Introduction to parallel and high-performance computing (part II)

Simon Scheidegger
simon.scheidegger@gmail.com
Jan. 30th, 2017
University of Zürich

Including adapted teaching material from books, lectures and presentations by
B. Barney, B. Cumming, G. Hager, R. Rabenseifner, O. Schenk, G. Wellein

Roadmap – fast forward:

Day 1, Saturday – Jan 28th (14:00-15.30)

1. Introduction to parallel programming and high-performance computing (HPC).
2. Introduction 'Unix-like' HPC environments (hands on).

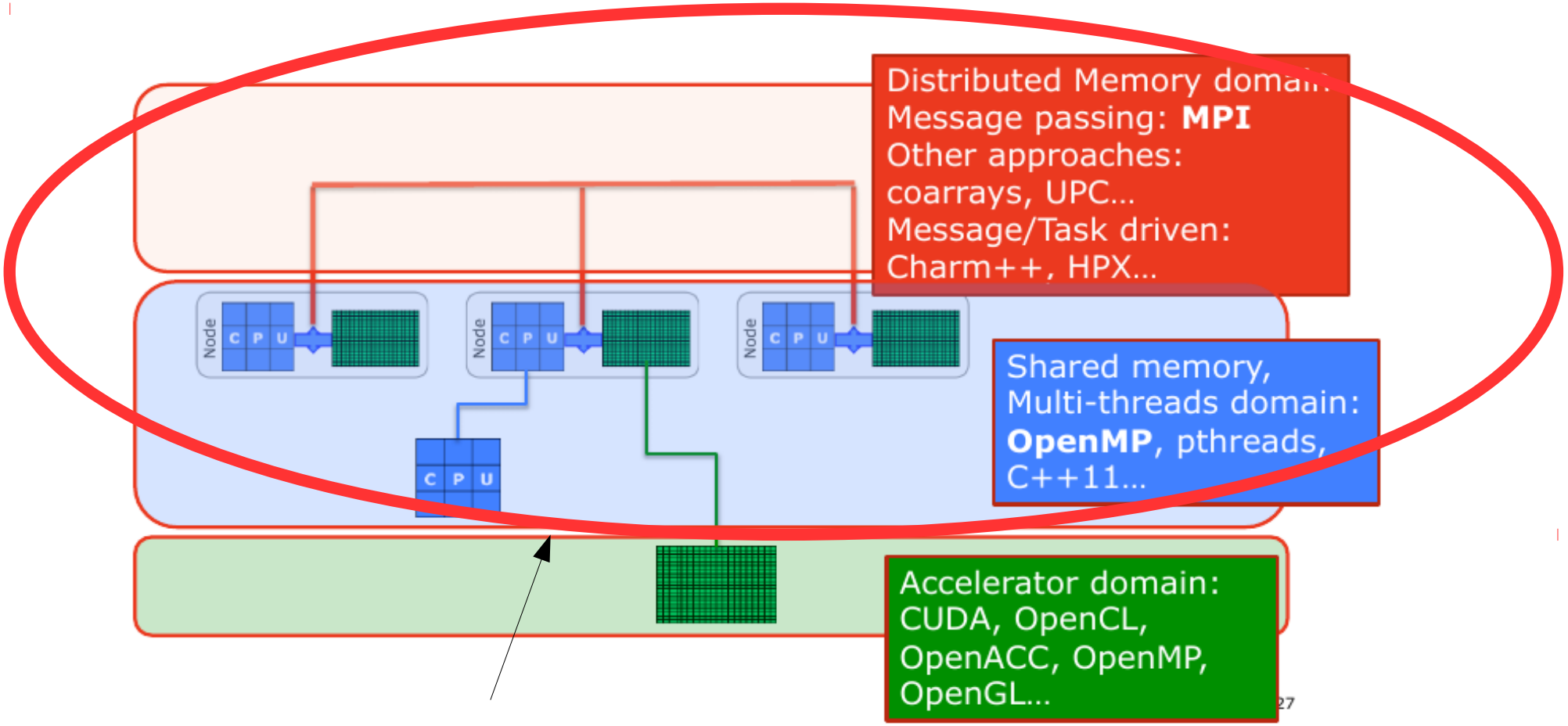
***Homework** – Introduction to Fortran or CPP, and compilation of code.

Day 2, Monday – Jan 30th (9:30-11.00)

1. Notes on basic code optimization (hands on).
2. Introduction to OpenMP (hands on).
3. Introduction MPI (hands on).
4. MPI & Python (hands on).

****Exercise sheet related to the day's topics** (Jan 31th, 15.45 – 17.15).

Wrap-up 1st lecture



Today we are concerned with "classical" hardware

(Slide from C. Gheller)

Before we start...

On ALPHACRUNCHER – set the environment

```
>cd ~
>vi .bashrc
```

→ add the following lines to your .bashrc

```
module load gcc
module rm gcc/6.1.0
module add openmpi/gcc/64/1.10.1
module load python/2.7.11
```

1. Basic optimization of serial code

In the age of multi-1000-processor parallel computers, writing code that runs efficiently on a single CPU has grown slightly old-fashioned.

Nevertheless there can be no doubt that single-processor optimizations are of premier importance.

→ If a **speed-up of two** can be achieved by some **simple code changes**, the user will be satisfied with much fewer CPUs in the parallel case.

→ This frees resources for other users and projects, and puts hardware that was often acquired for considerable amounts of money to better use.

→ If an existing parallel code is to be optimized for speed, it must be the first goal to **make the single processor run as fast as possible**.

Profiling

- Gathering information about a program's behaviour, specifically its use of resources, is called **profiling**.

- The most important "resource" in terms of high performance computing is **runtime**.

→ Hence, a common profiling strategy is to find out how much time is spent in the different functions, and maybe even lines, of a code in order to **identify hot spots**, i.e., the parts of the program that require the dominant fraction of runtime.

- These hot spots are subsequently analysed for possible optimization opportunities.

→ Software for profiling (e.g. GNU gprof,...)

Common sense optimizations

- Very simple code changes can often lead to a significant performance boost.
- Some of those hints may **seem trivial**, but experience shows that many scientific codes can be improved by the simplest of measures.

→ **e.g. do less work**

```
logical :: FLAG
FLAG = .false.
do i=1,N
  if(complex_func(A(i)) < THRESHOLD) then
    FLAG = .true.
  endif
enddo
```

If complex_func() has no side effects, the only information that gets communicated to the outside of the loop is the value of **FLAG**. In this case, depending on the probability for the conditional to be true, much computational effort can be saved by leaving the loop as soon as FLAG changes state.

EXIT THE LOOP

```
logical :: FLAG
FLAG = .false.
do i=1,N
  if(complex_func(A(i)) < THRESHOLD) then
    FLAG = .true.
    exit
  endif
enddo
```

Common sense optimizations II

Avoid branching!

In this multiplication of a matrix with a vector, the upper and lower triangular parts get different signs and the diagonal is ignored. The **if** statement serves to decide about which factor to use.

```
do j=1,N
  do i=1,N
    if(i.ge.j) then
      sign=1.d0
    else if(i.lt.j) then
      sign=-1.d0
    else
      sign=0.d0
    endif
    C(j) = C(j) + sign * A(i,j) * B(i)
  enddo
enddo
```

→ Fortunately, the loop nest can be transformed so that all if statements vanish:

```
do j=1,N
  do i=j+1,N
    C(j) = C(j) + A(i,j) * B(i)
  enddo
enddo
do j=1,N
  do i=1,j-1
    C(j) = C(j) - A(i,j) * B(i)
  enddo
enddo
```

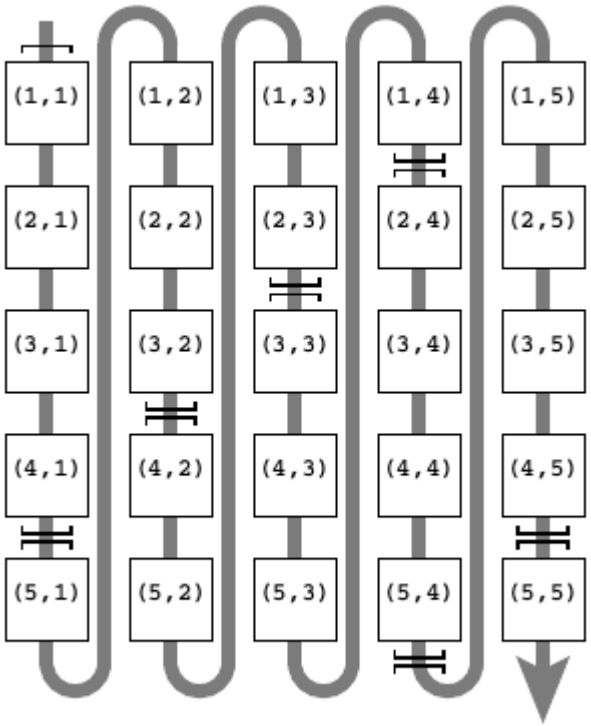
Data access (example)

Stride-N access

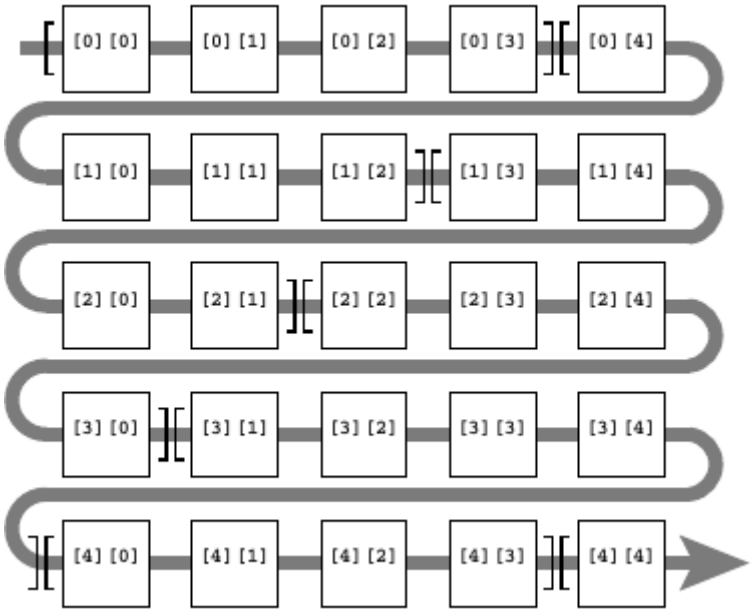
```
do i=1,N
  do j=1,N
    A(i,j) = i*j
  enddo
enddo
```

Stride-1 access

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



Fortran: Column major



C: Row major

Small exercise on loop ordering

check directory:

> cd zice17/scheidegger/intro_to_hpc/loop/
> vi loop.f90

compile:

>./compile_loop.sh

run:

>./test_loop

To be done:

a) Inspect code

b) run code (play with array size & re-compile)

Using SIMD instruction sets → Vectorization

Vectorization performs multiple operations in parallel on a core with a single instruction (SIMD – single instruction multiple data).

Data is loaded into vector registers that are described by their width in bits:

-256 bit registers: 8 x float, or 4x double
-512 bit registers: 16x float, or 8x double

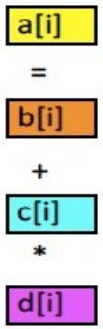
Vector units perform arithmetic operations on vector registers simultaneously.

Vectorization is key to maximizing computational performance.

Vectorization illustrated

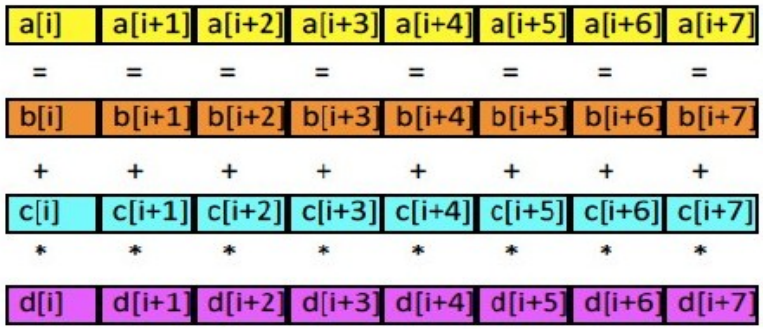
sequential

```
a [i] = b[i] + c[i] * d[i];
```



8× vectorization

```
a [i:8] = b[i:8] + c[i:8] * d[i:8];
```



In an optimal situation all this is carried out by the compiler automatically. Compiler directives can be used to give hints as to where vectorization is safe and/or beneficial.

```
! vectorized part
rest = mod(N,4)
do i=1,N-rest,4
  load R1 = [x(i),x(i+1),x(i+2),x(i+3)]
  load R2 = [y(i),y(i+1),y(i+2),y(i+3)]
  ! "packed" addition (4 SP flops)
  R3 = ADD(R1,R2)
  store [r(i),r(i+1),r(i+2),r(i+3)] = R3
enddo
! remainder loop
do i=N-rest+1,N
  r(i) = x(i) + y(i)
enddo
```

Advanced Vector Extensions (AVX)

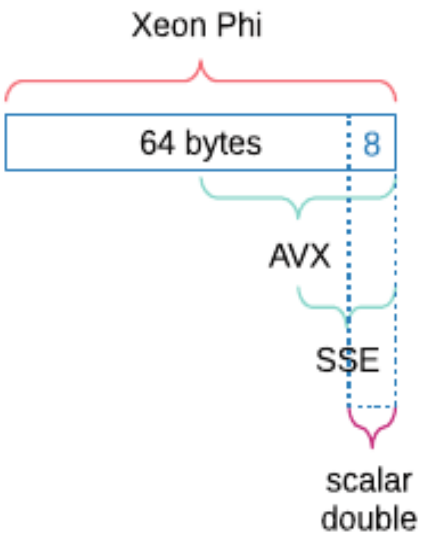


Fig. 7. Vector registers on modern CPUs: a scalar program can utilize only 1/4 of computational parallelism on AVX-enabled CPUs, e.g. the SandyBridge.

How to use vectorization

- use vector intrinsics (**see example**)

explicit hardware-specific instructions.
high performance.
non-portable and hard to maintain.
→ see, e.g. https://software.intel.com/sites/landingpage/IntrinsicsGuide/

-automatic compiler vectorization

compiler will vectorize where it is possible.
compilers can do a poor job.

-use libraries that are already vectorized

let somebody else do the work for you.

Does my code vectorize?

→ Not clear a priori.

→ Compilers can generate reports that summarize which loops vectorized.

→ You can ask for different levels of detail e.g. only loops that failed to vectorize, e.g., whether to explain why a loop didn't vectorize.

→ The flags vary from compiler to compiler, e.g.:

- **Intel** : `-vec-report=n`, or `-opt-report=n`
- **GCC**: `-ftree-vectorizer-verbose=n`
- **Cray**: `-h list=a`

You can also use the `disassemble` command in **gdb***, if you like reading assembly.

Small example on vectorization

check directory:

```
> cd /zice17/scheidegger/intro_to_hpc/vectorization/
> vi avx-intrinsics.cpp
```

compile:

```
>./compile_avx.sh
```

run:

```
>./avx-example
```

To be done:

a) Inspect code

Compilers

- Most codes benefit, to varying degrees, from employing **compiler-based optimizations**, e.g. standard optimization options (**-O0, -O1,. . .**).

- Every modern compiler has command line switches that allow a (more or less) fine-grained tuning of the available optimization options.

- Sometimes it is even worthwhile **trying a different compiler** just to check whether there is more performance potential. One should be aware that the compiler has the extremely complex job of mapping source code written in a high-level language to machine code, thereby utilizing the processor's internal resources as well as possible.

- However, there is no guarantee that this is actually the case and the programmer should at least be aware of the basic strategies for automatic optimization and potential stumbling blocks that prevent the latter from being applied. **It must be understood that compilers can be surprisingly smart and stupid at the same time**.

- A common statement in discussions about compiler capabilities is **"The compiler should be able to figure that out." This is often a false assumption.**

2. Introduction to OpenMP

(**Open M**ulti **P**rocessing)

- **OpenMP** website is a good source of information:

→ openmp.org

→ You can find there:

- tutorials and examples for all levels.
- the standard.
- quick references guide.

Some literature & other resources

Full standard/API specification:
- http://openmp.org

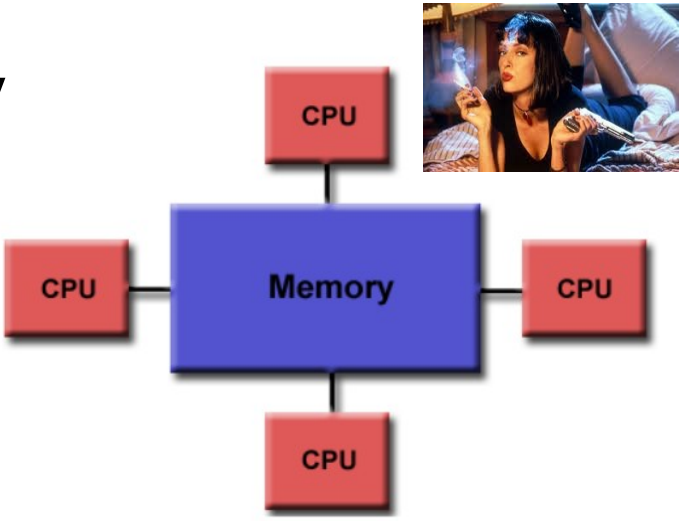
Tutorials:
- https://computing.llnl.gov/tutorials/openMP/

Books:
- “Introduction to High Performance Computing for Scientists and Engineers” Georg Hager, Gerhard Wellein

Shared memory systems

- **Process can access same GLOBAL memory**

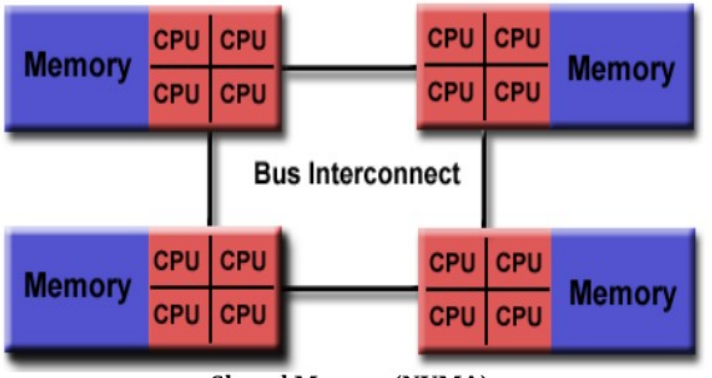
- **Uniform Memory Access (UMA) model**
  - Access time to memory is uniform.
  - Local cache, all other peripherals are shared.



Shared Memory (UMA)

- **Non-Uniform Memory Access (NUMA) model**
  - Memory is physically distributed among processors.
  - Global virtual address spaces accessible from all processors.
  - Access time to local and remote data is different.

→ **OpenMP**, but other solutions available (e.g. Intel's TBB).



Shared Memory (NUMA)

N-

What is OpenMP?

Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors (e.g. GNU, Intel, Cray, PGI,...).

OpenMP provides a portable, scalable model for developers of shared memory parallel applications.

Supports C/C++ and Fortran on a wide variety of architectures.

→ API may be used to explicitly direct multi-threaded, shared memory parallelism.

The API comprised of three main components:

1) Compiler Directives
2) Runtime Library Routines
3) Environment variables

Goals of OpenMP

- Standardization

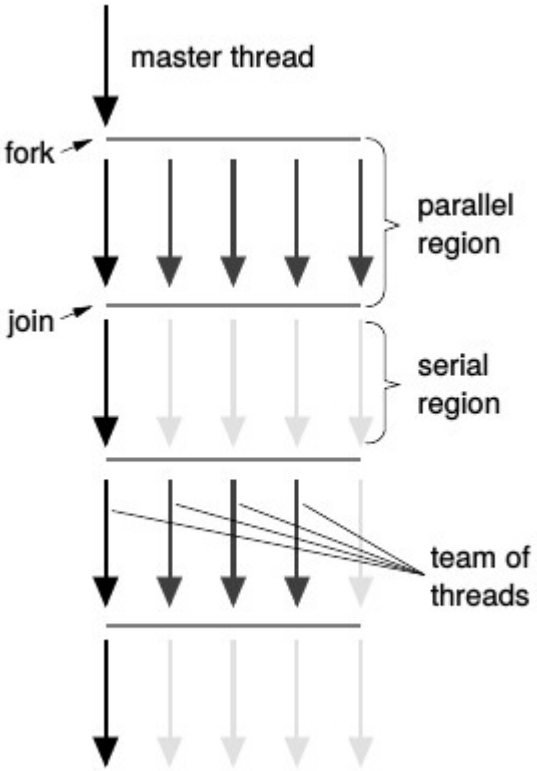
- Ease of use:
  - → Concise and simple set of directives.
  - → **Possible to get good speed-up with a handful of directives.**
  - → You can incrementally add it to the code without major changes.

- Should I use OpenMP?
  - → **Path least resistance to parallelize your code...**
  - → For many-core architectures like Xeon Phi,
    lightweight threading is required since MPI does not scale there...

Fork and Join Model

- OpenMP uses **fork** and **join** model for threading.

- The application starts with a master thread:
  - **FORK**: a team of parallel worker threads is started at the beginning of each parallel block.
  - The block is executed in parallel by each thread.
  - **JOIN**: the worker threads are synchronized at the end of the parallel block and join with the master thread.



- Threads are numbered **0:*N*-1** (*N* is the total number of threads).

- The **master** thread is always numbered 0.

OpenMP compiler directives

Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise usually by **specifying the appropriate compiler flag**.

OpenMP compiler directives are used for various purposes:

- Spawning a parallel region.
- Dividing blocks of code among threads.
- Distributing loop iterations between threads.
- Serializing sections of code.
- Synchronization of work among threads.

Compiler directives have the following syntax:

sentinel	directive-name	[clause ...]

All Fortran OpenMP directives must begin with a sentinel. The accepted sentinels depend upon the type of Fortran source. Possible sentinels are:
`!$OMP`
`C$OMP`
`*$OMP`
	A valid OpenMP directive. Must appear after the sentinel and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.

→ `!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)`

Compiling OpenMP

- Most compilers require a **flag** to enable OpenMP compilation
  → without any flag, the **#pragma** or **!$** directives are ignored by the compiler and a serial application is created.

- Compilers that don't understand OpenMP will simply ignore the directives (no portability problems).

```
Cray    : on by default for -O1 and greater, disable with -h noomp
Intel   : off by default, enable with -openmp
GNU     : off by default, enable with -fopenmp
PGI     : off by default, enable with -mp
```

Runtime library

- OpenMP API includes a growing number of **runtime library routines**.

These are used for a variety of purposes:

- Setting and querying the number of threads.
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size.
- Setting and querying the dynamic threads feature.
- Querying if in a parallel region, and at what level.
- Setting and querying nested parallelism.
- Setting, initializing and terminating locks and nested locks.
- Querying wall clock time and resolution.

- For C/C++, all of the runtime library routines are actual subroutines.
- For Fortran, some are actually functions, and some are subroutines.

Runtime library (II)

- OpenMP has runtime library routines for controlling your application, including

Function:	omp_get_num_threads()
C/ C++	int omp_get_num_threads(void);
Fortran	integer function omp_get_num_threads()
Description: Returns the total number of threads currently in the group executing the parallel block from where it is called.	

Function:	omp_get_thread_num()
C/ C++	int omp_get_thread_num(void);
Fortran	integer function omp_get_thread_num()
Description: For the master thread, this function returns zero. For the child nodes the call returns an integer between 1 and omp_get_num_threads()-1 inclusive.	

- There are many others, however these are probably the most commonly used.

Runtime library III

The runtime library requires that the OpenMP **header/module** is included:

```c
#include <omp.h>
...
int threads = omp_get_max_threads();
int outside = omp_get_num_threads();
int inside;
#pragma omp parallel
{
  inside = omp_get_num_threads();
}
printf("%d in, %d out, %d max \n",
       inside, outside, threads);
```

```fortran
use omp_lib
...
integer :: threads, inside, outside
threads = omp_get_max_threads()
outside = omp_get_num_threads()
!$omp parallel
inside = omp_get_num_threads()
!$omp end parallel
print *, inside, ' in ', outside, ' out ',
         threads, ' max'
```

```
> OMP_NUM_THREADS=8 ./a.out
8 in, 1 out, 8 max
```

Running OpenMP applications

- The default number of threads is set with an environment variable **OMP_NUM_THREADS**

csh/tcsh	`setenv OMP_NUM_THREADS 8`
sh/bash	`export OMP_NUM_THREADS=8`

- Compiling and running:

```
OpenMP$ gfortran hello_world.f90 -fopenmp -o app.exe
OpenMP$ export OMP_NUM_THREADS=4
OpenMP$ ./app.exe
```

compile openmp code

set 4 threads

run

“Hello world” in OpenMP (FORTRAN)

```
      program hello_world

      use omp_lib       !module with API declarations

      integer :: tid    !thread ID

!-----------------------------------------------------------------------
      write(*,*) '=== serial section ==='

      write(*,*) 'hello world from thread ', omp_get_thread_num(), ' of ', omp_get_num_threads()

!-----------------------------------------------------------------------
      write(*,*) '=== parallel section ==='
!$omp parallel private(tid)

!.....get thead ID
      tid = omp_get_thread_num();

!.....write a personalized message from this thread
      write(*,*) 'hello world from thread ', omp_get_thread_num(), ' of ', omp_get_num_threads()

!.....section only executed by master
      if(tid .eq. 0 ) then
        write(*,*) 'hello world from master thread with TID = ', tid
      end if

!....All threads join master thread and disband
!$omp end parallel

!-----------------------------------------------------------------------
      end program hello_world
!-----------------------------------------------------------------------
```

“Hello world” in OpenMP (CPP)

```
#include <omp.h>

main ()
{
   int nthreads, tid;
   #pragma omp parallel private(nthreads, tid)
   {
      tid = omp_get_thread_num();
      printf("Hello World from thread = %d\n", tid);
      if (tid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }
   }
}
```

Non shared copies of data for each thread

OpenMP directive to indicate START segment to be parallelized

Code segment that will be executed in parallel

OpenMP directive to indicate END segment to be parallelized

Data scoping

- Any variables that existed before a parallel region still exist inside, and are **by default shared between all threads**.

- True work sharing, however, makes sense only if **each thread can have its own, private variables**.

- OpenMP supports this concept by defining a separate stack for every thread.

1. A variable that exists before entry to a parallel construct can be privatized, i.e., made available as a **private instance for every thread**, by a **PRIVATE** clause to the OMP PARALLEL directive. The private variable's scope extends until the end of the parallel construct.

2. The index variable of a work-sharing loop is automatically made private.

3. Local variables in a subroutine called from a parallel region are private to each calling thread. This pertains also to copies of actual arguments generated by the call-by-value semantics, and to variables declared inside structured blocks in C/C++. However, local variables carrying the SAVE attribute in Fortran (or the static storage class in C/C++) will be shared. **Shared variables that are not modified in the parallel region do not have to be made private**.

Scope of Variables

OpenMP provides clauses that describe how variables should be shared between threads

– **shared**: all variables access the same copy of a variable.

→ this is the default behavior

→ WARNING: take care when writing to shared variables.

– **private**: each thread gets its own copy of the variable

→ private copy is uninitialized.

→ use *firstprivate* to initialize variable with value from master.

Example 1:”hello world from thread”

1. go to /zice17/scheidegger/intro_to_hpc/openmp:

> cd /zice17/scheidegger/intro_to_hpc/openmp

2. Have a look at the code

> vi 1.hello_world.f90 / vi 1a.hello_world.cpp

3. compile by typing:

> make

4. Experiment with different numbers of threads

> export OMP_NUM_THREADS=1 (play around with #threads)
> ./hello_world.exec (FORTRAN)
> ./1a.hello_world.exec (CPP)

Example 1:”hello world from thread”

5. experiment with slurm (the settings)

> cd /zice17/scheidegger/intro_to_hpc/openmp
> vi submit_openmp.sh

```
#!/bin/bash -l

#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8

#SBATCH --time=00:01:00


#SBATCH --job-name=test_submission
#SBATCH --output=openmp_test.out
#SBATCH --error=openmp_test.err

export OMP_NUM_THREADS=8


### openmp executable
./1.hello_world.exec
```

6. see **1.hello_world.f90** line 25: !$omp parallel private(tid)
   → what happens if we remove **private(tid)** → try out!

Shared memory model

- OpenMP uses a shared memory model.

- All threads can read and write to the same memory locations simultaneously.

- By default variables are shared, so one copy is used by all threads.

- The result of computations where multiple threads attempt to read/write to a variable are undefined.

→ **see /zice17/scheidegger/intro_to_hpc/openmp/2.example_racing_cond.f90**

→ this is a very common parallel programming bug called **race condition**.

Example 2: Race condition

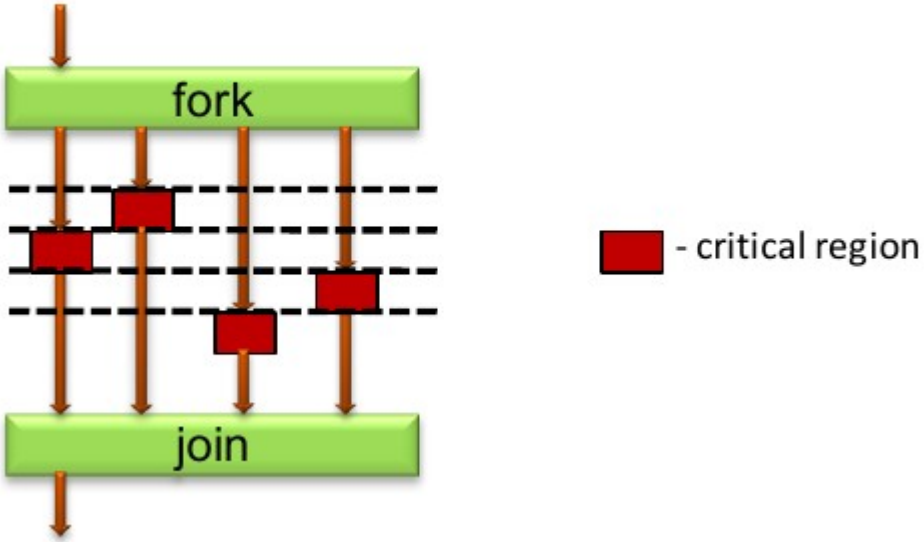
```
!=====================================================================
!
!     OpenMP example: Racing conditions example
!
!=====================================================================

      program racing_cond

      use omp_lib

      implicit none

      integer :: num_threads, expected
      integer :: sum = 0

!---------------------------------------------------------------------
      num_threads = omp_get_max_threads()
      write(*,*) 'sum with ', num_threads, ' threads'

!---------------------------------------------------------------------
!.....section where we can have racing conditions
!$omp parallel
      sum = sum + omp_get_thread_num() + 1
!$omp end parallel

!---------------------------------------------------------------------
!.....use formula for sum of arithmetic sequence: sum(1:n) = (n+1)*n/2
      expected = (num_threads + 1)*num_threads/2

      if (sum .eq. expected) then
        write(*,*) "sum ", sum, ' matches the expected value'
      else
        write(*,*) "sum ", sum, ' does not match the expected value ', expected
      endif

!---------------------------------------------------------------------
      end program racing_cond
!---------------------------------------------------------------------
```

Synchronization/critical regions

- Concurrent write access to a shared variable or, in more general terms, a shared resource, must be avoided by all means to circumvent race conditions.

- Critical regions solve this problem by making sure that at most one thread at a time executes some piece of code.

- If a thread is executing code inside a critical region, and another thread wants to enter, the latter must wait (block) until the former has left the region.



Example 3: Race conditions fixed

```fortran
!=====================================================================
!
!     OpenMP example: Racing conditions example
!
!=====================================================================

      program racing_cond_fix

      use omp_lib

      implicit none

      integer :: num_threads, expected
      integer :: sum = 0

!---------------------------------------------------------------------
      num_threads = omp_get_max_threads()
      write(*,*) 'sum with ', num_threads, ' threads'

!---------------------------------------------------------------------
!.....section where we can have racing conditions
!$omp parallel

!.....all theads will execute block, one at at time
!$omp critical
      sum = sum + omp_get_thread_num() + 1
!$omp end critical

!$omp end parallel

!---------------------------------------------------------------------
!.....use formula for sum of arithmetic sequence: sum(1:n) = (n+1)*n/2
      expected = (num_threads + 1)*num_threads/2

      if (sum .eq. expected) then
        write(*,*) "sum ", sum, ' matches the expected value'
      else
        write(*,*) "sum ", sum, ' does not match the expected value ', expected
      endif

!---------------------------------------------------------------------
      end program racing_cond_fix
!---------------------------------------------------------------------
```

Example 2 fixed: Racing conditions

1. vi /zice17/scheidegger/intro_to_hpc/openmp/3.racing_cond_fix.f90

2. Have a look at the code

3. compile by typing:

> make

4. Experiment with different numbers of threads

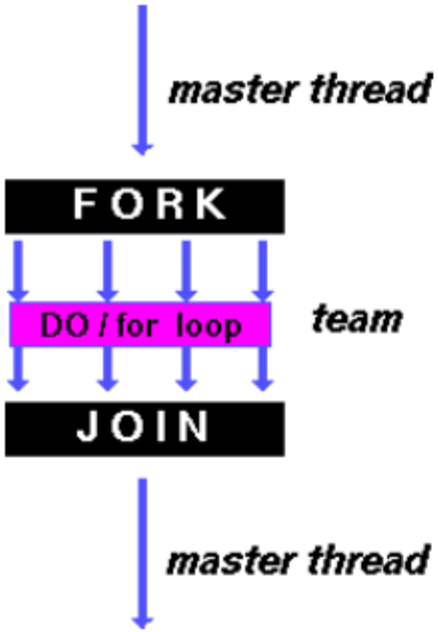
the **CRITICAL** and **END CRITICAL** directives bracket the update to ***sum*** so that the result is always correct.

→ **WARNING: SYNCHRONIZATION (AND SERIAL CODE REGIONS) CAN QUICKLY LIMIT POTENTIAL SPEED-UP FROM PARALLELISM**

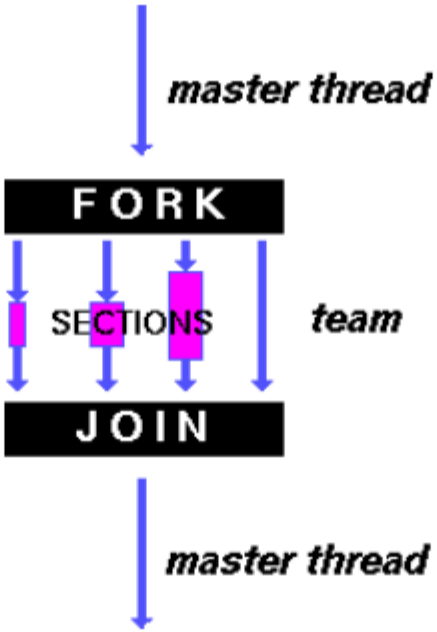
Worksharing constructs in OpenMP

See https://computing.llnl.gov/tutorials/openMP/

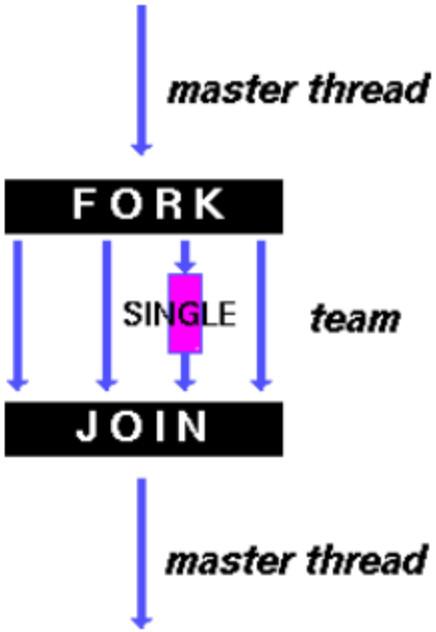
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



SINGLE - serializes a section of code



Example: “do loops”

Serial code

```
double *x, *y, *z;
int n;
for(int i=0; i<n; ++i) {
  z[i] = x[i] + y[i];
}
```

C++

```
real(kind=8) :: x(:), y(:), z(:)
integer      :: i, n
do i=1,n
  z(i) = x(i) + y(i)
end do
```

Fortran

Parallel code

→ **compiler handles loop bounds for you.**

→ **there is a compact single-line directive.**

→ **!$OMP DO (in Fortran)**

```
double *x, *y, *z;
int n, i;
#pragma omp parallel
{
  #pragma omp for
  for(i=0; i<n; ++i) {
    z[i] = x[i] + y[i];
  }
}
```

C++

loop index variable i is private by default

```
real(kind=8) :: x(:), y(:), z(:)
integer      :: i, n
!$omp parallel
!$omp do
do i=1,n
  z(i) = x(i) + y(i)
end do
!$omp end do
!$omp end parallel
```

Fortran

→ let's attempt to parallelize the integral

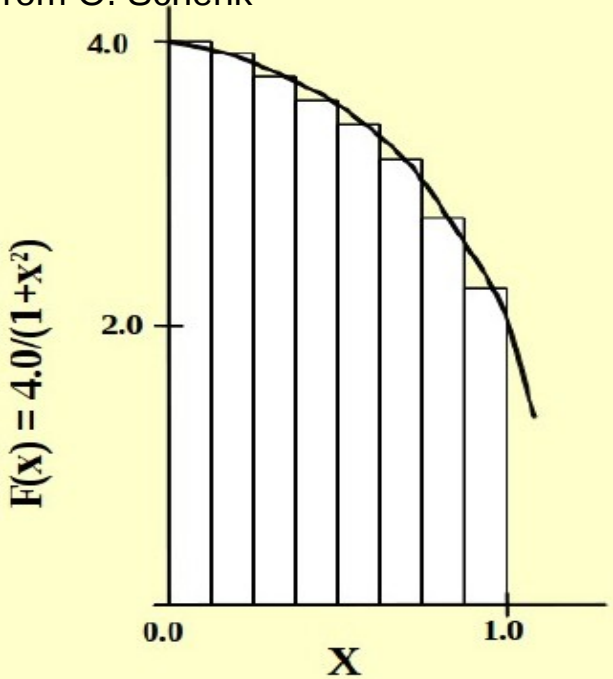
$$\pi = \int_0^1 dx \frac{4}{1+x^2}$$

by using techniques learnt so far (“**summation the hard way**”).

→ **zice17/scheidegger/intro_to_hpc/openmp/4.integration_pi.f90**

Exercise 4: Computing Pi

Fig. from O. Schenk



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

```
      program integration_pi

      use omp_lib

      implicit none

      real(8) :: pi,w,sum,x, time
      integer(8) :: i, N = 500000000

!-----------------------------------------------------------------------
      pi = 0.d0
      w = 1.d0/N
      sum = 0.d0

!.....start timer
      time = -omp_get_wtime()
!-----------------------------------------------------------------------
!$OMP PARALLEL PRIVATE(x) FIRSTPRIVATE(sum)
!$OMP DO
      do i = 1 , n
        x = w*(i-0.5d0)
        sum = sum + 4.d0/(1.d0+x*x)
      enddo
!$OMP END DO

!$OMP CRITICAL
      pi= pi + w*sum
!$OMP END CRITICAL
!$OMP END PARALLEL

!.....end timer
      time = time + omp_get_wtime()

!-----------------------------------------------------------------------
      write(*,*) 'The approximation of Pi =', pi
      write(*,*) 'took ', time, ' seconds'

!-----------------------------------------------------------------------
      end program integration_pi
!-----------------------------------------------------------------------
```

Reductions
→ e.g. summation the "easy way"

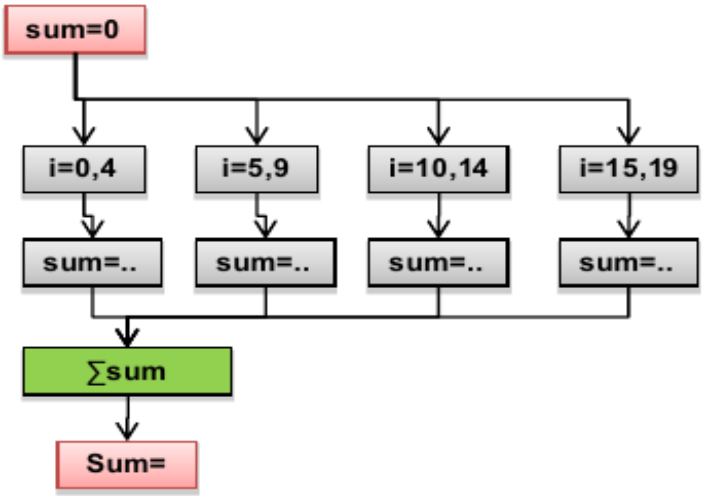
The **REDUCTION** clause performs a reduction on the variables that appear in the list.

→ reduction(op:list)

e.g. !$OMP DO REDUCTION(+:pi),
!$OMP DO REDUCTION(max:Maxval)

A private copy for each list variable is created for each thread.

At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.



Example: Reduction

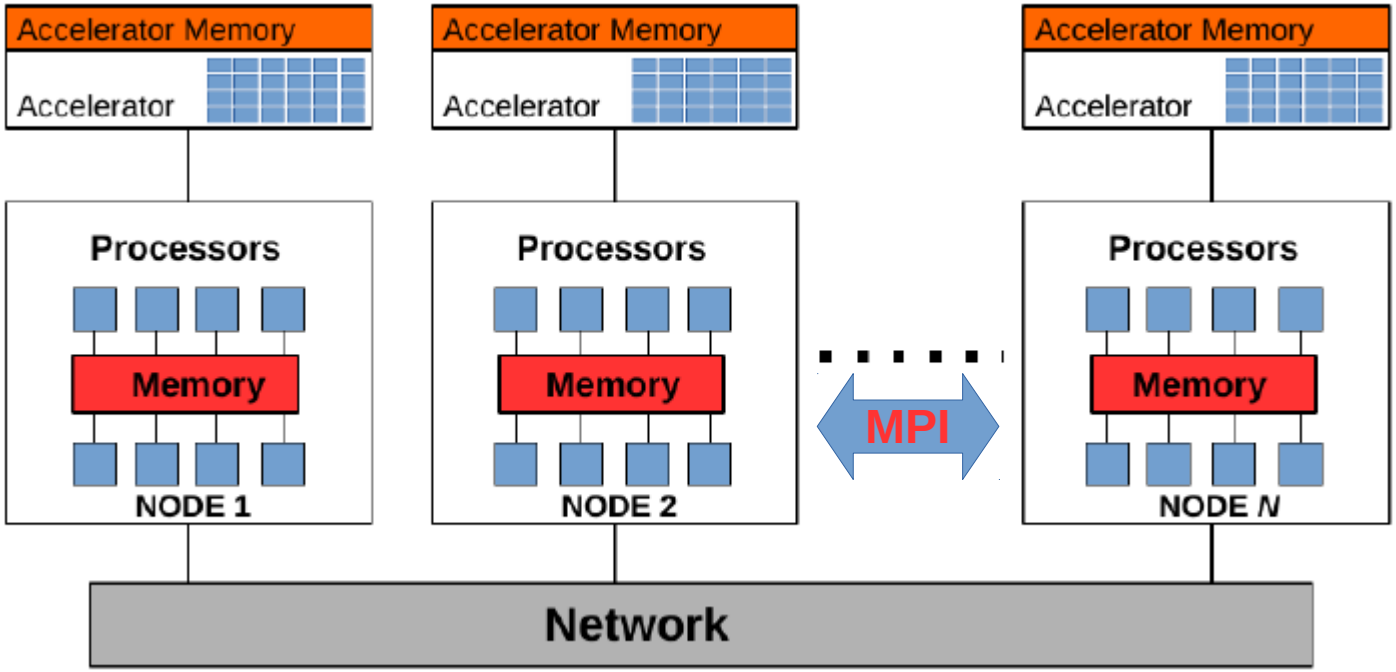
>/zice17/scheidegger/intro_to_hpc/openmp/4a.integration_pi_reduction.cpp
>/zice17/scheidegger/intro_to_hpc/openmp/4b.integration_pi_reduction.f90
(play with OMP_NUM_THREADS & timing)

```
!=====================================================================
!
!     OpenMP example: program to approximation pi by a REDUCTION
!
!=====================================================================
      program integration_pi_reduction

      use omp_lib

      implicit none

      real(8) :: pi,w,x, time
      integer(8) :: i, N = 500000000

!---------------------------------------------------------------------
      pi = 0.d0
      w = 1.d0/N

!.....start timer
      time = -omp_get_wtime()

!---------------------------------------------------------------------
!$OMP PARALLEL PRIVATE(x)
!$OMP DO REDUCTION(+:pi)
      do i = 1 , n
        x = w*(i-0.5d0)
        pi = pi + 4.d0/(1.d0+x*x)
      enddo
!$OMP END DO
!$OMP END PARALLEL

!.....end timer
      time = time + omp_get_wtime()

!---------------------------------------------------------------------
      write(*,*) 'The approximation of Pi =', pi*w
      write(*,*) 'took ', time, ' seconds'

!---------------------------------------------------------------------
      end program integration_pi_reduction
!---------------------------------------------------------------------
```

Reduction

Break a nice mountain

What is MPI?



- Ever since parallel computers hit the HPC market, there was an intense discussion about what should be an appropriate programming model for them.

- Message passing is required if a parallel computer is of the distributed memory type, i.e. if there is no way for one processor to directly access the address space of another.

- The use of explicit **message passing** (**MP**), i.e., communication between processes, is surely the most tedious and complicated but also the most flexible parallelization method.

Message Passing Interface (MPI)

- Resources are LOCAL (different from shared memory).

- Each process runs in an “isolated” environment. Interactions requires **Messages** to be exchanged

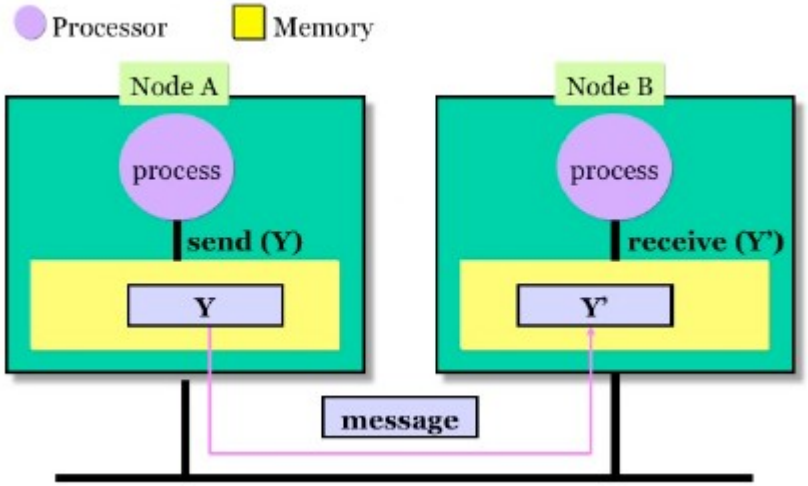
- Messages can be: **instructions, data, synchronization.**

- **MPI works also on Shared Memory systems**.

- Time to exchange messages is much larger than accessing local memory.

→ **Massage Passing is a COOPERATIVE Approach, based on 3 operations:**

- **SEND** (a message)
- **RECEIVE** (a message)
- **SYNCHRONIZE**



MPI availability

- MPI is standard defined in a set of documents compiled by a consortium of organizations : http://www.mpi-forum.org/

- In particular the MPI documents define the APIs (application programming interfaces) for C, C++, FORTRAN77 and FORTRAN 90.

- Bindings available for Perl, Python, Java...

- In all systems MPI is implemented as a library of subroutines/functions over the network drivers and primitives.

Messages

- A message can be as simple as a **single item** (like a number) or even a **complicated structure**, perhaps scattered all over the address space.

- For a message to be transmitted in an orderly manner, some parameters have to be fixed in advance, such as:

• Which process is sending the message?

• Where is the data on the sending process?

• What kind of data is being sent?

• How much data is there?

• Which process is going to receive the message?

• Where should the data be left on the receiving process?

• What amount of data is the receiving process prepared to accept?

→ **All MPI calls that actually transfer data have to specify those parameters in some way.**

MPI: Pro's & Con's

Pro's

- **Distribute Memory → use more nodes and cores.**
- Communications is the most important part of HPC.
  → It can be (and is) highly optimized.
- MPI is portable (runs on almost any platform).
- Many current applications/libraries use MPI.
- MPI de-facto standard for distributed memory processing.

Con's

- **Error-prone**.
- Discourages frequent communications (overhead).
- **Technically “hard” to implement**.

General program structure

Header file

→ required for all programs that make MPI calls.
→ contains definitions of MPI constants, types and functions.

Fortran 90
```
USE MPI
```

C/C++
```
#include <mpi.h>
```

MPI initialize/finalize

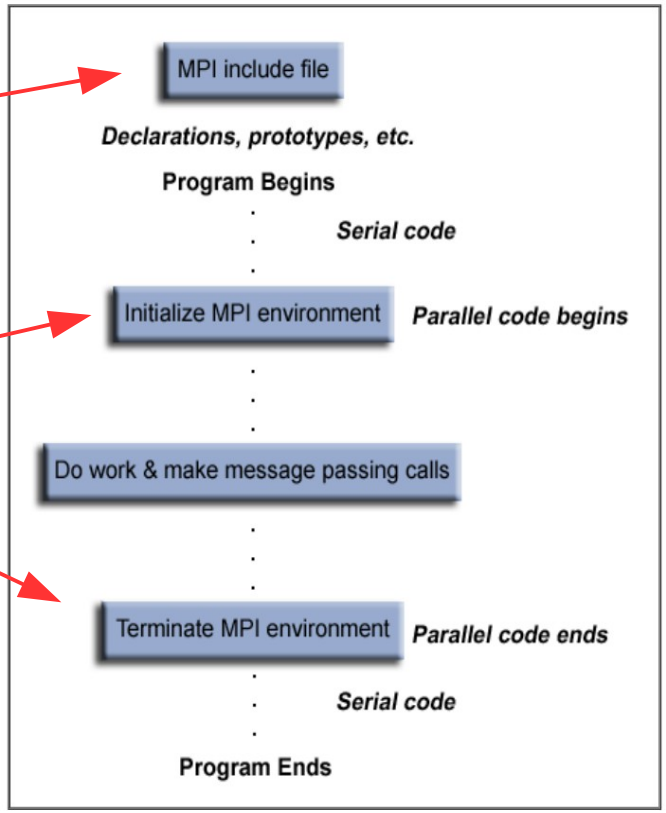
→ Every MPI program starts by calling **MPI_Init**
→ Every MPI program ends by calling **MPI_Finalize**

Fortran
```
INTEGER IERR
MPI_INIT(IERR)
```

Fortran
```
INTEGER IERR
MPI_FINALIZE(IERR)
```

C/C++
```
int MPI_Init(int*argc, char***argv)
```

C/C++
```
int MPI_Finalize()
```



Format of MPI calls

→ C names: case sensitive; Fortran not.
→ Programs must not declare variables or functions with the prefix MPI_ …

MPI communicators and ranks

Communicator

- MPI uses objects called **communicators** and groups to **define which collection of processes may communicate with each other**.
- Most MPI routines require you to specify a communicator as an argument.

- Communicators and groups will be covered in more detail later. For now, simply use **MPI_COMM_WORLD** whenever a communicator is required.

- **It is the predefined communicator that includes all of your MPI processes**.

Rank

- Within a communicator, every process has its own unique, **integer identifier** assigned by the system when the process initializes (**=Rank**)

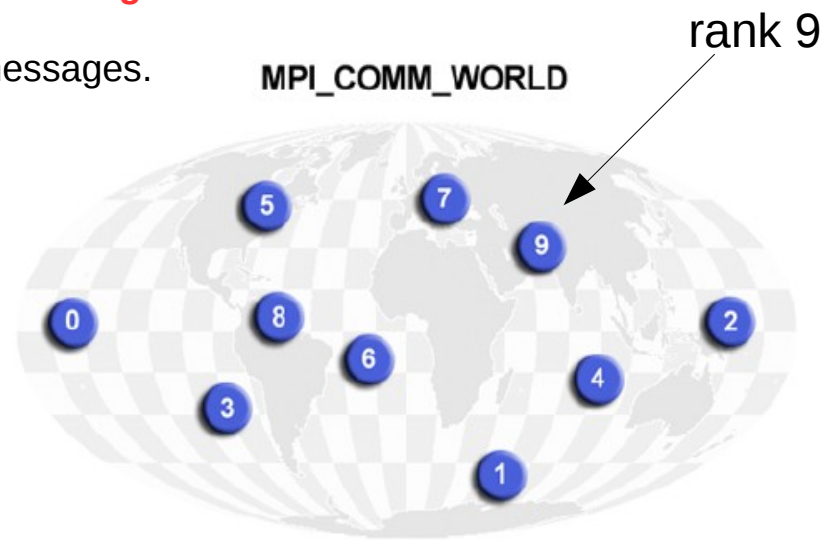
- A rank is sometimes also called a **"task ID"**. Ranks are contiguous and **begin at zero**.

- Used by the programmer to specify the source and destination of messages.

- Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).

Error Handling

Most MPI routines include a return/error code parameter.



Process Identification

- How many processes are associated with a communicator?

Fortran
```
INTEGER COMM, SIZE, IERR
CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
```

C/C++
```
MPI_Comm_size(MPI_Comm comm, int *size)
```

- How to get the rank of a process?

Fortran
```
INTEGER COMM, RANK, IERR
CALL MPI_COMM_RANK(COMM, RANK, IERR)
OUTPUT: RANK
```

C/C++
```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Compiling and running MPI

MPI is always available as a **library**. In order to **compile and link** an MPI program, compilers and linkers need options that specify where include files (i.e., C headers and Fortran modules) and libraries can be found.

As there is considerable variation in those locations among installations, most MPI implementations provide compiler wrapper scripts (often called **mpicc, mpif90, etc.**), which supply the required options automatically but otherwise behave like “normal” compilers.

Note that the way that MPI programs should be compiled and started is not fixed by the standard, so please consult the system documentation by all means.

Compile →
Run →

```
$ mpif90 -O3 1.hello_world_mpi.f90 -o 1.hello_world_mpi.exec
$ mpiexec -np 4 ./1.hello_world_mpi.exe
```

Launch 4 MPI processes

“Hello World” in Fortran

```fortran
      program hello_world_mpi

      use mpi

      implicit none

      integer :: ierror, rank, size

!-----------------------------------------------------------------------
      call MPI_INIT(ierror)   !start MPI

!-----------------------------------------------------------------------
!.....Get the rank of each process
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

!-----------------------------------------------------------------------
!.....Get the size of the communicator
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)

!-----------------------------------------------------------------------
!.....Write code such that every process writes its rank and the
!.....size of the communicator, but only process 0 prints "hello world"
      if (rank .eq. 0) then
        write(*,*) 'Hello world!'
      end if

!-----------------------------------------------------------------------
!.....every process does something
      write(*,*) 'I am process', rank, ' out of', size

!-----------------------------------------------------------------------
      call MPI_FINALIZE(ierror)  !finish MPI

!-----------------------------------------------------------------------
      end program hello_world_mpi
!-----------------------------------------------------------------------
```

This initializes the parallel environment

The **MPI_COMM_WORLD** handle describes all processes that have been started as part of the parallel program.
If required, other communicators can be defined as subsets of **MPI_COMM_WORLD**.

```
mpiexec -np 4 ./1a.hello_world_mpi_cpp.exec
```

```
Hello World, I am 2 of 4
Hello World, I am 0 of 4
Hello World, I am 3 of 4
Hello World, I am 1 of 4
```

Note that no MPI process except rank 0 is guaranteed to execute any code beyond MPI_Finalize().

Hello World in MPI (Fortran/CPP)

1. go to scheidegger/intro_to_hpc/MPI:

> cd scheidegger/intro_to_hpc/MPI

2. Have a look at the code

>vi 1.hello_world_mpi.f90 / 1a.hello_world_mpi.cpp

3. compile by typing:

> make

4. Experiment with different numbers of MPI processes

>mpiexec -np 4 ./1.hello_world_mpi.exec (-np #processes)

>mpiexec -np 4 ./1a.hello_world_mpi_cpp.exec (-np #processes)

Hello World in C++

The MPI bindings for the C language follow the **case-sensitive name pattern** MPI_Xxxxx..., while Fortran is case-insensitive!

```c
#include <stdio.h>

#include <mpi.h>

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

int rank, size;

MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &size);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

printf("Hello World, I am %d of %d\n", rank, size);

MPI_Finalize();

return 0;

}
```

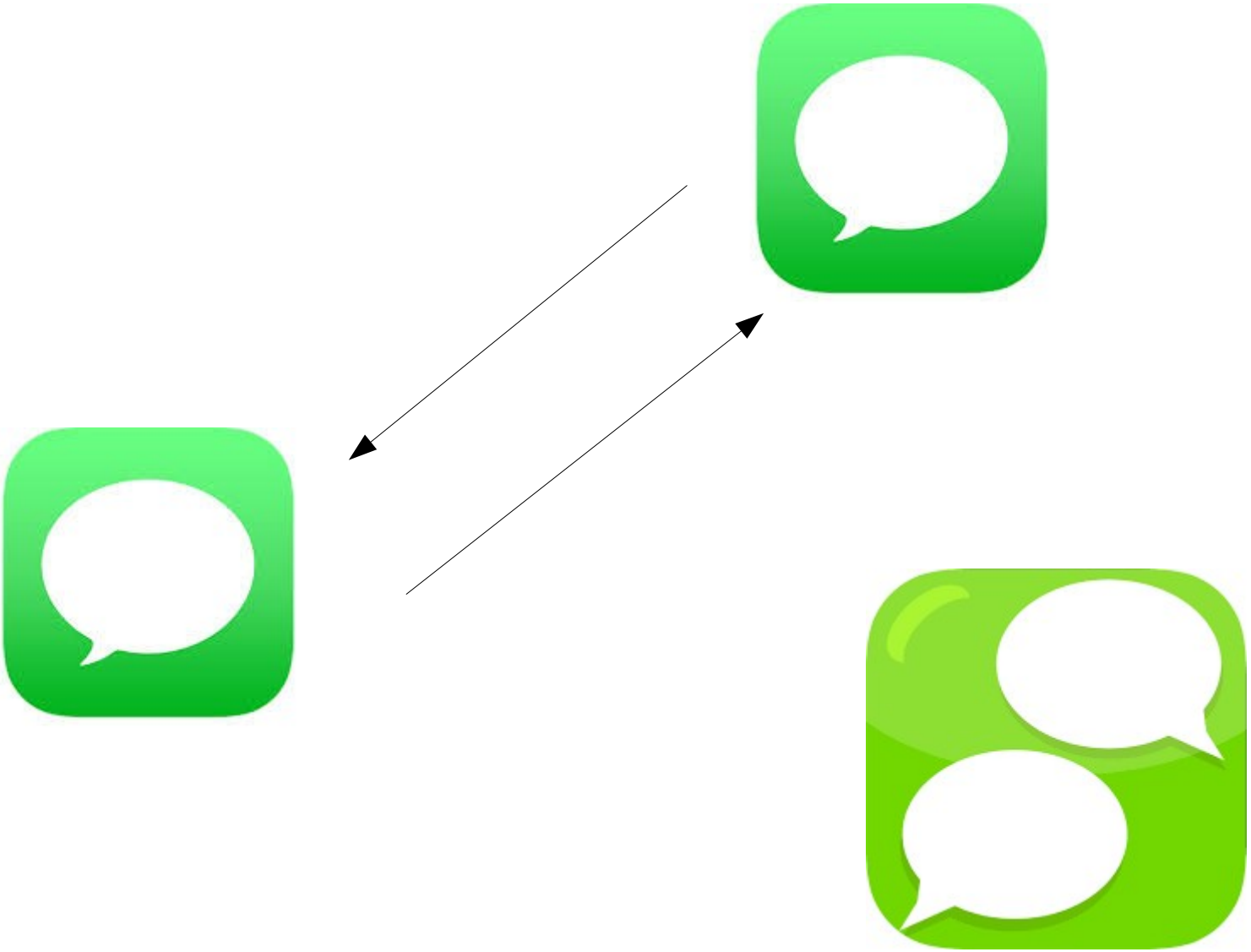
Note that the C bindings require output arguments (like rank and size above) to be specified as pointers.

```
mpiexec -np 4 ./1a.hello_world_mpi_cpp.exec
```

```
Hello World, I am 2 of 4
Hello World, I am 0 of 4
Hello World, I am 3 of 4
Hello World, I am 1 of 4
```

Below, I will mostly stick to the Fortran MPI bindings, and only describe the differences to C/C++ where appropriate.

Messages and point-to-point communication



Message Envelope

- Communication is performed by explicitly **sending** and **receiving** messages.

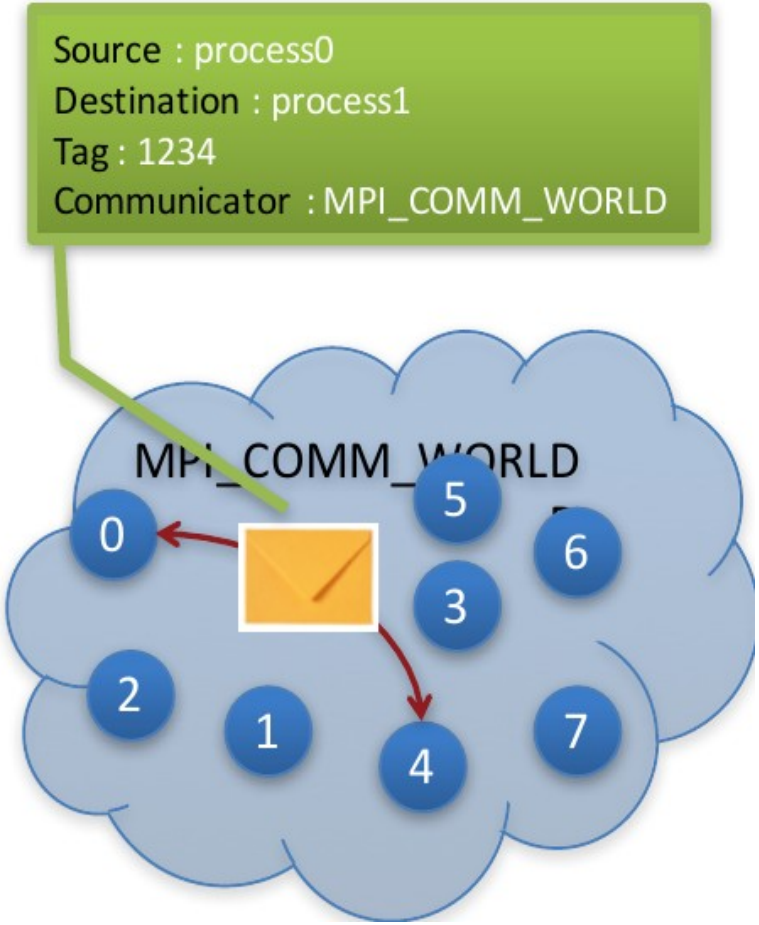
- Messages need **meta data** to describe sender and receiver as well as message contents.

- Sender and receiver are described by ranks within a group.

- Message contents given by starting address, data type, and count, where data type is:

  → Elementary (all C and Fortran data types)
  → Contiguous array of data types
  → Strided blocks of data types
  → Indexed array of blocks of data types
  → General structure

MPI also allows messages to be tagged so that programs can deal with the arrival of messages in an orderly way.



Point-to-Point communication

It is the fundamental communication facility provided by a MPI library.

Communication between 2 processes.

It is conceptually simple:

→ **source process A sends a message to destination process B.**
→ **B receives the message from A.**

Communication takes place within a communicator.

Source and Destination are identified by their rank in the Communicator.

Simple MPI communication model

Process with rank 1 sends to process with rank 2 (pseudo code)

```
Rank 1: MPI_Send(<send data buffer>, 2, MyCommunicator)
Rank 2: MPI_Recv(<recv data buffer>, 1, MyCommunicator)
```

- same communicator: MyCommunicator

- send and recv buffer should be compatible:
  - → **receive buffer should be large enough**.
  - → **data type should match**.

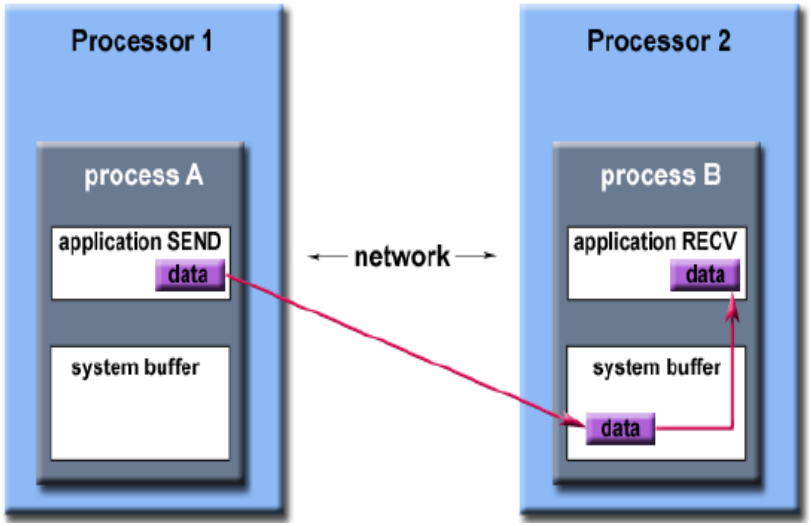
- Rank 2 is prepare to receive data from Rank 1
  - → MPI_Recv is called in ”the right order” (deadlock).
  - → Rank 2 knows the maximum bound on the buffer size.

Message Buffering

- In a perfect world, every send operation would be perfectly synchronized with its matching receive.

- This is rarely the case.

- The MPI implementation must be able to deal with storing the data **when the two tasks are out of sync.**



Consider the following two cases:

→ A send operation occurs 5 seconds before the receive is ready where is the message while the receive is pending?

→ Multiple sends arrive at the same receiving task which can only accept one send at a time what happens to the messages that are "backing up"?

- The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a system buffer area is reserved to hold data in transit.

Point-to-point communication

- “Hello World” example did not contain any real communication apart from starting and stopping processes.

- **Point-to-Point** communication is the fundamental communication facility provided by MPI.
→ Communication between two separate processes.

→ **Source process A** sends a message to **destination process B**.
→ B receives the message from A.
→ Communication takes place within a communicator.
→ Source and destination are identified by their rank in the communicator.

```
integer :: count, datatype, dest, tag, comm, ierror
call MPI_Send(buf,        ! message buffer
              count,      ! # of items
              datatype,   ! MPI data type
              dest,       ! destination rank
              tag,        ! message tag (additional label)
              comm,       ! communicator
              ierror)     ! return value
```

```
integer :: count, datatype, source, tag, comm,
integer :: status(MPI_STATUS_SIZE), ierror
call MPI_Recv(buf,        ! message buffer
              count,      ! maximum # of items
              datatype,   ! MPI data type
              source,     ! source rank
              tag,        ! message tag (additional label)
              comm,       ! communicator
              status,     ! status object (MPI_Status* in C)
              ierror)     ! return value
```

Messages – terminology

Data is exchanged in the **buffer, an array of count elements of some particular MPI data type**.

→ One argument that usually must be given to MPI routines is the **type of the data being passed**.

→ This allows MPI programs to run automatically in heterogeneous environments.

Messages are identified by their envelopes.

A message could be exchanged only if the sender and receiver specify the correct envelope.

body			envelope			
buffer	count	datatype	source	destination	communicator	tag

Example Ping - Pong

```
      integer, parameter :: process_a = 0
      integer, parameter :: process_b = 1

      integer, parameter :: ping = 17 !message tag
      integer, parameter :: pong = 23 !message tag

      integer, parameter :: length = 1
      integer :: status(MPI_STATUS_SIZE)
      real :: buffer(length)
      integer :: i

      integer :: ierror, my_rank, size

!-----------------------------------------------------------------------
      call MPI_INIT(ierror)

      call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)


!.....the example should only be run with 2 procs, else abort
      if (size .ne. 2) then
        write(*,*) 'please run this with 2 processors'
        call MPI_Finalize(ierror)
        stop
      end if

!-----------------------------------------------------------------------
!.....write a loop of number_of_messages iterations. Within the loop, process A sends a message
!.....(ping) to process B. After receiving the message, process B sends a message (pong) to process A)
      if (my_rank .eq. process_a) then
        call MPI_SEND(buffer, length, MPI_REAL, process_b, PING, MPI_COMM_WORLD, ierror)
        call MPI_RECV(buffer, length, MPI_REAL, process_b, PONG, MPI_COMM_WORLD, status, ierror)
      else if (my_rank .eq. process_b) then
        call MPI_RECV(buffer, length, MPI_REAL, process_a, PING, MPI_COMM_WORLD, status, ierror)
        call MPI_SEND(buffer, length, MPI_REAL, process_a, PONG, MPI_COMM_WORLD, ierror)
      end if

      write(*,*) 'Ping-pong on process complete - no deadlock on process', my_rank

      call MPI_FINALIZE(ierror)
!-----------------------------------------------------------------------
      end program ping_pong
!-----------------------------------------------------------------------
```

Example Ping - Pong

1. go to scheidegger/intro_to_hpc/MPI:

> cd scheidegger/intro_to_hpc/MPI

2. Have a look at the code

>vi 2.ping_poing.f90

3. compile by typing:

> make

4. run the code

>mpiexec -np 2 ./2.ping_pong.exec

Program fragment: parallel Integration

→ we integrate f(x) = 0.5*x in the domain [0,2]

```
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
   res=psum
   do i=1,size-1
      call MPI_Recv(tmp, &    ! receive buffer
                    1,   &    ! array length
                              ! data type
                    MPI_DOUBLE_PRECISION,&
                    i,   &    ! rank of source
                    0,   &    ! tag (unused here)
                    MPI_COMM_WORLD,& ! communicator
                    status,& ! status array (msg info)
                    ierror)
      res=res+tmp
   enddo
   write(*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
   call MPI_Send(psum,    &    ! send buffer
                 1,       &    ! message length
                 MPI_DOUBLE_PRECISION,&
                 0,       &    ! rank of destination
                 0,       &    ! tag (unused here)
                 MPI_COMM_WORLD,ierror)
endif
```

Split work/domain wrt all ranks available

Results on rank 0

Where is partial sum from?

Send partial sum from there

Example – Integration

1. go to scheidegger/intro_to_hpc/MPI:

> cd scheidegger/intro_to_hpc/MPI

2. Have a look at the code

>vi 2a.integrate.f90 (we integrate f(x) = 0.5*x in the domain [0,..,2])

3. compile by typing:

> make

4. run the code

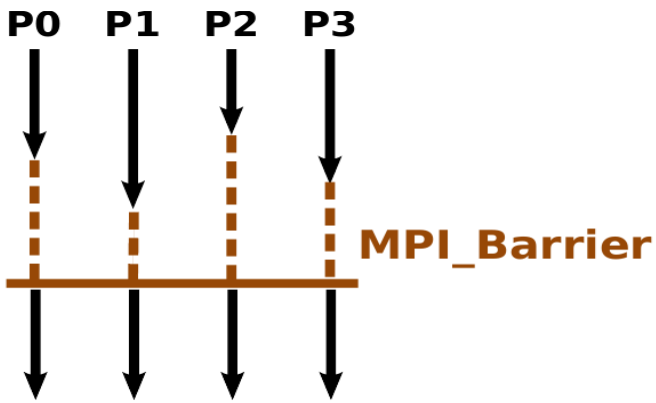
>mpiexec -np 2 ./2a.integrate.exec

→ **Play with different number of procs.**
→ **Change f(x) to sin(x), and also the integration bounds.**

Collective Communication

Collective communication in MPI (II)

Collective communication routines must involve all processes within the scope of a Communicator (e.g. MPI_COMM_WORLD).



Types of Collective Operations:

Synchronization:

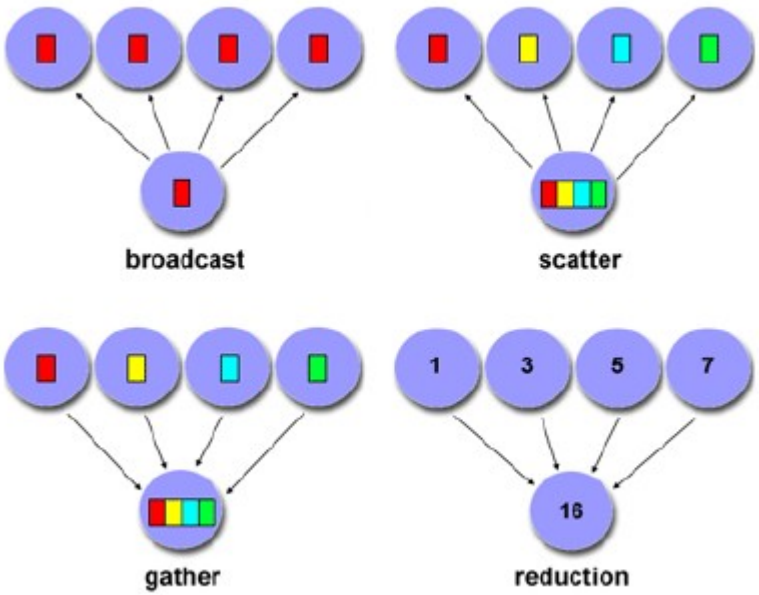
→ processes wait until all members of the group have reached the synchronization point (e.g. a barrier).

Data Movement:

→ broadcast, scatter/gather, all to all.

Collective Computation (reductions):

→ one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.



Reductions

The reduction operation allows to:

- Collect data from each process.

- Reduce the data to a single value.

- Store the result on the root processes.

- Store the result on all processes.

- Overlap communication and computation.

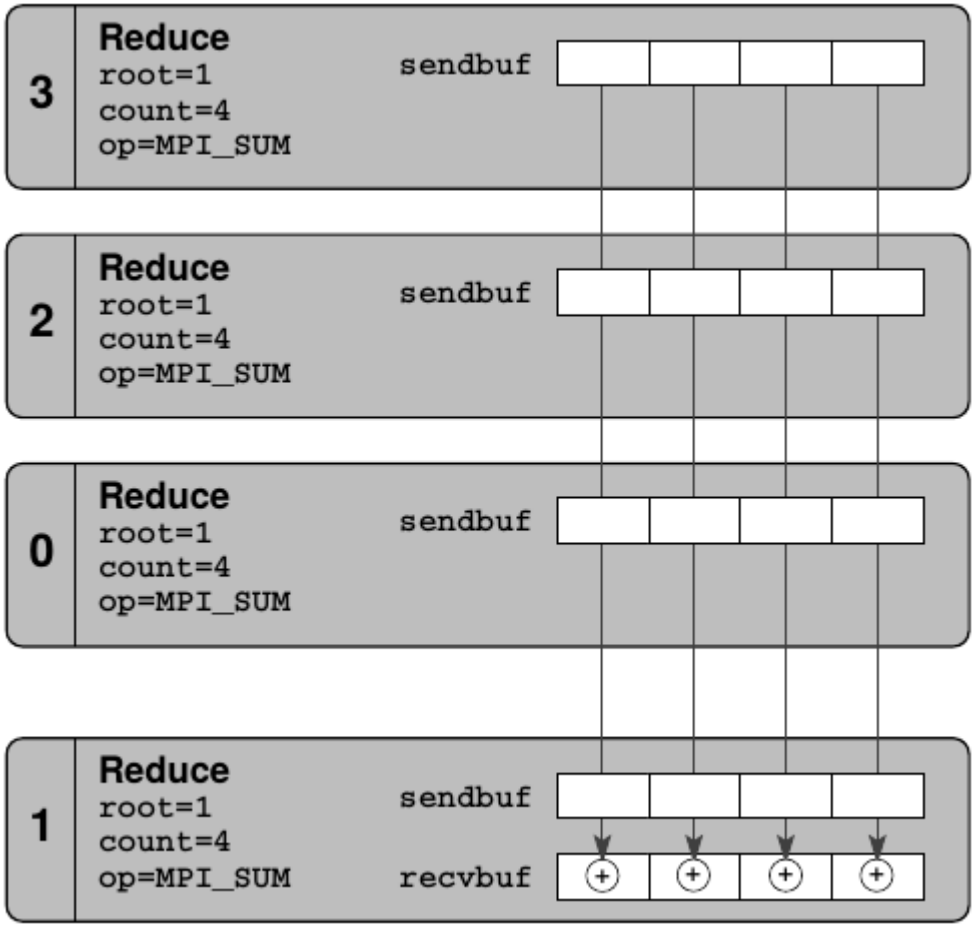
Reduction graphically

A reduction on an array of length count (a sum in this example) is performed by MPI_Reduce().

Every process must provide a send buffer.

The receive buffer argument is only used on the root process.

The local copy on root can be prevented by specifying MPI_IN_PLACE instead of a send buffer address.



Example: Reductions → MPI_SUM

- MPI, has mechanisms that make reductions much simpler (and in most cases) more efficient than looping over all ranks/collecting results.

- There are at the moment 12 predefined operators.

MPI op	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

```
integer :: count, datatype, op, root, comm, ierror
call MPI_Reduce(sendbuf,      ! send buffer
                recvbuf,      ! receive buffer
                count,        ! number of elements
                datatype,     ! MPI data type
                op,           ! MPI reduction operator
                root,         ! root rank
                comm,         ! communicator
                ierror)       ! return value
```

```
call MPI_Reduce(psum, &          ! send buffer (partial result)
                res,  &          ! recv buffer (final result @ root)
                1, &             ! array length
                MPI_DOUBLE_PRECISION, &
                MPI_SUM, &       ! type of operation
                0, &             ! root (accumulate result there)
                MPI_COMM_WORLD,ierror)
```

Specific call of MPI_Reduce() → **MPI_SUM**

Example – Reduction

```
      program reduce

      use mpi

      implicit none

      integer :: rank, input, result, ierror

!-----------------------------------------------------------------------
      call MPI_Init(ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

!-----------------------------------------------------------------------
      input = rank + 1

!-----------------------------------------------------------------------
!.....reduce the values of the different ranks in input to result of rank 0
!     with the operation sum (max, logical and)
      call MPI_Reduce(input, result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD, ierror)

      if (rank .eq. 0) then
        write (*,*) 'result', result
      end if

      call MPI_Finalize(ierror)
!-----------------------------------------------------------------------
      end program reduce
!-----------------------------------------------------------------------
```

Example – Reduction

1. go to zice17/scheidegger/intro_to_hpc/MPI:

> cd zice17/scheidegger/intro_to_hpc/MPI:

2. Have a look at the code

>vi 3.MPI_reduce.f90

3. compile by typing:

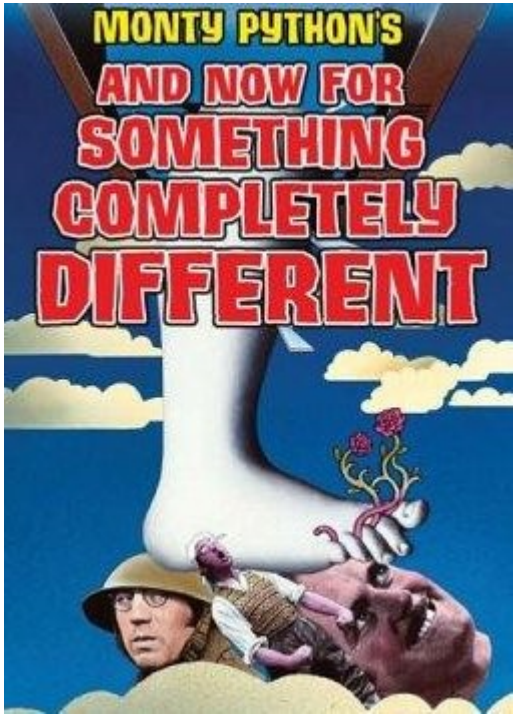
> make

4. run the code

>mpiexec -np 2 ./3.MPI_reduce.exec (experimpent with # processes)

5. change the “root” (line 27) from 0 to 1. What happens?

4. MPI & Python



4. MPI in Python

See **https://mpi4py.scipy.org**

→ **MPI for Python** supports convenient, pickle-based communication of generic Python object as well as fast, near C-speed, direct array data communication of buffer-provider objects (e.g., NumPy arrays).

Communication of generic Python objects:
You have to use **all-lowercase methods** (of the Comm class), like send(), recv(), bcast(). Note that isend() is available, but irecv() is not.

Collective calls like scatter(), gather(), allgather(), alltoall() expect/return a sequence of Comm.size elements at the root or all process. They return a single value, a list of Comm.size elements, or None.

Global reduction operations reduce() and allreduce() are naively implemented, the reduction is actually done at the designated root process or all processes.

“Hello World” in Python

Go to **zice17/scheidegger/intro_to_hpc/alphacruncher/MPI4PY**

Run with

> mpiexec -np 4 python helloworld.py

Make MPI available

```
#hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = MPI.COMM_WORLD.Get_size()
print "hello world from process ", rank, " from total ", size , "processes"
```

Point-to-Point Communication

>**zice17/scheidegger/intro_to_hpc/MPI4PY/alphacruncher/pointtopoint.py**

```
#passRandomDraw.py
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print "Process", rank, "drew the number", randNum[0]
        comm.Send(randNum, dest=0)

if rank == 0:
        print "Process", rank, "before receiving has the number", randNum[0]
        comm.Recv(randNum, source=1)
        print "Process", rank, "received the number", randNum[0]
```

MPI Broadcast in Python

> **zice17/scheidegger/intro_to_hpc/MPI4PY/alphacruncher/bcast.py**

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

#intialize
rand_num = numpy.zeros(1)

if rank == 0:
        rand_num[0] = numpy.random.uniform(0)

comm.Bcast(rand_num, root = 0)
print "Process", rank, "has the number", rand_num
```

MPI Reductions in Python

- Estimate integrals using the trapezoid rule.
- A range to be integrated is divided into many vertical slivers, and each sliver is approximated with a trapezoid.

$$area \approx \sum_{i=0}^{n} \frac{[f(a) + f(b)]}{2} \cdot \Delta x = \left[\frac{f(a) + f(b)}{2} + \sum_{i=0}^{n} f(a + i\Delta x) + f(a + (i+1)\Delta x) \right] \cdot \Delta x$$

MPI Reductions in Python

```
import numpy
import sys
from mpi4py import MPI
from mpi4py.MPI import ANY_SOURCE

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

#takes in command-line arguments [a,b,n]
a = float(sys.argv[1])
b = float(sys.argv[2])
n = int(sys.argv[3])

#we arbitrarily define a function to integrate
def f(x):
        return x*x

#this is the serial version of the trapezoidal rule
#parallelization occurs by dividing the range among processes
def integrateRange(a, b, n):
        integral = -(f(a) + f(b))/2.0
        # n+1 endpoints, but n trapazoids
        for x in numpy.linspace(a,b,n+1):
                        integral = integral + f(x)
        integral = integral* (b-a)/n
        return integral


#h is the step size. n is the total number of trapezoids
h = (b-a)/n
#local_n is the number of trapezoids each process will calculate
#note that size must divide n
local_n = n/size

#we calculate the interval that each process handles
#local_a is the starting point and local_b is the endpoint
local_a = a + rank*local_n*h
local_b = local_a + local_n*h

#initializing variables. mpi4py requires that we pass numpy objects.
integral = numpy.zeros(1)
total = numpy.zeros(1)

# perform local computation. Each process integrates its own interval
integral[0] = integrateRange(local_a, local_b, local_n)

# communication
# root node receives results with a collective "reduce"
comm.Reduce(integral, total, op=MPI.SUM, root=0)

# root process prints results
if comm.rank == 0:
        print "With n =", n, "trapezoids, our estimate of the integral from"\
        , a, "to", b, "is", total
```

>zice17/scheidegger/intro_to_hpc/MPI4PY/alphacruncher/reduction.py

Run with
> mpiexec -n 4 python reduction.py **a b N**
→ integration range **[a,b]**, discretiztion **N**

> mpiexec -n 4 python reduction.py 0.0 1.0 1000

OUTPUT = ???

f(x) = x^2

Reduction

Questions?

1. Advice – RTFM
https://en.wikipedia.org/wiki/RTFM

2. Advice – http://lmgtfy.com/
http://lmgtfy.com/?q=message+passing+interface