PACE

Introduction to Parallel Programming
July 22, 2014

Mehmet (Memo) Belgin, PhD

Scientific Computing Consultant
Georgia Tech, OIT-ART, PACE
mehmet.belgin@oit.gatech.edu

www.pace.gatech.edu

What to expect (overview)

- "Jump start" your parallel programming adventure
- Reinforce your existing parallel programming skills
- NOT a comprehensive parallel programming class
- Familiarity with two main languages (extensions): OpenMP and MPI
- Familiarity with simple parallel computing theory and concepts
- Understanding of common mistakes and pitfalls
- Hands-on and replicable examples/demos
- Gain the ability to write simple "working" codes from scratch
- Self-contained class material, which can be replayed (= crowded slides!)

Fetching Course Material

- SSH to one of the PACE headnodes (assuming you have a PACE account)
- Then type:

```
$ pace-register-classes

    (and follow instructions, make sure you pick the correct class in the list)
```

- This script will:
  - Add your username to the registration list
  - Drop the codes and PDF slides in your home directory:

```
$(HOME)/PACE_Classes/IntroToParallel
```

- Alternatively, you can download the PDF slides from:

  http://www.pace.gatech.edu/workshop/PACE_Intro_to_ParallelComp.pdf

Why Parallel?

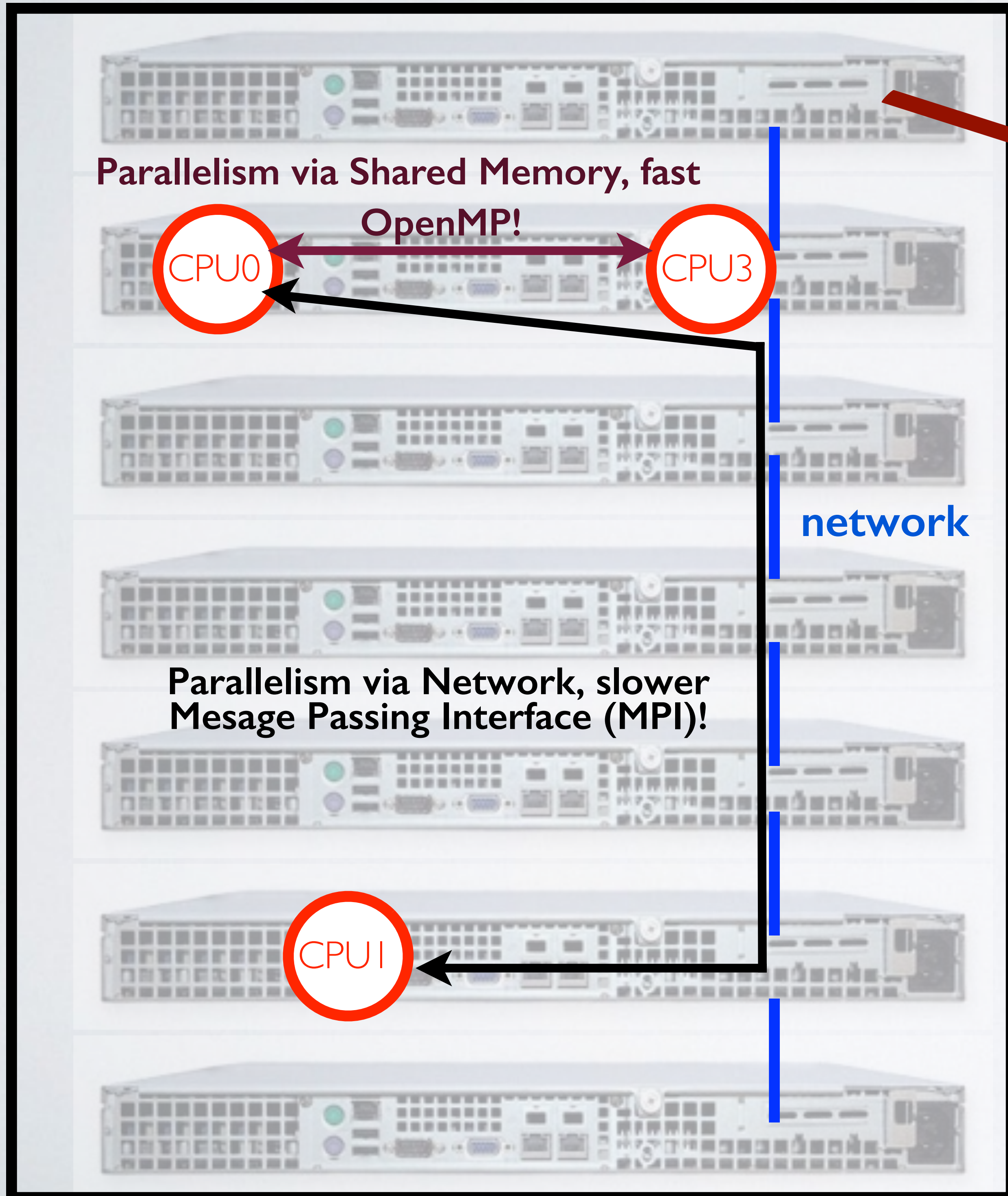
“In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.”

- Grace Hopper

- Light is **not** fast enough. It can travel “only” ~3” per cycle for a 4GHz CPU! (c = 0.983 ft/nanosec)

- Cannot make CPUs larger, and cannot fit more power in the limited area (without melting it).

A Cluster of "Multicore" Nodes



Node

Memory

16-core CPU

CPU0 CPU1 CPU2 CPU3

Processes vs. Threads

- **Process:** An instance of a computer program. Processes cannot access other processes' memory space.

- **Thread:** Single sequence of instructions. Multiple threads can exist in a single Process. All threads have access to that Process' memory space.

In most cases, each core runs a single thread or process, although there are exceptions (e.g. Intel's hyperthreading)

- For a single node: Shared memory multithreaded parallelism
  - OpenMP
  - Pthreads

- For multiple nodes: Message Passing, distributed parallelism
  - MPI (Message Passing Interface)

Shared Memory Parallelism
OpenMP

Shared Memory: OpenMP

- A library of extensions to existing programming languages, such as C, C++, Fortran.

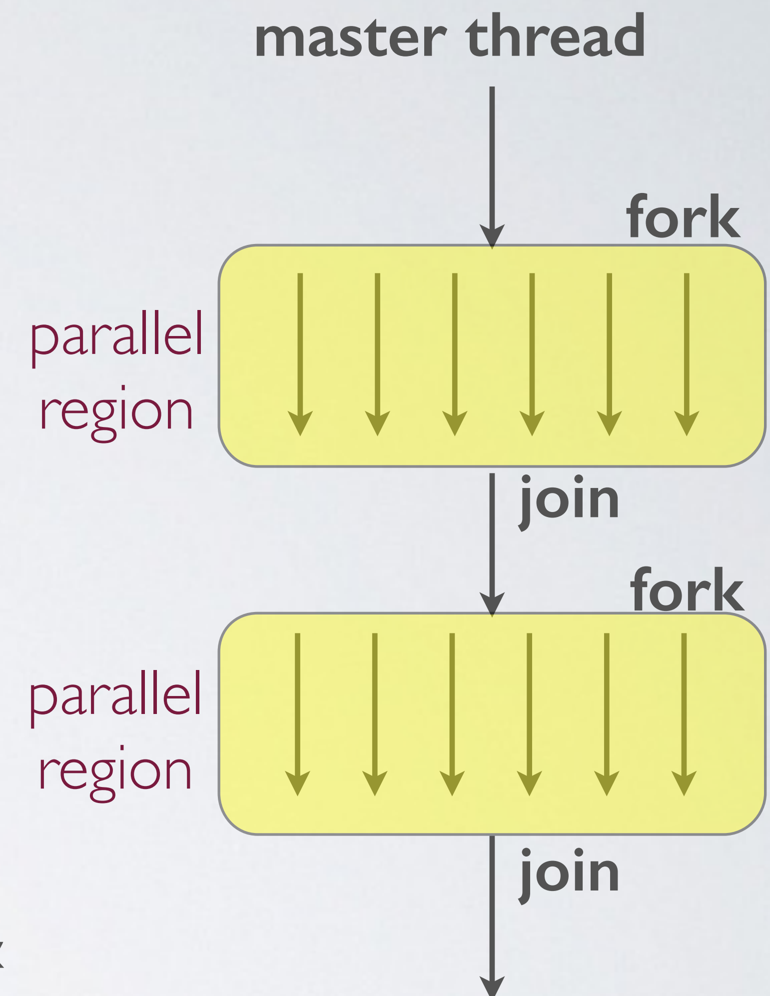
- Bundled with most leading compiler suites (GNU, Intel, PGI, etc)

- Uses "directives" to mark loops and/or specific code sections to be executed in parallel. E.g:

```
C/C++  : #pragma omp parallel for private ( i, j )
Fortran: !$omp parallel do private ( i, j )
```

- Begins execution as a single process, spawns multiple threads on demand (when parallel sections are encountered). **Fork & Join** model.

- Requires compilation with specific flags:
  - GNU: -fopenmp
  - Intel : -openmp
  - PGI : -mp



OpenMP Example: Sum of an Array

OpenMP is ideal for parallel loops. Loops in existing sequential codes can be simply marked with a "parallel" directive. Let's try it!

- Example: Calculate the sum of elements in an array (`openMPI/arraysum_first_attempt.c`)

```
sum = x[0] + x[1] + x[2] + ... + x[N]
```

- Here are some OpenMP functions/directives for starters:
  - `#pragma omp parallel for` : A parallel "for" loop ("do" in fortran)
  - `omp_get_num_threads()` : returns the number of threads
  - `omp_set_num_threads()` : sets the number of threads

Steps:

```
1. (in code) Include the omp.h header file
2. (in code) Define the number of threads (omp_set_num_threads)
3. (in code) Mark the parallel region (#pragma omp)
4. Compile with openMP parameters (see OpenMP/Makefile)
5. Compare the result and performance
```

(Buggy) OpenMP Example: Sum of an Array

(openMP/arraysum_first_attempt.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define N  1000000000
...
...
omp_set_num_threads(16);                # Set the number of threads
int numthr = omp_get_num_threads();     # Get the number of threads
printf ("Number of threads is %d.\n", numthr);
...
...
#pragma omp parallel for       # openMP directive for a parallel “loop” (not region!)
for ( i = 0; i < N; i++ )
    sum = sum + x[i];
```

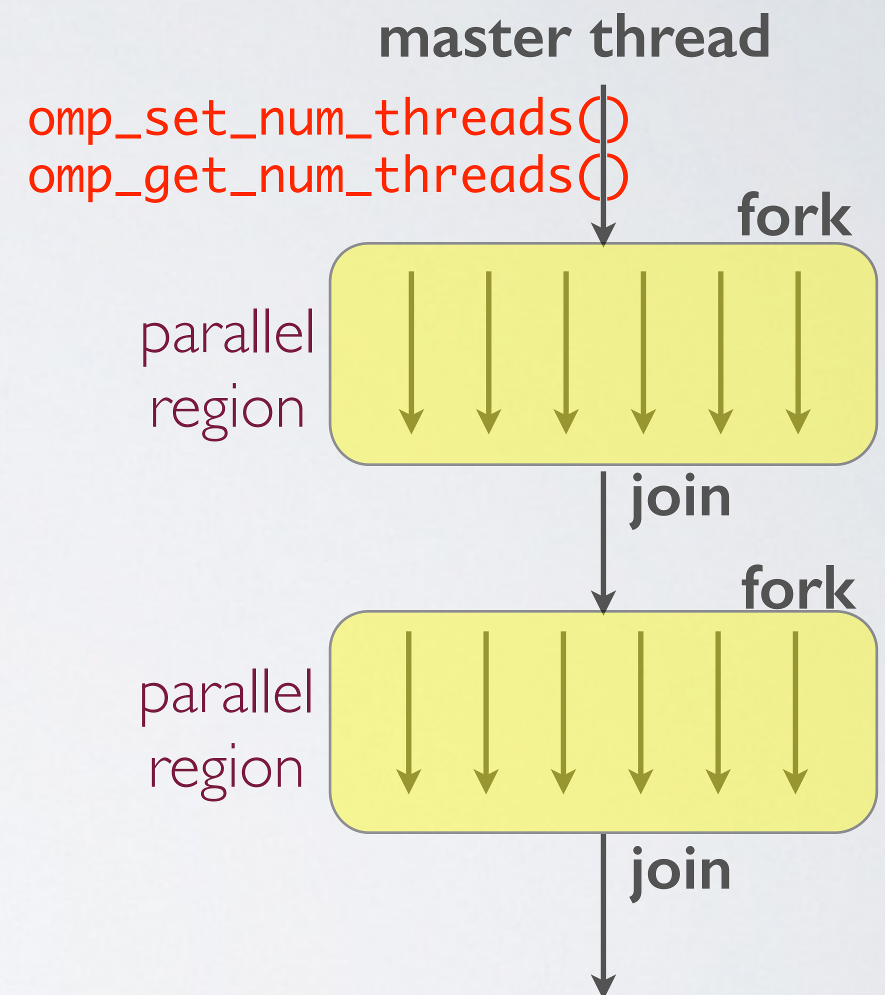
```
$ gcc -O2 -fopenmp arraysum_first_attempt.c -o arraysum_first_attempt # Compilation
$ ./arraysum_first_attempt
Number of threads is 1.                                    # Was expecting 16 (?)
Serial   SUM = 49501072177, process took: 1.951827 sec.
Parallel SUM = 3094097417, process took: 0.336662 sec.   # Incorrect Result (?)
Speedup for 1 cores: 5.797586x
```

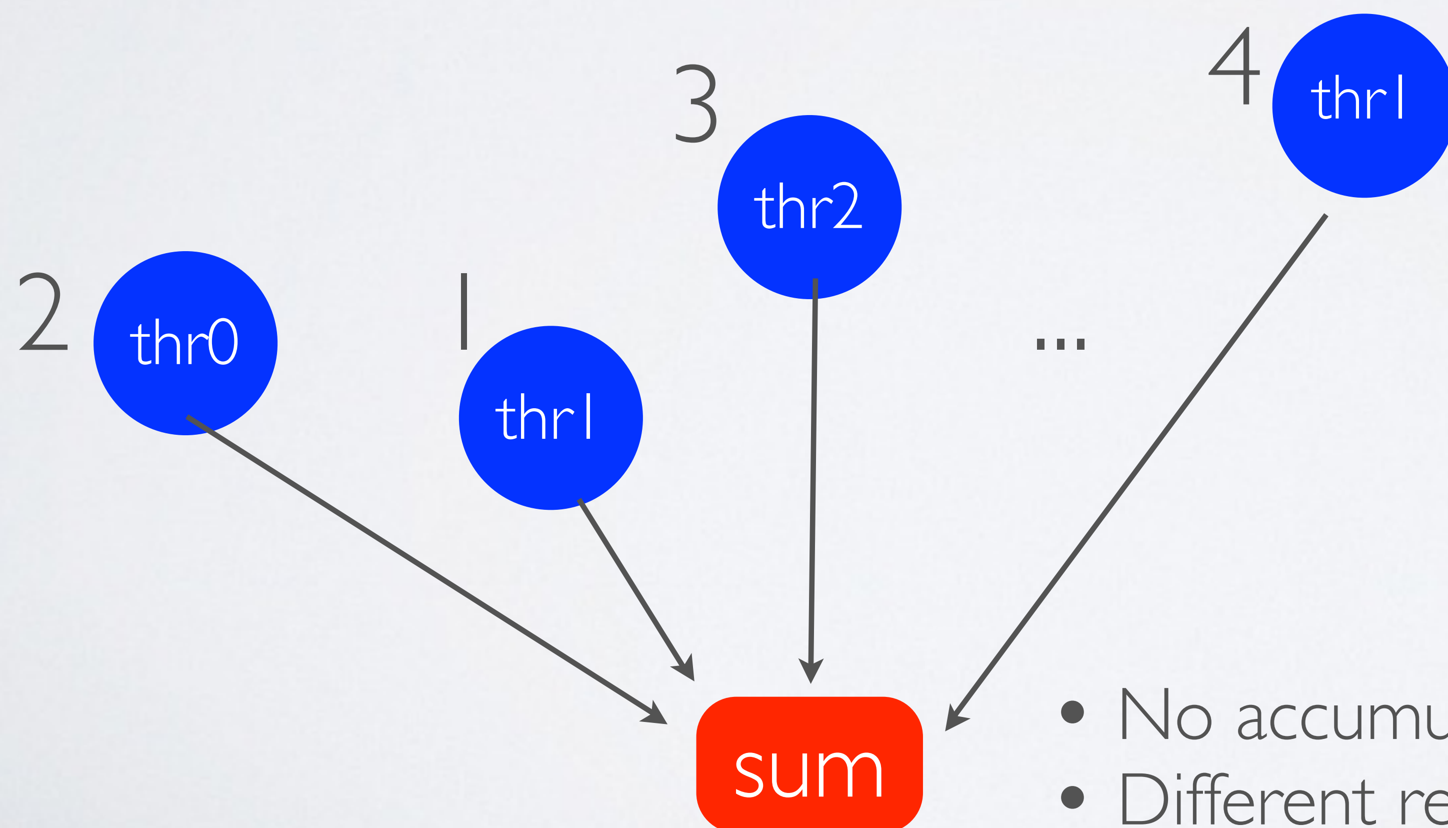
OpenMP Example: Sum of an Array

`(openMP/arraysum_first_attempt.c)`

What went wrong?

- Called `omp_get_num_threads()` in a sequential region!
- Threads tried to overwrite "sum" defined in their local copy! (race condition)





- No accumulation on sum, just "overwrite"
- Different results at each run!

(sum = the local sum for whichever thread arrived the last)

Parallel Programming Challenges

Programmers need to:

- have excellent command of the underlying language (C/C++/Fortran)
- identify regions which can be parallelized and are worth doing so (hotspots)
- think in parallel and sequential at the same time
- keep track of variable scopes (private vs. shared)
- handle data conflicts, race conditions, deadlocks and accumulation of error
- be aware of thread execution order/pattern (code must be deterministic)
- choose whether or not to synchronize and where (barriers)
- be aware of implementation-specific differences (loose MPI/OpenMP standards!)

OpenMP Components

- Compiler Directives

```
Format:  sentinel directive-name [clause, ...]

C/C++   e.g. : #pragma omp parallel for private ( i, j )
Fortran e.g. : !$omp parallel do private ( i, j )
```

- Runtime Libraries

```
C/C++  e.g. : int omp_get_num_threads(void)
Fortan e.g. : INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

- Environment Variables:

  Independent of the language but dependent on shell (bash, csh, etc)

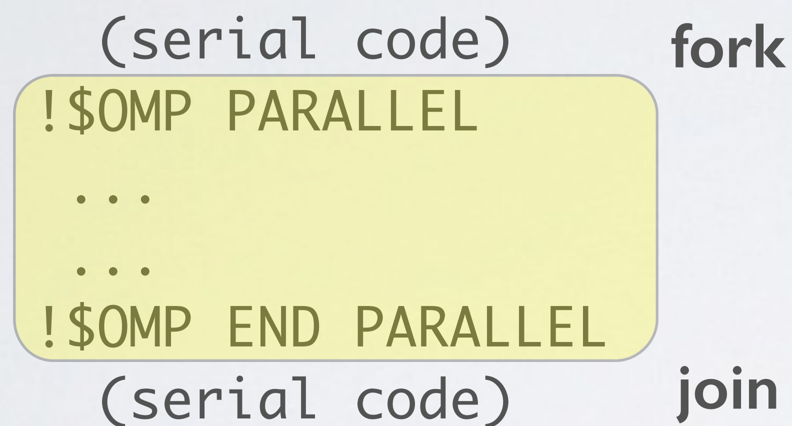
```
    bash e.g. : export OMP_NUM_THREADS=8
    csh  e.g. : setenv OMP_SCHEDULE “guided, 4”
```

- When in conflict, values set by runtime libraries win

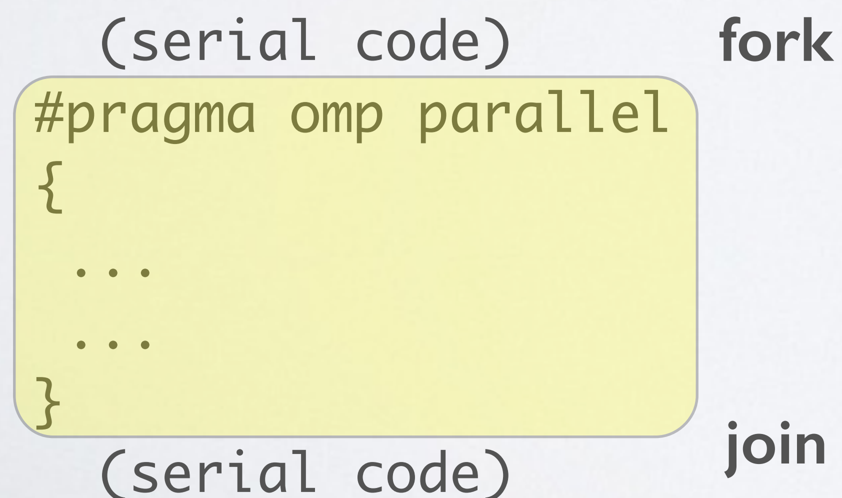
```
export OMP_NUM_THREADS=16
numThr = omp_set_num_threads(64);     # this one is in the code, so numThr=64!
```

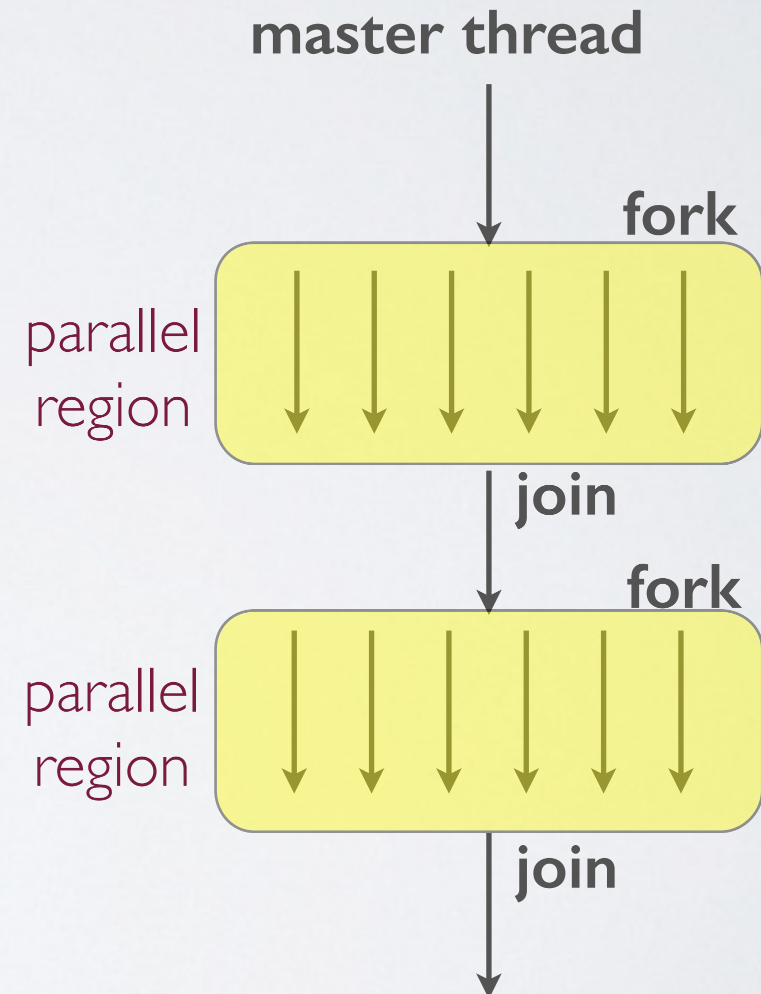
OpenMP Parallel "Regions"

- Parallel Region: All threads do the same thing in parallel
  - Fortran



- C/C++





OpenMP Parallel Region Example

(openMP/parallel_region.c)

```
#include <stdio.h>
#include <omp.h>
main ()
{
        int numThreads, threadID;
        // OMP_NUM_THREADS no longer effective
        omp_set_num_threads(8);

        // Fork into a parallel region. ThreadID must be private!
        printf ("==== Into the Parallel region ========\n");
        #pragma omp parallel private(threadID)
        {
            threadID = omp_get_thread_num();
            printf("Hello! I am thread #    %d\n", threadID);

            // If master, print out the number of threads
            if (threadID == 0) {
                numThreads = omp_get_num_threads();
                printf("Number of threads = %d\n", numThreads);
            }
        }
        // Join: only master thread continues execution
        printf ("==== Back in the Sequential Region ========\n");
        numThreads = omp_get_num_threads();
        printf("Number of threads = %d\n", numThreads);
}
```

(openMP/parallel_region.f)

```
       PROGRAM REGIONDEMO
       USE OMP_LIB
       INTEGER NTHREADS, THREADID

C OMP_NUM_THREADS no longer effective
       call OMP_SET_NUM_THREADS(8)

C Fork into a parallel region. THREADID must be private!

      PRINT *, "==== Into the Parallel region ========"
!$OMP PARALLEL PRIVATE(THREADID)

      THREADID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello! I am thread #', THREADID

C If master, print out the number of threads
      IF (THREADID .EQ. 0) THEN
        NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
      END IF

!$OMP END PARALLEL
C Join: only master thread continue execution
      PRINT *, "==== Back in the Sequential Region ========"
      NTHREADS = OMP_GET_NUM_THREADS()
      PRINT *, 'Number of threads = ', NTHREADS

       END
```

Reduction

- Variables that are modified by a reduction "operator" must be identified in the reduction clause.

- Reduction examples:

```
- reduction ( + : sum1, sum2, sum3 ) // several sums inside the same region is OK
- reduction ( * : prdct)             // product
- reduction ( max : x )              // get the maximum value (only Fortran!)
```

- Reduction Operators

```
Fortran : +, *, -, .AND., .OR., .EQV., .NEQV, max, min
C       : +, *, -, /, &, ^, |, &&, ||
```

(be careful when using C++, which allows for operator overloading!!)

(still buggy) Sum of an Array: Parallel Regions

(openMP/arraysum_region.c)

```
...
sum = 0.0;
int mythrid, loopstart, loopend;
int workperthr = N / numthr; // Each thread performs apprx N/numthr of the work
int remainder = N % numthr;  // The last thread get the remainder, if any
...

#pragma omp parallel reduction ( +: sum)
{
        mythrid   = omp_get_thread_num();
        loopstart = (mythrid * workperthr);
        loopend   = (loopstart + workperthr);
        if ( mythrid == (numthr - 1) ) loopend += remainder;

        printf ("I am thr #%d, processing from %d to %d \n", mythrid, loopstart, loopend - 1);

        for ( i = loopstart; i < loopend; i++ )
                sum = sum + x[i];
}
```

```
$ export OMP_NUM_THREADS=4            # To reduce the clutter in the output below
$ ./arraysum_region
Serial   SUM = 49498583, process took: 0.001669 sec.
I am thr #0, processing from 0 to 249999
I am thr #3, processing from 750000 to 999999
I am thr #1, processing from 250000 to 499999
I am thr #2, processing from 500000 to 749999
Parallel SUM = 49479772, process took: 0.002278 sec.   # INCORRECT RESULTS... AGAIN!
Speedup for 4 cores: 0.732705x
```

(corrected) Sum of an Array: Parallel Regions

(openMP/arraysum_region.c)

```
...
sum = 0.0;
int mythrid, loopstart, loopend;
int workperthr = N / numthr; // Each thread performs apprx N/numthr of the work
int remainder = N % numthr;  // The last thread get the remainder, if any
...

#pragma omp parallel private(mythrid,loopstart,loopend) reduction( +: sum)
{
        mythrid   = omp_get_thread_num();
        loopstart = (mythrid * workperthr);
        loopend   = (loopstart + workperthr);
        if ( mythrid == (numthr - 1) ) loopend += remainder;

        printf ("I am thr #%d, processing from %d to %d \n", mythrid, loopstart, loopend - 1);

        for ( i = loopstart; i < loopend; i++ )
                sum = sum + x[i];
}
```

```
$ export OMP_NUM_THREADS=4            # To reduce the clutter in the output below
$ ./arraysum_region
Serial   SUM = 49498583, process took: 0.001727 sec.
I am thr #0, processing from 0 to 249999
I am thr #3, processing from 750000 to 999999
I am thr #1, processing from 250000 to 499999
I am thr #2, processing from 500000 to 749999
Parallel SUM = 49498583, process took: 0.002268 sec. # CORRECT!
```

(alternative) Sum of an Array: Parallel Regions

(openMP/arraysum_region.c)

```
...
sum = 0.0;
~~int mythrid, loopstart, loopend;~~
int workperthr = N / numthr; // Each thread performs apprx N/numthr of the work
int remainder = N % numthr;  // The last thread get the remainder, if any
...

#pragma omp parallel ~~private(mythrid,loopstart,loopend)~~ reduction( +: sum)
{
        // Variables declared in a par region block are private (in C)
        int mythrid   = omp_get_thread_num();  // Declared inside the par. region, hence private
        int loopstart = (mythrid * workperthr);     // Ditto
        int loopend   = (loopstart + workperthr);   // Ditto
        if ( mythrid == (numthr - 1) ) loopend += remainder;

        printf ("I am thr #%d, processing from %d to %d \n", mythrid, loopstart, loopend - 1);

        for ( i = loopstart; i < loopend; i++ )
                sum = sum + x[i];
}
```

```
$ export OMP_NUM_THREADS=4            # To reduce the clutter in the output below
$ ./arraysum_region
Serial   SUM = 49498583, process took: 0.001731 sec.
I am thr #0, processing from 0 to 249999
I am thr #3, processing from 750000 to 999999
I am thr #1, processing from 250000 to 499999
I am thr #2, processing from 500000 to 749999
Parallel SUM = 49498583, process took: 0.002261 sec. # CORRECT!
```

Data Sharing

- **Shared (default):** Threads share a single instance of a variable.
- **Private:** Each thread creates and uses its own copy.
- **Firstprivate:** Initializes each private copy of a variable with its value before entering the parallel region/loop.
- **Lastprivate:** The value obtained in the last iteration of the parallel region/loop (regardless of which thread) is passed on the variable before leaving.

```
int i, x = 5, *array;
array = (int *) malloc (N * sizeof (int));
 // #pragma omp parallel for private(x)                    // WRONG!!
 #pragma omp parallel for firstprivate(x) lastprivate(x)  // CORRECT
      for (i = 0; i < N ; ++i) {
        array[i] = x;
        x = i;
}

printf("array[0]=%d, x=%d\n", array[0], x);
return 0;
```

(openMP/first_last_private.c)

```
$ export OMP_NUM_THREADS=4
$ ./first_last_private
array[0]=5, x=9999
```

Changing Defaults

- The default is **shared** for both C and Fortran.
- This default can be overridden by:

```
#pragma omp default(claus)
```

- "claus" can be one of the following:

```
C/C++   : private, shared, none
Fortran : private ('shared' and 'none' are not supported!)
```

- 'none' enforces listing each variable as either shared or private (or first/last private).
- For C/C++, using 'none' helps reducing programming mistakes.

Sum of an Array: Parallel "Loop"

- Individual loops can be marked as "parallel"

```
C/C++   : #pragma omp parallel for
Fortran : !$omp parallel do
```

- Data distribution is managed transparently by the compiler

(openMP/arraysum_parfor.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define N  1000000
...
...
#pragma omp parallel for reduction ( +: sum)
   for ( i = 0; i < N; i++ )
       sum += x[i];
...
...
```

```
$ export OMP_NUM_THREADS=4
$ ./arraysum_parfor
Serial   SUM = 49498583, process took: 0.001767 sec.
Parallel SUM = 49498583, process took: 0.000901 sec. // CORRECT!
Speedup for 4 cores: 1.961101x
```

Distribution of work in Parallel Loops

Programmers can influence the work distribution using the **"schedule"** clause

```
#pragma omp parallel for schedule (method [,chunk])
```

- **static [,chunk]** : Evenly distribute "chunks" of data to threads .
- **dynamic [, chunk]** : Each thread grabs a chunk until all finished.
- **guided [, chunk]** : Similar to dynamic, but blocks with start large blocks and

blocks gradually shrink down to size of "chunk" as the iteration proceeds. This might reduce the scheduling overhead.

- **runtime** : The method is determined from `OMP_SCHEDULE`

Exercise: Experiment with `arraysum_parfor.c` to find out the fastest one

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

```
export OMP_SCHEDULE="static",         => Speedup for 16 cores: 0.438531x
export OMP_SCHEDULE="static,1000"     => Speedup for 16 cores: 5.735622x
export OMP_SCHEDULE="dynamic, 10000"  => Speedup for 16 cores: 6.106666x
export OMP_SCHEDULE="guided, 10000"   => Speedup for 16 cores: 6.263389x
```

Parallel Regions: Synchronization

It is possible to create parallel regions and sub-regions with special synchronization restrictions

```
#pragma omp parallel
{
...
   #pragma omp clause
   {
      ...
   }
}
```

check the demo in:

(`openMP/synchronization_fun.c`)

"clause" can be one of the following:

- **critical:** Mutual exclusion for a block of code. All threads execute it, but only one at a time.
- **atomic:** Mutual exclusion for updating a variable, one thread at a time
- **master:** Can only be executed by the master (good for printing things out!)
- **single:** Similar to master, can be executed by "any" but "single" thread
- **barrier:** No thread can proceed until all threads arrive at the instruction where barrier is set.
- **nowait:** Some regions (e.g. 'parallel for') have implied barriers. "nowait" removes them.
- **ordered:** A sequential region in a parallel region

Task Level Parallelism

Task level parallelism can be achieved by "sections"

```
#pragma omp parallel
{
...
   #pragma omp sections
   {
      #pragma omp section
         function_1();
      #pragma omp section nowait // 'section' implies a barrier, overriding using nowait
         function_2_long();
      #pragma omp section
         function_3();
      #pragma omp section
         function_4();
   }

   // A barrier is implied here, with the exception of the thread running function_2()
}
```

Suggested web search: "**task**" construct in OpenMP 3.0 and above

OpenMP Data Dependencies

- Dependencies between iteration steps prevent data parallelism
- Compilers will NOT detect these problems, you will simply get incorrect results
- Not to be confused with instruction level dependencies

1) **flow** (read after write, RAW)

	iteration k	iteration k + d
Statement 1	write X	
Statement 2		read X

2) **anti** (write after read, WAR)

	iteration k	iteration k + d
Statement 1	read X	
Statement 2		write X

3) **output** (write after write, WAW)

	iteration k	iteration k + d
Statement 1	write X	
Statement 2		write X

OpenMP **Data** Dependencies

```
for (i = 1; i < N; i++)
{
      S1: x[i]         = 0
      S2: x[i - 1]     = tmp;
      S3: y[i]         = x[i];
      S4: z[i]         = y[i - 1];
}
```

```
S3W - S4R: anti   (WAR) on y
S1W - S2W: output (WAW) on x
S3R - S2W: flow   (RAW) on x
```

OpenMP **Data** Dependencies

```
for (i = 1; i < N; i++)
{
      S1: x[i]         = 0
      S2: x[i - 1]     = tmp;
      S3: y[i]         = x[i];
      S4: z[i]         = y[i - 1];
}
```

```
S3W - S4R: anti   (WAR) on y
S1W - S2W: output (WAW) on x
S3R - S2W: flow   (RAW) on x
```

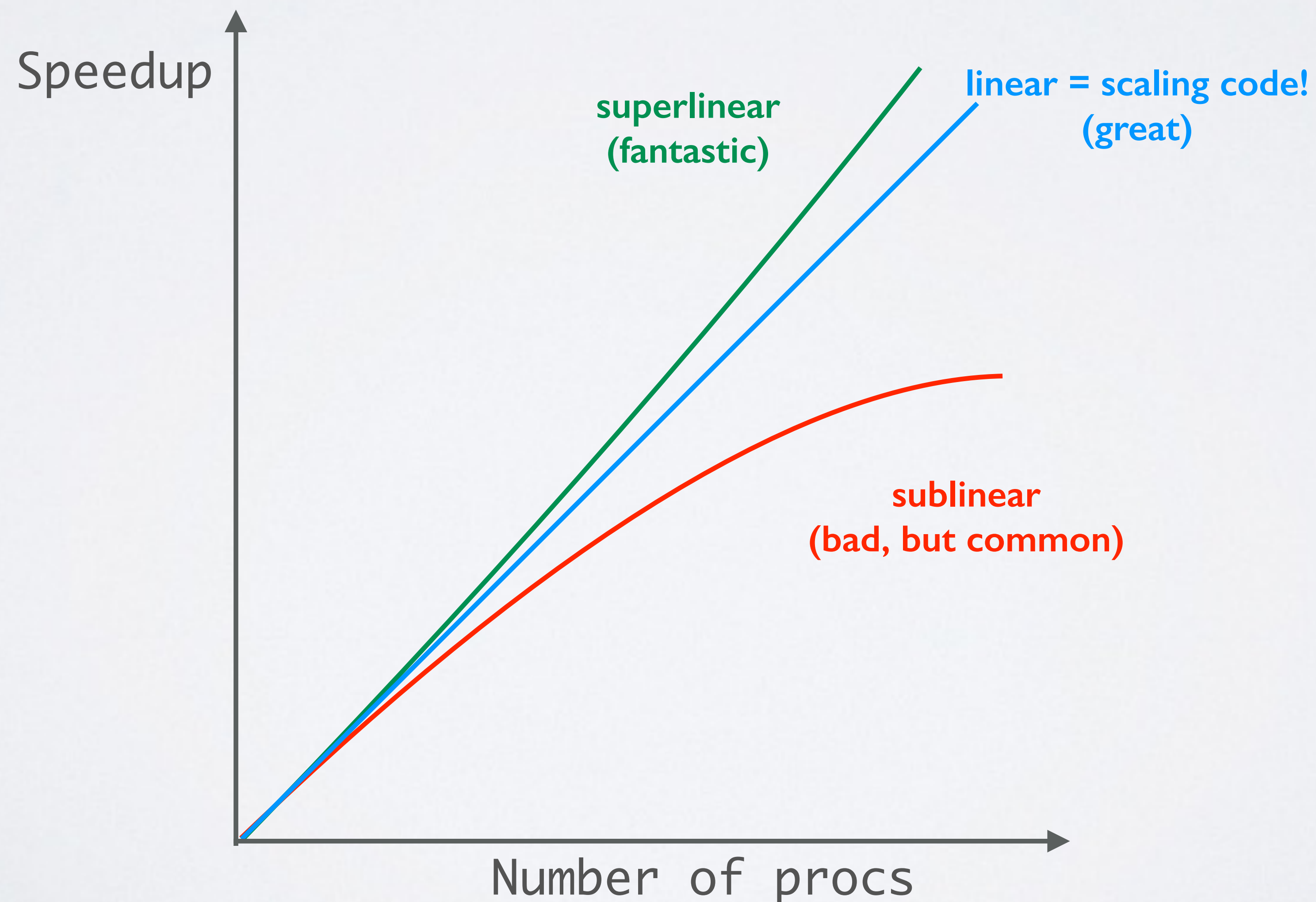
Hint: Run the loop in reverse order. If you get the same result, the loop can be parallelized safely.

Some Theoretical Background

Parallel Speedup and Efficiency

$$S = \frac{T_{seq}}{T_{par}}$$

```
S     : Speedup
Tseq  : Sequential runtime
Tpar  : Parallel runtime
```



Parallel Efficiency

$$E = \frac{S}{n} = \frac{T_{seq}}{nT_{par}}$$

E	: Efficiency
S	: Speedup
T_{seq}	: Sequential runtime
n	: Number of processors

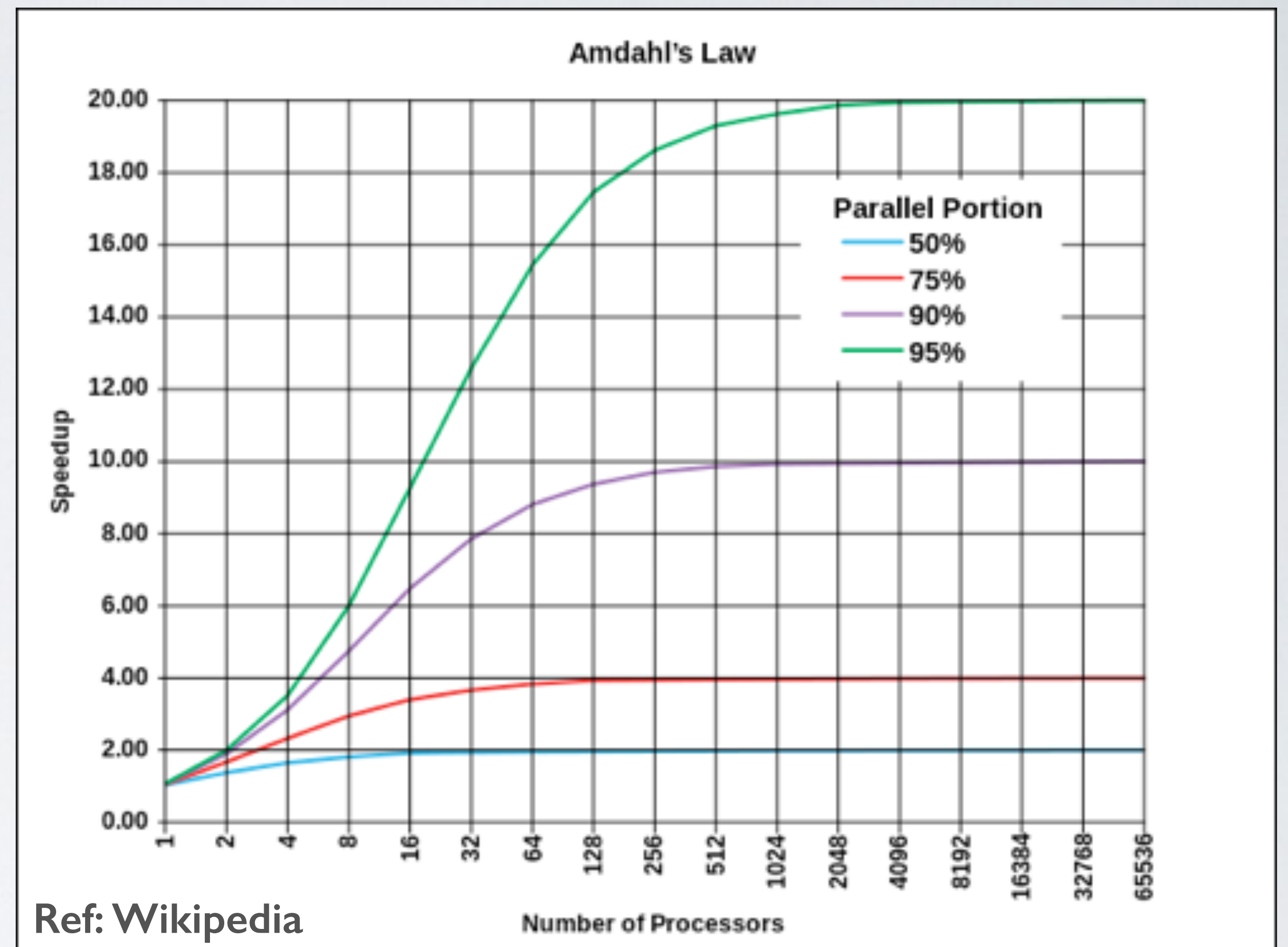
$$0 < E < 1$$

(>1 if superlinear!)

- Efficiency is a function of number of processors (n) and not a constant rate
- For difficult to parallelize algorithms, efficiency may approach to zero as n grows

Limits for Speedup (Amdahl's Law)

- Serial fraction (f_{seq}) in an algorithm limits the max speedup
- Amdahl's law defines an upper bound for parallel time T_n for a given number of processors (n)
- If an algorithm includes serial sections, the speedup will plateau after a certain number of processors.

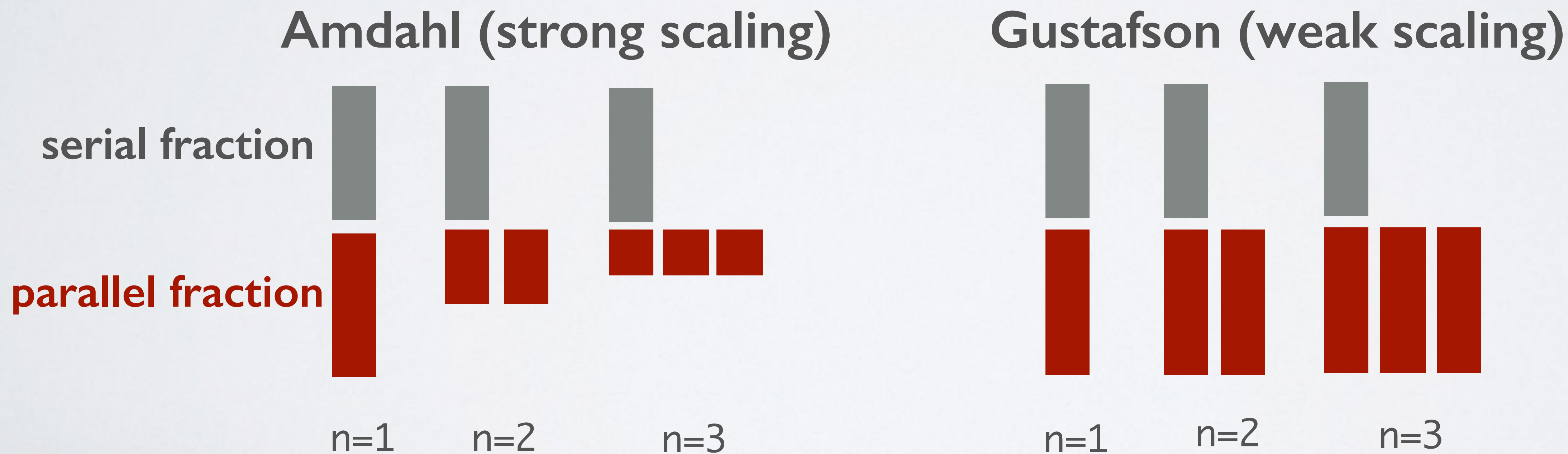


$$T_n = T_{seq}\left(f_{seq} + \frac{1 - f_{seq}}{n}\right)$$

T_n : Code execution time for (n) procs
n : Number of processors
f_{seq} : Serial fraction of the code ($0 < f_{seq} < 1$)
E.g. 30% sequential is $f_{seq} = 0.3$

Some good news (Gustafson's Law)

- Amdahl's law assumes a fixed problem size
- If we have more number of processors, why not solve larger problems?
- Gustafson redefined speedup, dubbed "scaled speedup", assuming that we increase the problem size with the growing number of processors, in a way that the work per processor is kept fixed.



f_{seq} = 0.5 (50% serial)

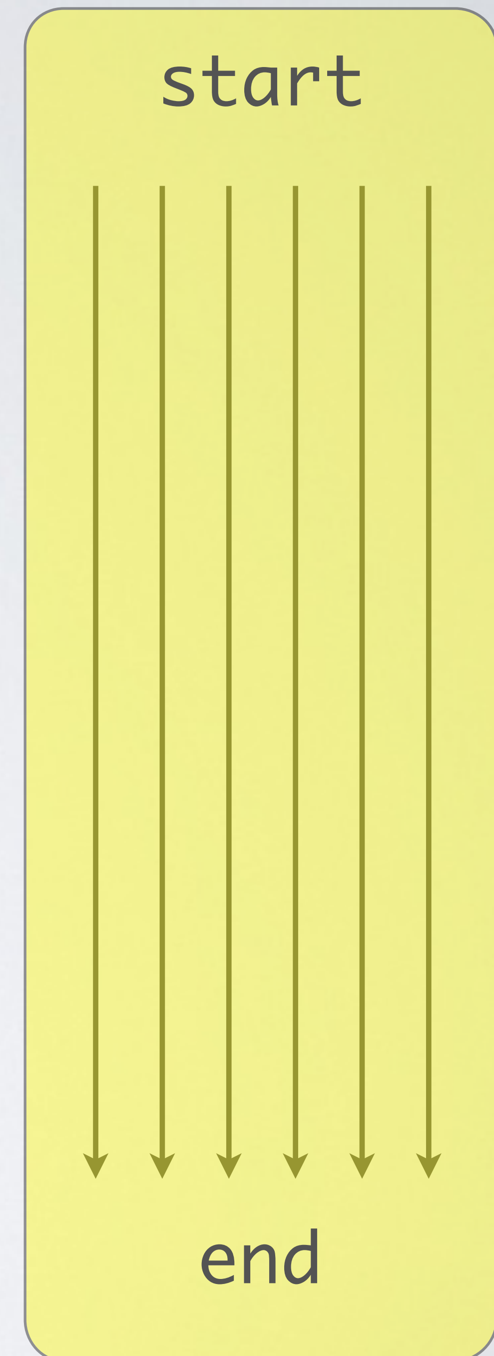
Distributed Parallelism

MPI

(**M**assage **P**assing **I**nterface)

Distributed Parallelism: MPI

- Everything is a process (as opposed to threads in openMP).
- No Forks/Joins. Starts in parallel, ends in parallel.
- A copy of the code is distributed to each process and all processes run the very same code!
- Processes are identified using unique IDs.
- Installed as extensions (libraries) on C, C++, Fortran, Python, Java
- Processes cannot access each other's data, so they "exchange messages", hence the name "message passing".
- Each process can communicate with any other processes in the pool. If they are in the same node, they use the memory bus (fast). If they are on separate nodes, they use the network (slower). This is completely transparent from the programmer.



Distributed Parallelism: MPI

PACE maintains three MPI Stacks:

- **MVAPICH2:** Only for Infiniband (IB) Networks! Developed by Ohio State University
- **OpenMPI:** Works on IB and Ethernet, developed as a collaboration by many members
- **iMPI:** Provided by Intel, commercial product.

In theory, an MPI code can be compiled using any of these stacks. In practice, there can be compatibility issues depending on the stack and/or versions.

Compilation: Use wrappers for compilers (mpicc, mpic++, mpif90)

```
module load intel          #  (or GCC, PGI, etc)
module load mvapich2       #  (or openMPI)
mpicc -o prog.exe prog.c
mpif90 -o prog.exe prog.f
```

Running: "mpirun" distributes and starts the code on all participating processes

```
mpirun -np NUMPROC prog.exe  # NUMPROC = number of processes in the pool
```

Basic MPI Routines

MPI Communicator: A group of processes that can communicate with each other. Default communicator: "`MPI_COMM_WORLD`" which contains all of the processes.

- **MPI_Init:** Initializes the MPI environment

```
C/C++  : MPI_Init (&argc, &argv)
Fortran: MPI_INIT (ierr)
```

- **MPI_Comm_size:** Returns the number of MPI processes in the communicator

```
C/C++  : MPI_Comm_size (comm, &size)
Fortran: MPI_COMM_SIZE (comm,size,ierr)
```

- **MPI_Comm_rank:** Returns the rank (process ID) of the calling process

```
C/C++  : MPI_Comm_rank (comm, &rank)
Fortran: MPI_COMM_RANK (comm, rank, ierr)
```

- **MPI_Finalize:** Finalizes the MPI execution in the code, synchronizes all processes.

```
C/C++  : MPI_Finalize ()
Fortran: MPI_FINALIZE (ierr)
```

MPI "Hello Wold" Example

Simple MPI code: Hello world

```c
/* C Example */
#include <stdio.h>
#include <mpi.h>

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

/* Initialize MPI */
  MPI_Init (&argc, &argv);

/* Get current process ID (rank) */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

/* Get number of processes s */
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  printf( "Hello from proc %d of %d\n", rank, size );

/* Finalize MPI */
  MPI_Finalize();
  return 0;
}
```

(MPI/hello_world.c)

```fortran
C   Fortran example
    program HelloWorld
    use mpi
    integer rank, size, ierror, tag
    integer status(MPI_STATUS_SIZE)

C   Initialize MPI
    call MPI_INIT(ierror)

C   Get current process ID (rank)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)

C   Get number of processes
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

    print*, 'Hello from proc', rank, 'of', size

C   Finalize MPI
    call MPI_FINALIZE(ierror)
    end
```

(MPI/hello_world.f)

Compiling and Running Examples

Simple MPI code: Hello world

- Get a compute node on one of the PACE clusters

```
$ msub -I -q iw-shared-6 -l nodes=4,pmem=2gb,walltime=30:00
```

- When given an interactive session, load modules and compile the codes

```
$ cd MPI
$ source load_modules
$ make all       # Check the mpicc/mpif90 statements in the Makefile
```

- Run the code(s) using mpirun

```
$ mpirun -np 4 ./hello_world_c      # or ‘hello_world_f’

  Hello from proc 3 of 4
  Hello from proc 0 of 4
  Hello from proc 1 of 4
  Hello from proc 2 of 4
```

Message Passing (send/receive)

- **MPI_Send:** Sends a package of data to another process

```
C/C++  : MPI_Send (&buf, count, datatype, dest, tag, comm)
Fortran: MPI_SEND (buf, count, datatype, dest, tag, comm, ierr)
```

```
buf (const void *)         : The address of the data to be sent
count (int)                : the number of data items
datatype (MPI_Datatype)    : "MPI-standardized" data type (MPI_INT, MPI_FLOAT…)
dest (int)                 : The rank (process ID) to which data is sent
tag (int)                  : A unique identifier (int), which should match the corresponding receive
comm (int)                 : the name of the communicator (MPI_COMM_WORLD)
```

- **MPI_Recv:** Receives a package of data sent by another process

```
C/C++  : MPI_Recv (&buf, count, datatype, source, tag, comm, &status)
Fortran: MPI_RECV (buf, count, datatype, source, tag, comm, status, ierr)
```

```
count, datatype, tag, communicator  : Same as above (see MPI_send)
buf  (void *)                       : The address where the received data will be written
source (int)                        : The rank (process ID) which sends the received data
status (MPI_Status )                : Diagnostic information
```

Matching send & receive:

```
MPI_Send (&buf, count, datatype, dest, tag, comm)
MPI_Recv (&buf, count, datatype, source, tag, comm, &status)
```

Notes for Fortran

- All MPI routines in Fortran (except for `MPI_WTIME` and `MPI_WTICK`) have an additional argument `ierr` at the end of the argument list. `ierr` is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.

- All MPI objects (e.g., `MPI_Datatype`, `MPI_Comm`) are of type `INTEGER` in Fortran.

- The status argument must be declared as an array of size `MPI_STATUS_SIZE`, as in:

```
integer status(MPI_STATUS_SIZE)
```

MPI Standard Datatypes

MPI	C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI	FORTRAN
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER

```
MPI_Send (&buf, count, datatype, dest, tag, comm)
MPI_Recv (&buf, count, datatype, source, tag, comm, &status)
```

Congratulations!

You learned MPI.

In theory, it is possible to write a MPI code using only these 6 routines:

- `MPI_Init`
- `MPI_Finalize`
- `MPI_Comm_size`
- `MPI_Comm_rank`
- `MPI_Send`
- `MPI_Recv`

Of course, we don't stop there :-)

More MPI Routines

- **MPI_Barrier:** Synchronizes all of the processes in the MPI Communicator. Must be called by ALL processes, otherwise the code hangs!

```
C/C++  : int MPI_Barrier( MPI_Comm comm )
Fortran: MPI_BARRIER(COMM, IERR)
```

- **MPI_Wtime:** Returns an elapsed time since an arbitrary time in the past

```
C/C++  : double MPI_Wtime()
Fortran: DOUBLE PRECISION MPI_WTIME()  # No IERR this time!
```

```
t1 = MPI_Wtime();
    ... (some computation)
t2 = MPI_Wtime();
elapsed_time = t2 - t1;
```

- **MPI_Abort:** Aborts an MPI program (e.g. on error). Different from MPI_Finalize!

```
C/C++  : int MPI_Abort(MPI_Comm comm, int errorcode)
Fortran: MPI_ABORT(COMM, ERRORCODE, IERROR)
```

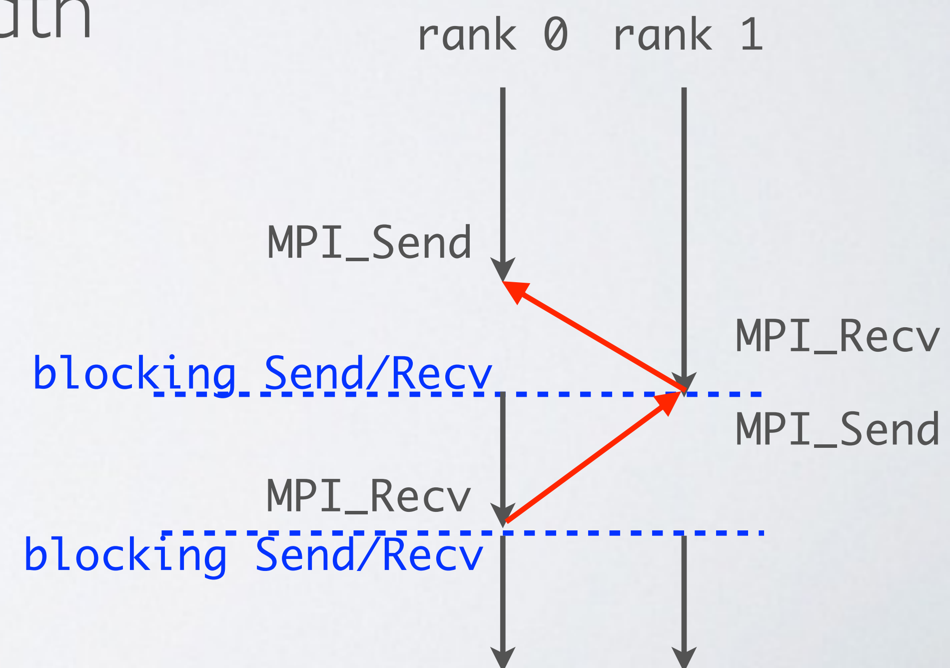
MPI "PingPong" Example

See: `MPI/pingpong.c`

- Runs using 2 processors
- At each step, each processes send a package to each other

  (`0->1 then 0<-1`)

- The package size is incremented at each step of a loop
- Measures the latency (size = 0) and bandwidth
- Demonstrates the following:
  - Using rank 0 (master) for printing on the screen
  - Synchronization (MPI_Barrier)
  - Measuring time (walltime, MPI_Wtime)
  - Checking for conditions and abort when necessary



MPI "PingPong" Example

(MPI/pingpong.c)

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

// The largest array size to exchange
#define MAXARRAYSIZE 1073741824
#define INCREMENTS   16777216

int main (int argc, char *argv[])
{
    int i, rank, size, test_size;
    double package_array[MAXARRAYSIZE];

    // Initialize MPI
    MPI_Init (&argc, &argv);
    // Get current process ID (rank)
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    // Get number of processes
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    // We only need 2 processes for this task. Only the 'master' node (rank == 0) performs this check
    if (rank == 0) {
        if (size != 2) {
            printf ("Please run with 2 procs only.\n");
            MPI_Abort (MPI_COMM_WORLD, 1);
        }
        // Fill-in (initialize) the array
        printf("Please wait while the master fills-in the array.\n");
        for (i = 0; i < MAXARRAYSIZE; i++)
            package_array[i] = (double) i;

    }
    MPI_Barrier (MPI_COMM_WORLD);
```

(... continues on the next slide)

MPI "PingPong" Example

(MPI/pingpong.c)

```
int pingpong, from = 0, to = 1;
double now  = 0, then = 0;

for (test_size=0; test_size < MAXARRAYSIZE; test_size += INCREMENTS)
{
    //Convert the array size to MB
    double size_in_MB=((double)test_size * sizeof(double))/1024/1024;
    if ( rank == 0 )
        printf ("=== Testing size: %d, (%7.2f MB) ===\n", test_size, size_in_MB);

    // Repeat twice. ping (->) & pong (<-)
    for (pingpong = 0; pingpong < 2; ++pingpong) {
        if ( rank == from ) {
            now = MPI_Wtime();
               MPI_Send(package_array, test_size, MPI_DOUBLE, to, 0, MPI_COMM_WORLD);
            then = MPI_Wtime();
            double time_spent = then - now;
            printf ("%d -> %d, size=%d, time/element = %e\n", rank, to, test_size, time_spent);
        } else {
            now = MPI_Wtime();
               MPI_Recv(package_array, test_size, MPI_DOUBLE, from, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            then = MPI_Wtime();
            double time_spent = then - now;
            printf ("%d <- %d, size=%d, time/element = %e\n", rank, from, test_size, time_spent);
        }
        from = 1 - from; // flip 'to' and 'from' (1,0,1,0,...)
         to  = 1 - to;   //  ditto
    }
    // Flush the output buffer before running the next iteration (for tidy output)
    fflush(stdout);
}

MPI_Finalize();
```

MPI "PingPong" Example

```
$ source load_modules   # Load the same set of modules that you used to compile the codes!
$ ulimit -s unlimited   # This is required, since the default is 10MB!

$ mpirun -np 2 ./pingpong
Please wait while the master fills-in the array.
=== Testing size: 0, (   0.00 MB) ===
(ping) 0 -> 1, size=0, time/element = 6.198883e-05
(pong) 1 <- 0, size=0, time/element = 3.390312e-04
(ping) 1 -> 0, size=0, time/element = 3.004074e-05
(pong) 0 <- 1, size=0, time/element = 3.561974e-04
=== Testing size: 16777216, ( 128.00 MB) ===
(ping) 0 -> 1, size=16777216, time/element = 2.230930e-01
(pong) 1 <- 0, size=16777216, time/element = 2.241111e-01
(ping) 1 -> 0, size=16777216, time/element = 1.552320e-01
(pong) 0 <- 1, size=16777216, time/element = 1.571040e-01
=== Testing size: 33554432, ( 256.00 MB) ===
(ping) 0 -> 1, size=33554432, time/element = 3.505630e-01
(pong) 1 <- 0, size=33554432, time/element = 3.536661e-01
(ping) 1 -> 0, size=33554432, time/element = 3.364141e-01
(pong) 0 <- 1, size=33554432, time/element = 3.385811e-01
=== Testing size: 50331648, ( 384.00 MB) ===
(ping) 0 -> 1, size=50331648, time/element = 6.451380e-01
(pong) 1 <- 0, size=50331648, time/element = 6.462100e-01
(ping) 1 -> 0, size=50331648, time/element = 4.994631e-01
(pong) 0 <- 1, size=50331648, time/element = 5.006590e-01
=== Testing size: 67108864, ( 512.00 MB) ===
(ping) 0 -> 1, size=67108864, time/element = 8.525629e-01
(pong) 1 <- 0, size=67108864, time/element = 8.543918e-01
(ping) 1 -> 0, size=67108864, time/element = 6.741462e-01
(pong) 0 <- 1, size=67108864, time/element = 6.760552e-01
=== Testing size: 83886080, ( 640.00 MB) ===
(ping) 0 -> 1, size=83886080, time/element = 1.295615e+00
(pong) 1 <- 0, size=83886080, time/element = 1.296800e+00
…
…
(truncated)
```

MPI Wildcards and 'status'

- MPI includes a set of 'wildcards' for sources and tags

  `MPI_ANY_SOURCE`: Used in MPI_Recv, allows for receiving from any sender process

  `MPI_ANY_DEST` : Used in MPI_Send, allows for sending to any receiver process

  `MPI_ANY_TAG` : Used in MPI_Recv, allows for receiving any message, regardless of their tag

- These tags can be used together
  (`MPI_Recv()` with `MPI_ANY_SOURCE` and `MPI_ANY_TAG`)

- The 'status' variable can be used to discover which sender/tags were used for a received message at runtime. Can be ignored using "`MPI_STATUS_IGNORE`".

```
FORTRAN : source = status(MPI_SOURCE)
          tag = status(MPI_TAG)

C       : source = status.(MPI_SOURCE);
          tag = status.(MPI_TAG);

C++     : source = status.Get_source();
          tag = status.Get_tag();
```

Asynchronous Communication

- Non-blocking MPI_send/MPI_recv (often as an optimization technique where beneficial)
- A communication can be blocking on one end and asynchronous on the other.
- MPI_send and MPI_recv return **immediately**, regardless of the status. Testing is done manually.
- Takes an additional parameter: "**request**"

MPI_Isend: Asynchronously send a package of data to another process

```
C/C++  : MPI_Isend (&buf, count, datatype, dest, tag, comm, &request)
Fortran: MPI_ISEND (buf, count, datatype, dest, tag, comm, request, ierr)
        request (MPI_Request *)  : so-called 'opaque object', communication operation identifier
```

MPI_Irecv: Asynchronously receive a package of data sent by another process

```
C/C++  : MPI_Irecv (&buf, count, datatype, source, tag, comm, &request, &status)
Fortran: MPI_IRECV (buf, count, datatype, source, tag, comm, status, request,ierr)
```

MPI_Wait: make your process hang until the operation identified by the request is complete.

```
C/C++  : MPI_Wait (&request, &status)
Fortran: MPI_WAIT (request, status, ierr)
```

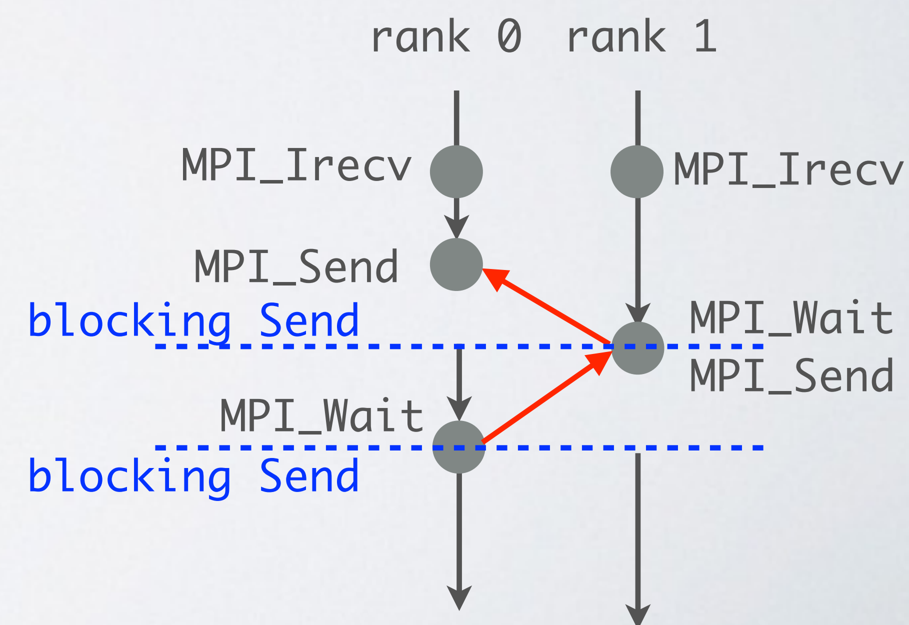
MPI_Test: check for comm success, then do something else, then check again, in a loop. Does NOT block.

```
C/C++  : MPI_Test (&request, &flag, &status)
Fortran: MPI_TEST (request, flag, status, ierr)
        flag (int *) for C/C++  and (logical) for Fortran :
```

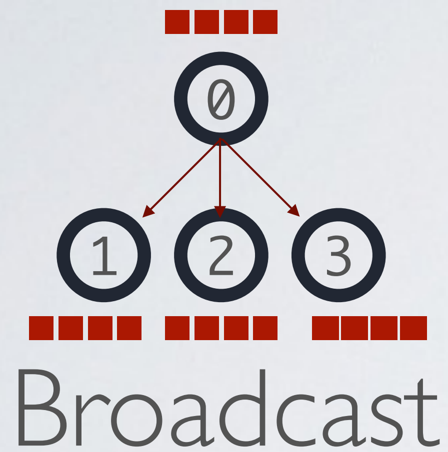
Asynchronous "PingPong" Example

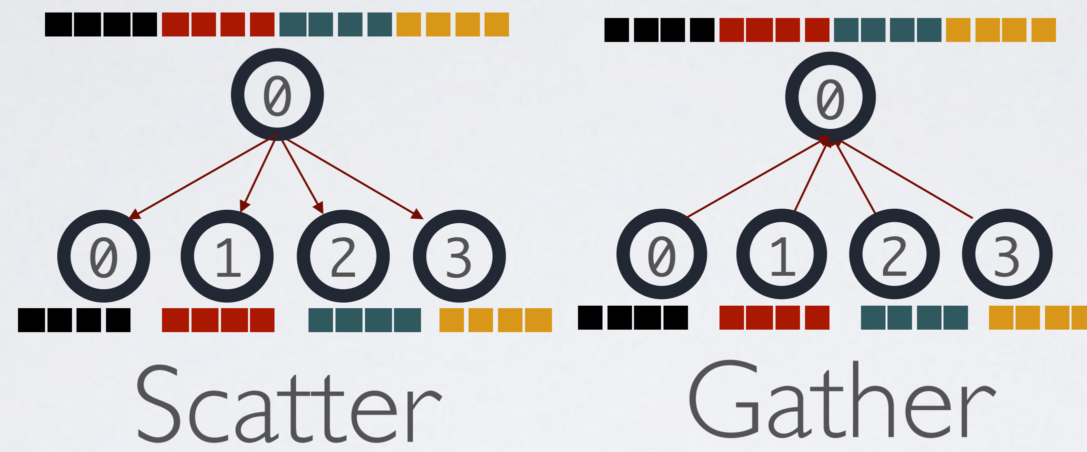
See: `MPI/async_pingpong.c`

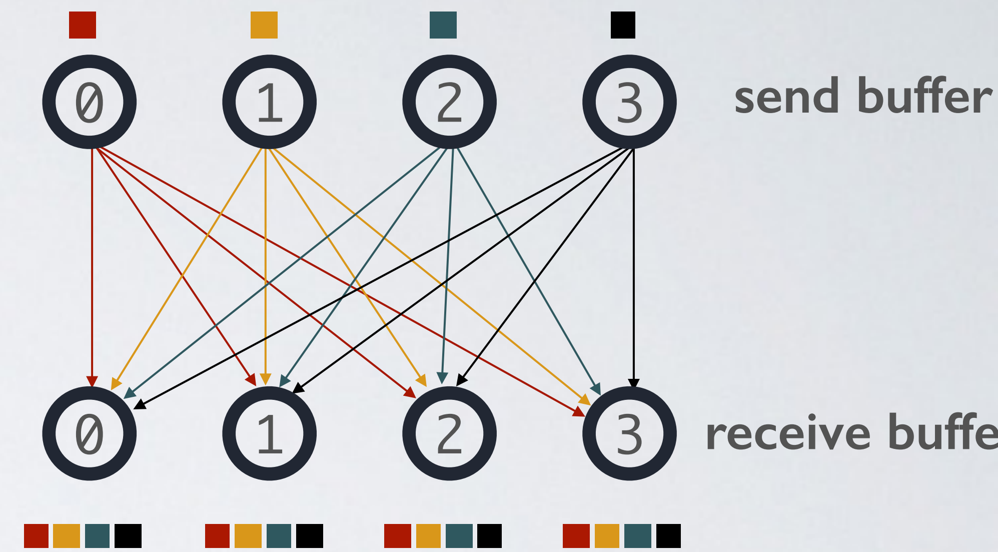
- Runs using 2 processors
- Both processes can make an async receive calls right away
- Process #0 sends the packages, which is blocking
- Process #1 waits until it receives the message, then sends it back
- Process #0 waits until it received the message
- Demonstrates the following:
  - mixed use of blocking and asynchronous communication
  - not blocking on receives, but still blocking on MPI_Wait
  - only beneficial if there are other tasks to do while waiting



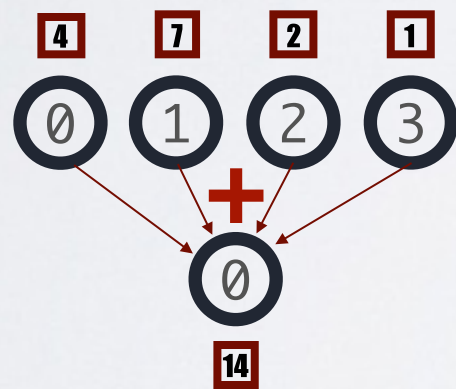
MPI Collective Communications

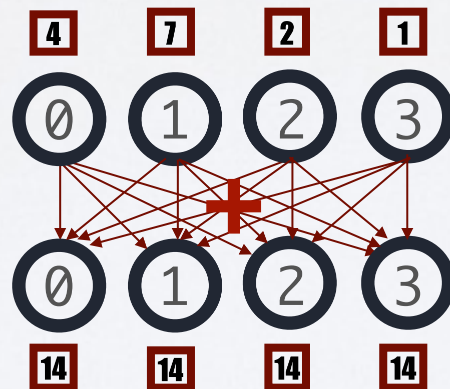






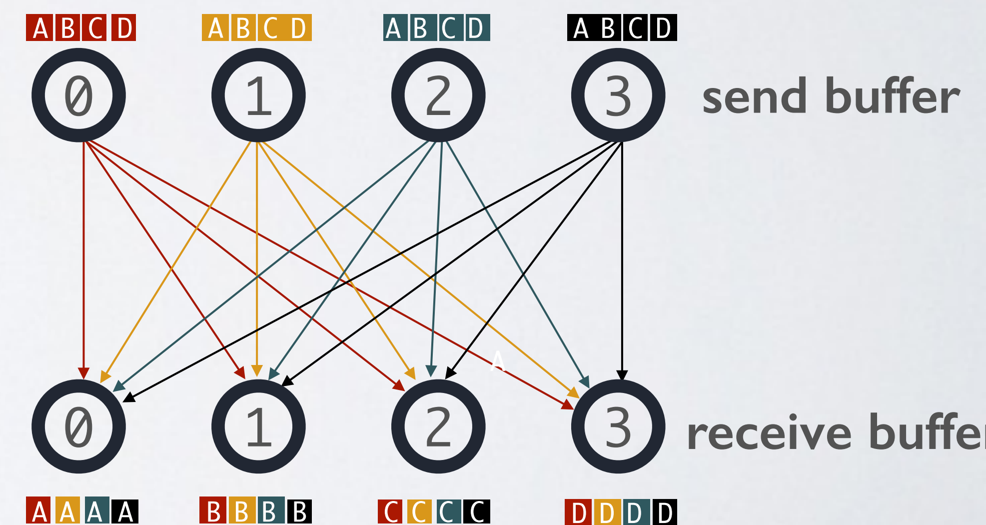
All Gather





Reduce

Allreduce



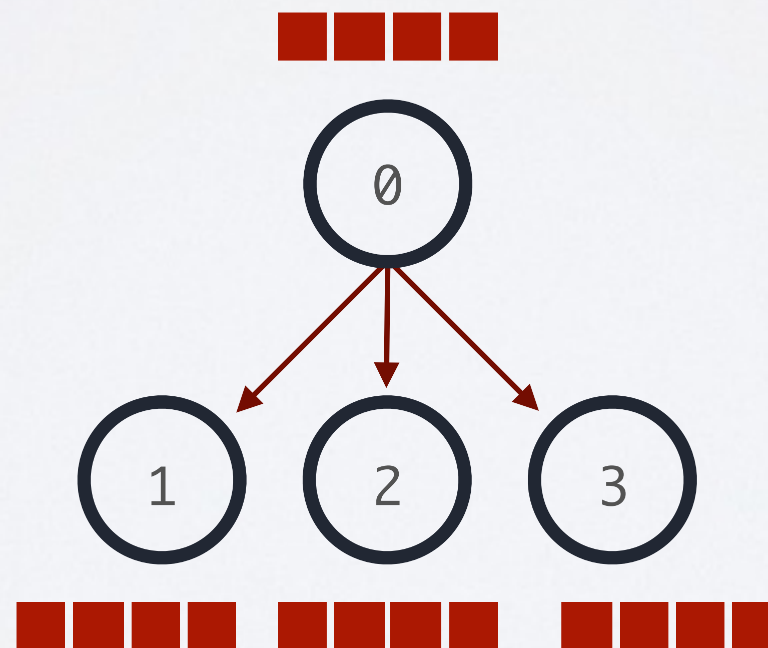
All to All

MPI Broadcast

MPI_Bcast: Sends a message from a source to all other processes

```
C/C++  : MPI_Bcast (&buf, count, datatype, source, comm)
Fortran: MPI_BCAST (buf, count, datatype, source, comm, ierr)
```

```
buf (const void *)         : The address of the data to be sent/received
count (int)                : the number of data items
datatype (MPI_Datatype)    : "MPI-standardized" data type (MPI INT, MPI FLOAT...)
source (int)               : The rank (process ID) from which message is sent
comm (int)                 : the name of the communicator (MPI COMM WORLD)
```

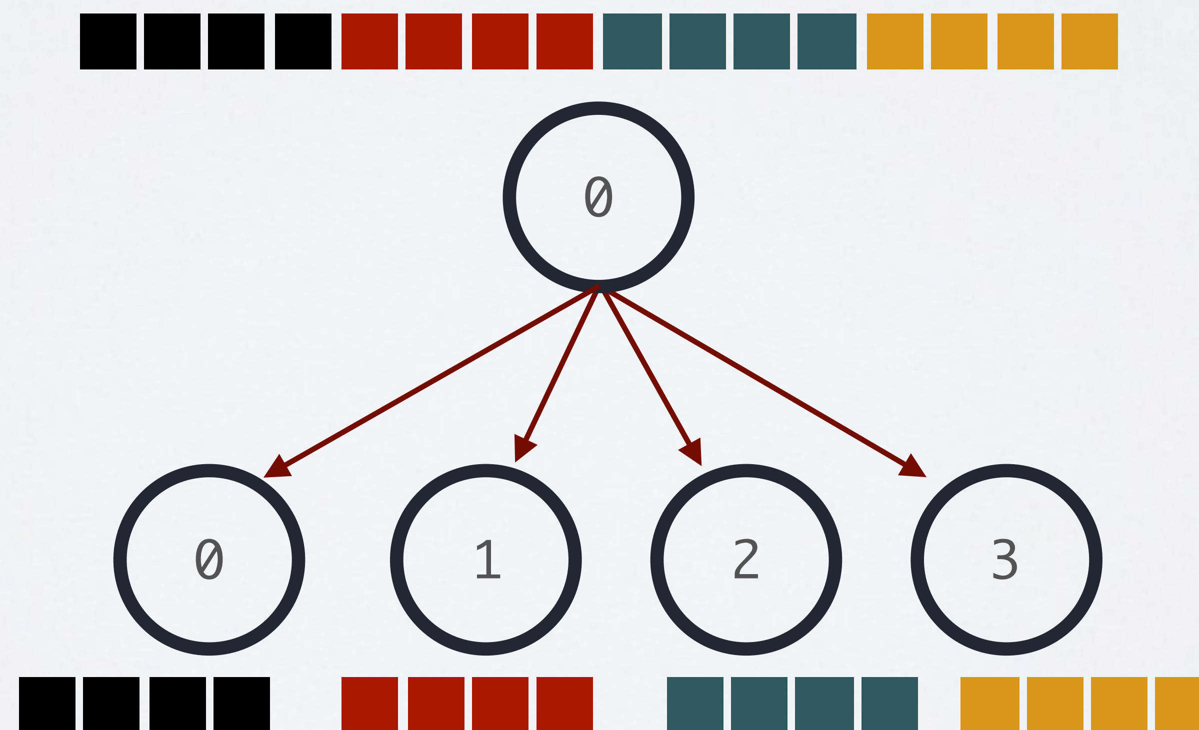


MPI Scatter

MPI_Scatter: Sends chunks of messages from a source to other processes

```
C/C++  : MPI_Scatter (&send_buf, send_count, send_datatype, &recv_buf, recv_count, recv_datatype, source, comm)
Fortran: MPI_SCATTER (send_buf, send_count, send_datatype, recv_buf, recv_count, recv_datatype, source, comm, ierr)
```

```
send/recv_buf (const void *)         : The address of the data to be sent/received
send/recv_count (int)                : the number of data items to send/receive
send/recv_datatype (MPI_Datatype)    : “MPI-standardized” data type (MPI INT, MPI FLOAT...)
source (int)                         : The rank (process ID) from which message chunks are sent
comm (int)                           : the name of the communicator (MPI COMM WORLD)
```

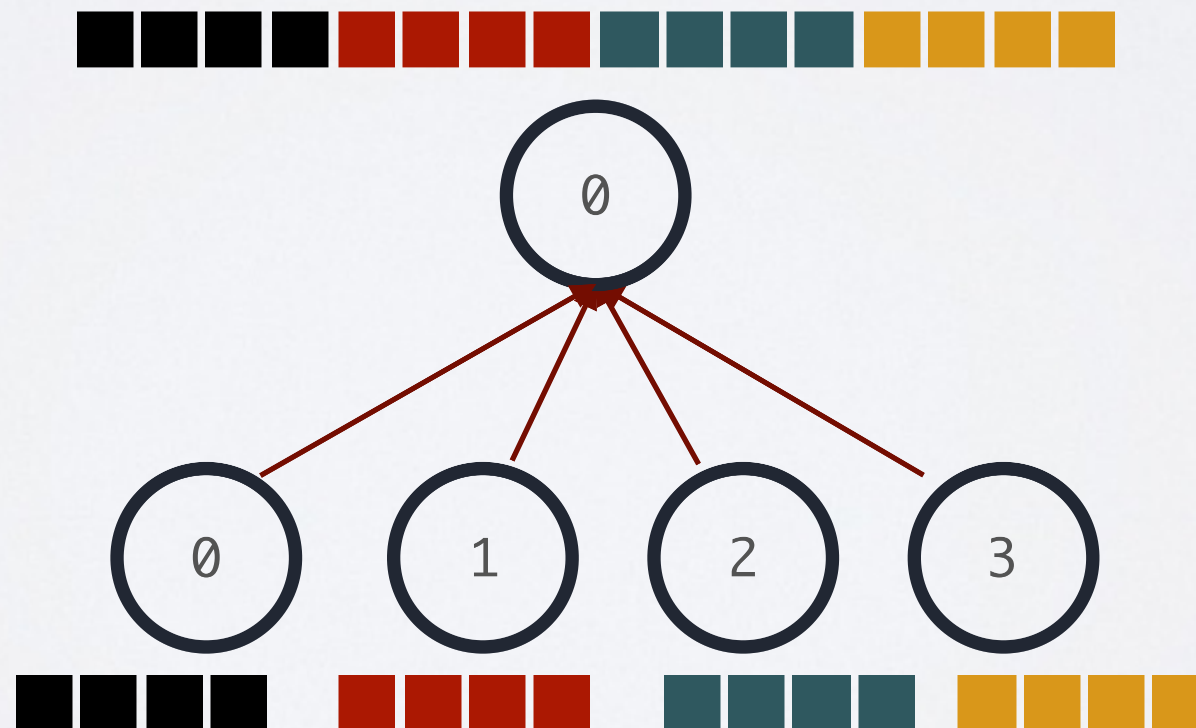


MPI Gather

MPI_Gather: Gather chunks of messages from a group of processes

```
C/C++  : MPI_Gather (&send_buf, send_count, send_datatype, &recv_buf, recv_count, recv_datatype, target, comm)
Fortran: MPI_GATHER (send_buf, send_count, send_datatype, recv_buf, recv_count, recv_datatype, target, comm, ierr)
```

```
send/recv_buf (const void *)         : The address of the data to be sent/received
send/recv_count (int)                : the number of data items to send/receive
send/recv_datatype (MPI_Datatype)    : “MPI-standardized” data type (MPI INT, MPI FLOAT...)
target (int)                         : The rank (process ID) which message chunks are sent to
comm (int)                           : the name of the communicator (MPI COMM WORLD)
```

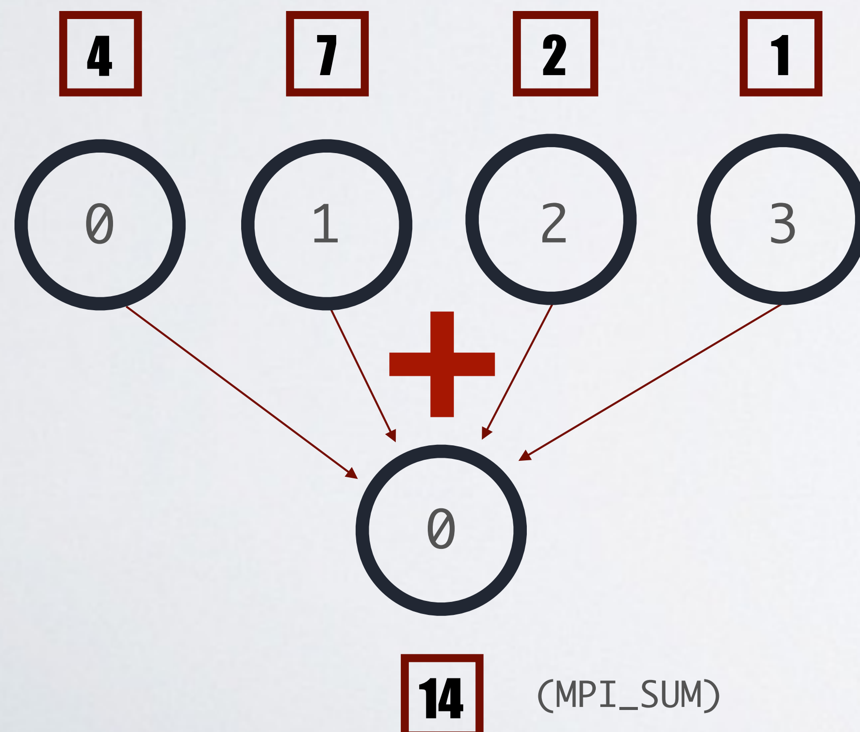


MPI Reduce

MPI_Reduce: Reduces values on all processes to a single value, using a given operator

```
C/C++  : MPI_Reduce (&send_buf, &recv_buf, count, data_type, operator, target, comm)
Fortran: MPI_REDUCE (send_buf, recv_buf, count, data_type, operator, target, comm, ierr)
```

```
send/recv_buf (const void *)       : The address of the data to be sent/reduced
count (int)                        : the number of data items to be reduced on
datatype (MPI_Datatype)            : “MPI-standardized” data type (MPI INT, MPI FLOAT...)
target (int)                       : The rank (process ID) which holds the reduce target
operator (MPI_Op)                  : MPI operator (see table given below)
comm (int)                         : the name of the communicator (MPI COMM WORLD)
```



MPI Reduce Operators

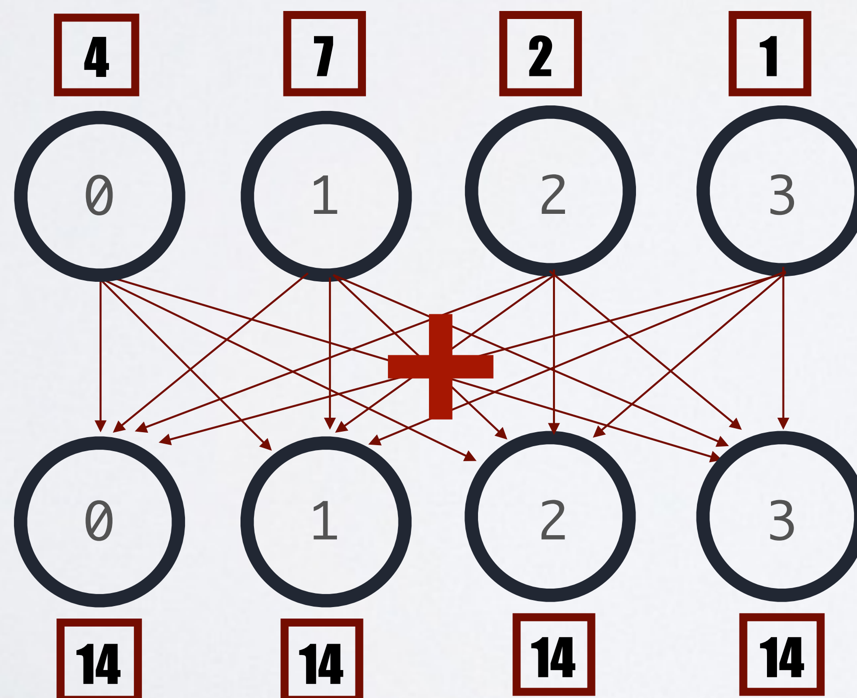
Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

MPI Allreduce

MPI_Reduce: Reduces values on all processes and distributes the result to all processes

```
C/C++  : MPI_Allreduce (&send_buf, &recv_buf, count, data_type, operator, comm)
Fortran: MPI_ALLREDUCE (send_buf, recv_buf, count, data_type, operator, comm, ierr)
```

```
send/recv_buf (const void *)       : The address of the data to be sent/reduced
count (int)                        : the number of data items to be reduced on
datatype (MPI_Datatype)            : “MPI-standardized” data type (MPI INT, MPI FLOAT...)
operator (MPI_Op)                  : MPI operator (see table given below)
comm (int)                         : the name of the communicator (MPI COMM WORLD)
```

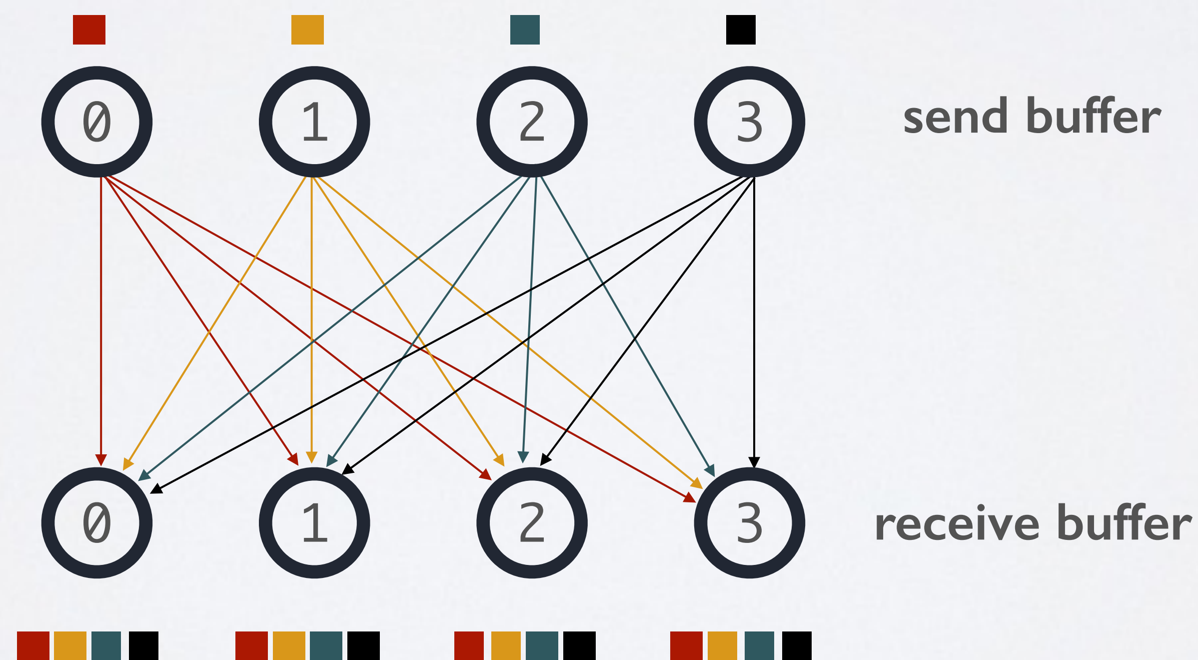


MPI Allgather

MPI_Allgather: Similar to MPI_Gather operation, but all processes receive the result

```
C/C++  : MPI_Allgather (&send_buf, send_count, send_datatype, &recv_buf, recv_count, recv_datatype, comm)
Fortran: MPI_ALLGATHER (send_buf, send_count, send_datatype, recv_buf, recv_count, recv_datatype, comm, ierr)
```

```
send/recv_buf (const void *)         : The address of the data to be sent/received
send/recv_count (int)                : the number of data items to send/receive
send/recv_datatype (MPI_Datatype)    : “MPI-standardized” data type (MPI INT, MPI FLOAT...)
comm (int)                           : the name of the communicator (MPI COMM WORLD)
```

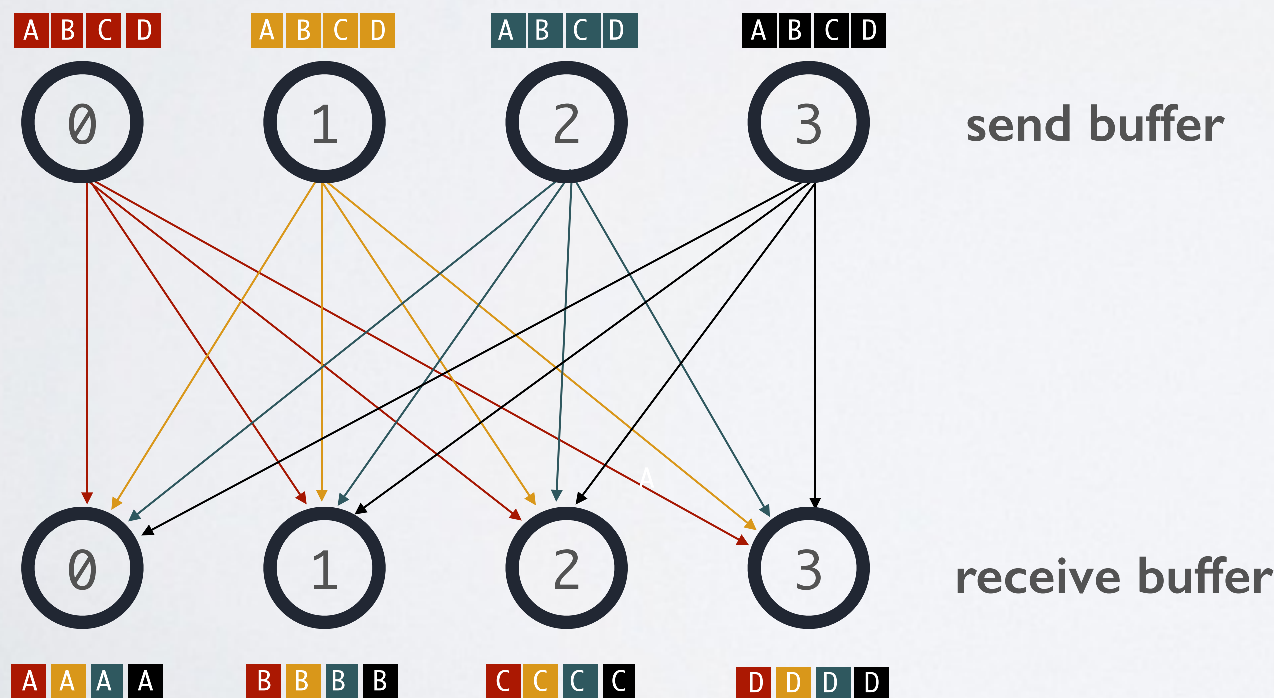


MPI All-to-All

MPI_Alltoall: Extension of MPI_Allgather, but each processes send/receive distinct data

```
C/C++  : MPI_Alltoall (&send_buf, send_count, send_datatype, &recv_buf, recv_count, recv_datatype, comm)
Fortran: MPI_ALLTOALL (send_buf, send_count, send_datatype, recv_buf, recv_count, recv_datatype, comm, ierr)
```

```
send/recv_buf (const void *)         : The address of the data to be sent/received
send/recv_count (int)                : the number of data items to send/receive
send/recv_datatype (MPI_Datatype)    : "MPI-standardized" data type (MPI INT, MPI FLOAT...)
comm (int)                           : the name of the communicator (MPI COMM WORLD)
```



send buffer

receive buffer

Perfect for Matrix Transpose!

rank	send_buf	recv_buf
0	a0,b0,c0,d0	a0,a1,a2,a3
1	a1,b1,c1,d1	b0,b1,b2,b3
2	a2,b2,c2,d2	c0,c1,c2,c3
3	a3,b3,c3,d3	d0,d1,d2,d3

MPI Allscatter ?

No such thing.

MPI Collectives Example

(MPI_collectives.c)

STEP 1: Root broadcasts the matrix size "N" to all processes

STEP 2: All processes allocate memory for their local row(N), tr_row(N) and tr_matrix(NxN)

STEP 3: Root initializes its local NxN matrix

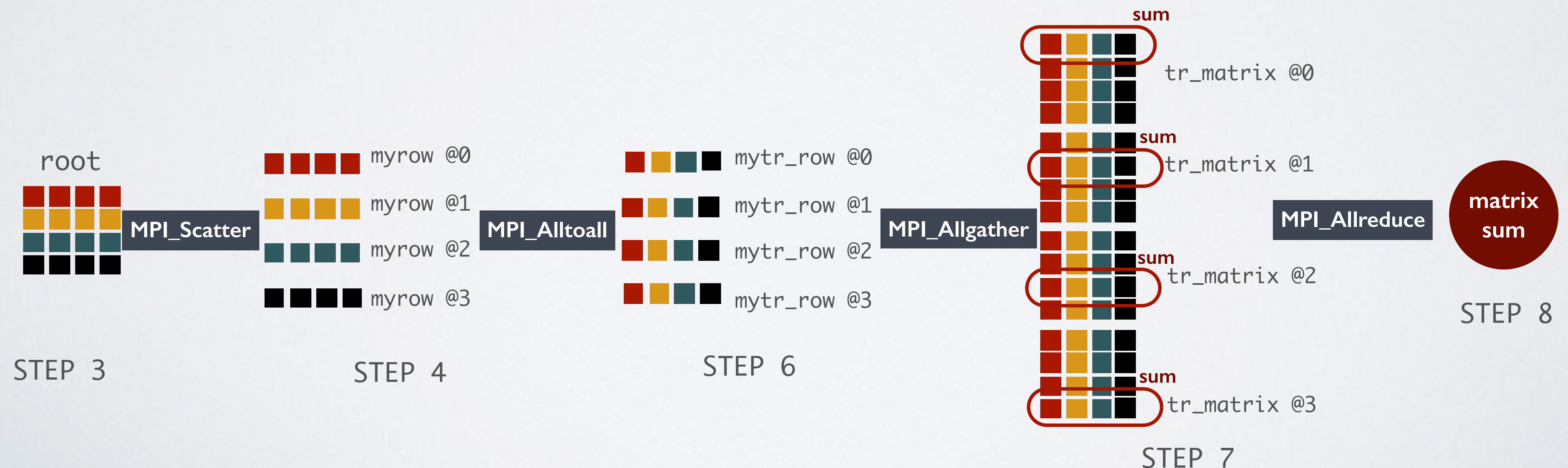
STEP 4: Root scatters each matrix row to the process with a rank matching the row number

STEP 5: Each process update their row. We will simply add '1' to each value

STEP 6: Processes All-to-all to exchange their row elements (which will compose the transposed matrix later)

STEP 7: Processes Allgather the rows to construct a local copy for the transposed matrix

STEP 8: All processes first take a sum of their local rows, then Allreduce (sum) on matrixsum to find the result



MPI Collectives with varying counts and displacements

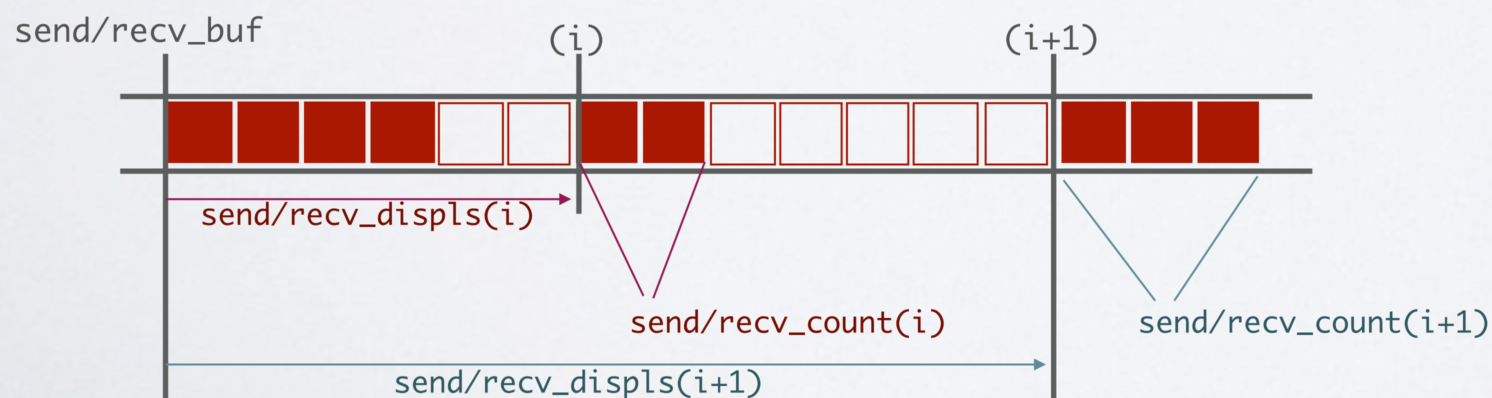
MPI_Gatherv (&send_buf, send_count, send_datatype, &recv_buf, recv_counts, recv_displs, source, recv_datatype, comm)

MPI_Scatterv (&send_buf, send_counts, send_displs, send_datatype, &recv_buf, recv_count, recv_datatype, source, comm)

MPI_Alltoallv (&send_buf, send_counts, send_displs, send_datatype, &recv_buf, recv_counts, recv_displs, recv_datatype, comm)

MPI_Allgatherv (&send_buf, send_count, send_datatype, &recv_buf, recv_counts, displs, recv_datatype, comm)

```
send/recv_counts (const int *)      : Array specifying the send/receive counts
send/recv_displs (const int *)      : Array specifying the displacement relative to send/recv_buf
```



What's next?

- Install OpenMP/MPI on your local machine, which almost certainly has multiple cores.

- Find fun examples and start coding them! A few fun projects:

  - Estimate Pi in parallel (you already know the correct answer)
  - Conway's Game of life (a websearch will return a lot of info)

- If you don't have a PACE cluster account, ask for one!

  `http://pace.gatech.edu/node/add/request`

- Take a look at the parallel profiling/debugging slides

  `http://pace.gatech.edu/workshop/DebuggingProfiling.pdf`

Thank You!

Your feedback will be appreciated! (mehmet.belgin@oit.gatech.edu)

- I need your brutally honest feedback so I can improve this class
- We *might* send you a survey later, and any comment will help.

Have More Time?