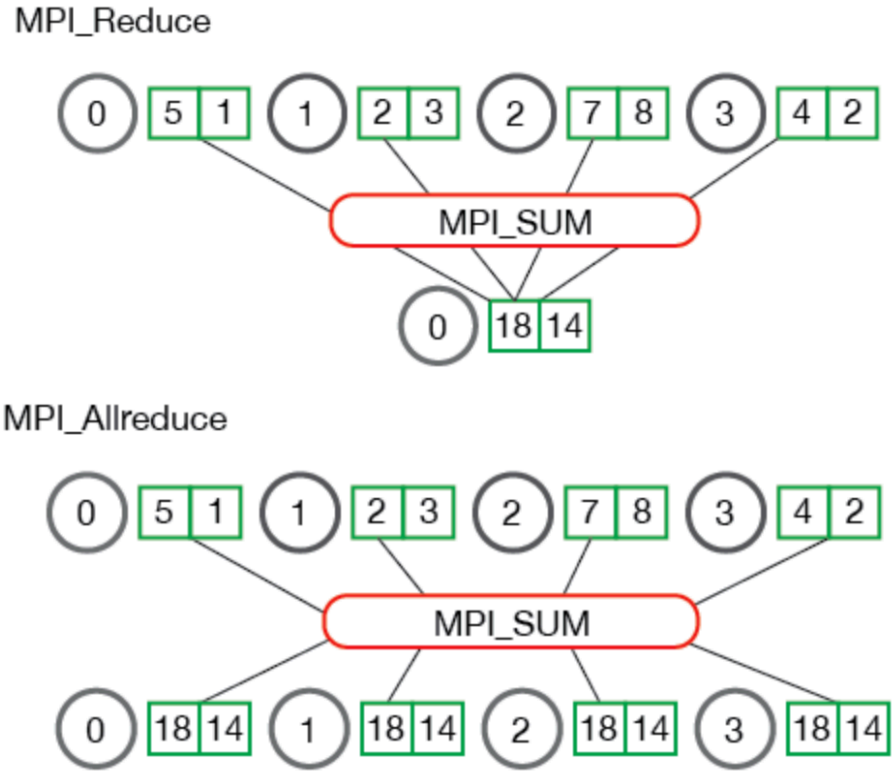


From: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Many other functions…

MPI_Reduce: takes an array of input elements on each process and returns an array of output elements to the root process given a specified operation

MPI_Allreduce: Like MPI_Reduce but distribute results to all processes



From: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

MPI Discussion

Gives full control to programmer

- Exposes number of processes
- Communication is explicit, driven by the program

Assume

- Long running processes
- Homogeneous (same performance) processors

Little support for failures, no straggler mitigation

Summary: achieve high performance by hand-optimizing jobs but requires experts to do so, and little support for fault tolerance

OpenMP

Based on the “Introduction to OpenMP” presentation:
(webcourse.cs.technion.ac.il/236370/Winter2009.../OpenMPLecture.ppt)

Motivation

Multicore CPUs are everywhere:

- Servers with over 100 cores today
- Even smartphone CPUs have 8 cores

Multithreading, natural programming model

- All processors share the same memory
- Threads in a process see same address space
- Many shared-memory algorithms developed

But…

Multithreading is hard

- Lots of expertise necessary
- Deadlocks and race conditions
- Non-deterministic behavior makes it hard to debug

Example

Parallelize the following code using threads:

```
for (i=0; i<n; i++) {
  sum = sum + sqrt(sin(data[i]));
}
```

Why hard?

- Need mutex to protect the accesses to sum
- Different code for serial and parallel version
- No built-in tuning (# of processors?)

OpenMP

A language extension with constructs for parallel programming:

- Critical sections, atomic access, private variables, barriers

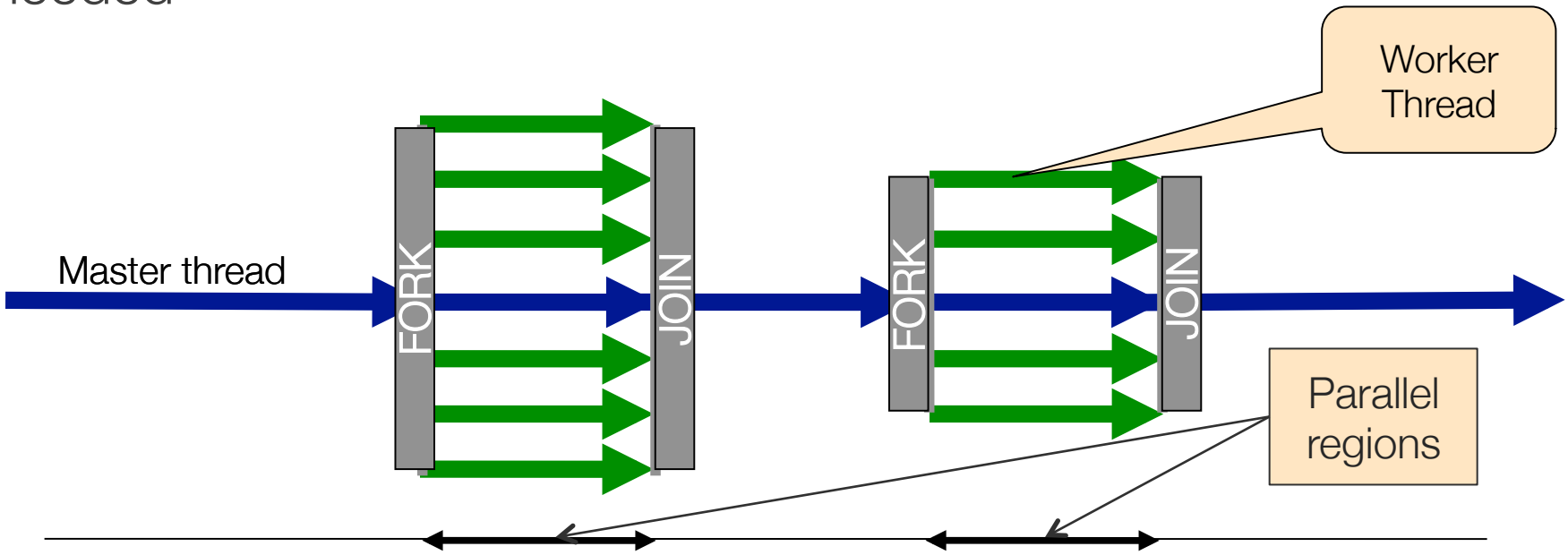
Parallelization is orthogonal to functionality

- If the compiler does not recognize OpenMP directives, the code remains functional (albeit single-threaded)

Industry standard: supported by Intel, Microsoft, IBM, HP

OpenMP execution model

Fork and Join: Master thread spawns a team of threads as needed



OpenMP memory model

Shared memory model

- Threads communicate by accessing shared variables

The sharing is defined syntactically

- Any variable that is seen by two or more threads is shared
- Any variable that is seen by one thread only is private

Race conditions possible

- Use synchronization to protect from conflicts
- Change how data is stored to minimize the synchronization

OpenMP: Work sharing example

```
answer1 = long_computation_1();
answer2 = long_computation_2();
if (answer1 != answer2) { … }
```

How to parallelize?

OpenMP: Work sharing example

```
answer1 = long_computation_1();
answer2 = long_computation_2();
if (answer1 != answer2) { … }
```

How to parallelize?

```
#pragma omp sections
{
  #pragma omp section
  answer1 = long_computation_1();
  #pragma omp section
  answer2 = long_computation_2();
}
if (answer1 != answer2) { … }
```

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nt = omp_get_num_threads();
  int i_start = id*N/nt, i_end = (id+1)*N/nt;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nt = omp_get_num_threads();
  int i_start = id*N/nt, i_end = (id+1)*N/nt;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

- Launch *nt* threads
- Each thread uses *id* and *nt* variables to operate on a different segment of the arrays

OpenMP: Work sharing example

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int nt = omp_get_num_
  i<
}
```

Automatic parallelizat
the for loop usi

```
#pragma omp parallel
  gma omp for  chedule  tatic)
  for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
}
```

#parallel for

Increment:
var++, var--,
var += incr, var -= incr

Comparison:
var *op* last, where
op: <, >, <=, >=

Initialization:
var = init

One signed variable in the loop

Challenges of #parallel for

Load balancing

- If all iterations execute at the same speed, the processors are used optimally
- If some iterations are faster, some processors may get idle, reducing the speedup
- We don’t always know distribution of work, may need to re-distribute dynamically

Granularity

- Thread creation and synchronization takes time
- Assigning work to threads on per-iteration resolution may take more time than the execution itself
- Need to coalesce the work to coarse chunks to overcome the threading overhead

Trade-off between load balancing and granularity

Schedule: controlling work distribution

schedule(static [, chunksize])

- Default: chunks of approximately equivalent size, one to each thread
- If more chunks than threads: assigned in round-robin to the threads
- Why might want to use chunks of different size?

schedule(dynamic [, chunksize])

- Threads receive chunk assignments dynamically
- Default chunk size = 1

schedule(guided [, chunksize])

- Start with large chunks
- Threads receive chunks dynamically. Chunk size reduces exponentially, down to chunksize

OpenMP: Data Environment

Shared Memory programming model

- Most variables (including locals) are shared by threads

```
{
   int sum = 0;
   #pragma omp parallel for
   for (int i=0; i<N; i++) sum += i;
}
```

- Global variables are shared

Some variables can be private

- Variables inside the statement block
- Variables in the called functions
- Variables can be explicitly declared as private

Overriding storage attributes

private:

- A copy of the variable is created for each thread
- There is no connection between original variable and private copies
- Can achieve same using variables inside { }

firstprivate:

- Same, but the initial value of the variable is copied from the main copy

lastprivate:

- Same, but the last value of the variable is copied to the main copy

```
int i;
#pragma omp parallel for private(i)
for (i=0; i<n; i++) { … }
```

```
int idx=1;
int x = 10;
#pragma omp parallel for \
  firsprivate(x) lastprivate(idx)
for (i=0; i<n; i++) {
   if (data[i] == x)
       idx = i;
}
```

Reduction

```
for (j=0; j<N; j++) {
  sum = sum + a[j]*b[j];
}
```

How to parallelize this code?

- sum is not private, but accessing it atomically is too expensive
- Have a private copy of sum in each thread, then add them up

Use the reduction clause

```
#pragma omp parallel for reduction(+: sum)
```

- Any associative operator could be used: +, -, ||, |, *, etc
- The private value is initialized automatically (to 0, 1, ~0 …)

#pragma omp reduction

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for(int i = 0; i < N; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

Conclusions

OpenMP: A framework for code parallelization

- Available for C++ and FORTRAN
- Based on a standard
- Implementations from a wide selection of vendors

Relatively easy to use

- Write (and debug!) code first, parallelize later
- Parallelization can be incremental
- Parallelization can be turned off at runtime or compile time
- Code is still correct for a serial machine