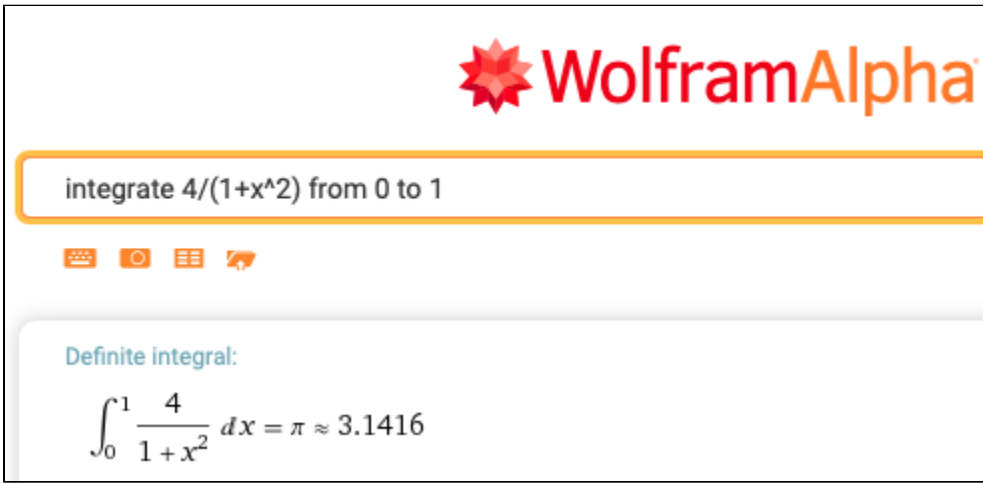


# MPI + OpenMP (or OpenACC )

## Toy problem: compute PI

( 10 minutes )



The screenshot shows the WolframAlpha interface. At the top, the WolframAlpha logo is displayed. Below it, a search bar contains the text "integrate 4/(1+x^2) from 0 to 1". Below the search bar, there are several icons for different input methods. Below the icons, the text "Definite integral:" is shown. Below that, the integral is displayed as  $\int_0^1 \frac{4}{1+x^2} dx = \pi \approx 3.1416$ .

Code description: <https://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/pi/C/main.html> :

The algorithm suggested here is chosen for its simplicity. The method evaluates the integral of  $4/(1+x^2)$  between 0 and 1. The method is simple: the integral is approximated by a sum of  $n$  intervals; the approximation to the integral in each interval is  $(1/n)*4/(1+x^2)$ . The master process (rank 0) asks the user for the number of intervals; the master should then broadcast this number to all of the other processes. Each process then adds up every  $n$ 'th interval ( $x = \text{rank}/n, \text{rank}/n + \text{size}/n, \dots$ ). Finally, the sums computed by each process are added together using a reduction.

**cpi.c**

```
1 #include <stdio.h>
2 #include <mpi.h>
3 #include <math.h>
4
5 int main(int argc, char *argv[])
6 {
7     int n, myid, numprocs, i;
8     double PI25DT = 3.141592653589793238462643;
9     double mypi, pi, h, sum, x;
10    MPI_Init(&argc, &argv);
11    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
12    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
13    while (1) {
14        if (myid == 0) {
15            printf("Enter the number of intervals: (0 quits) ");
16            scanf("%d", &n);
17        }
18        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
19        if (n == 0)
20            break;
21        else {
22            h = 1.0 / (double) n;
23            sum = 0.0;
24            for (i = myid + 1; i <= n; i += numprocs) {
25                x = h * ((double)i - 0.5);
26                sum += (4.0 / (1.0 + x*x));
27            }
28            mypi = h * sum;
29            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
30                    MPI_COMM_WORLD);
31            if (myid == 0)
32                printf("pi is approximately %.16f, Error is %.16f\n",
33                    pi, fabs(pi - PI25DT));
34        }
35    }
36    MPI_Finalize();
37    return 0;
38 }
```

## cpi.f90

```
1  program main
2  use mpi
3  double precision  PI25DT
4  parameter         (PI25DT = 3.141592653589793238462643d0)
5  double precision  mypi, pi, h, sum, x, f, a
6  integer n, myid, numprocs, i, ierr
7  !
8  !           function to integrate
9  f(a) = 4.d0 / (1.d0 + a*a)
10
11 call MPI_INIT(ierr)
12 call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
13 call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
14
15 do
16   if (myid .eq. 0) then
17     print *, 'Enter the number of intervals: (0 quits) '
18     read(*,*) n
19   endif
20   !           broadcast n
21   call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
22   !           check for quit signal
23   if (n .le. 0) exit
24   !           calculate the interval size
25   h = 1.0d0/n
26   sum = 0.0d0
27   do i = myid+1, n, numprocs
28     x = h * (dble(i) - 0.5d0)
29     sum = sum + f(x)
30   enddo
31   mypi = h * sum
32   !           collect all the partial sums
33   call MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION, &
34     MPI_SUM, 0, MPI_COMM_WORLD, ierr)
35   !           node 0 prints the answer.
36   if (myid .eq. 0) then
37     print *, 'pi is ', pi, ' Error is', abs(pi - PI25DT)
38   endif
39 enddo
40 call MPI_FINALIZE(ierr)
41 end
```

## Exercise : Add OpenMP directive(s) to the compute pi code.

- Use one of the MPI codes above or follow the "Code examples" link in the References below. ( 15 minutes )
  - Recall your MPI compiler commands and the flags for enabling OpenMP directives, you will want to mix those, on a generic linux mpich2-gnu for example:
    - `mpif90 -fopenmp cpi.f90`
    - don't forget to set `OMP_NUM_THREADS` at runtime
- Core placement will be important for performance. MPI's default rank-per-core may not leave any cores free for threads.
  - <https://bluewaters.ncsa.illinois.edu/using-aprun>
  - <https://portal.tacc.utexas.edu/user-guides/stampede2#launching-one-hybrid-mpithreads-application>
  - <https://docs.nersc.gov/jobs/affinity/> , <https://docs.nersc.gov/jobs/examples/>
  - [https://bluewaters.ncsa.illinois.edu/liferay-content/document-library/Documentation%20Documents/aprun/hello\\_world.c](https://bluewaters.ncsa.illinois.edu/liferay-content/document-library/Documentation%20Documents/aprun/hello_world.c)
  - Know the hardware. A Blue Waters XE compute node has the following arrangement:

```

numactl

arnoldg@nid25262:~> numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 16383 MB
node 0 free: 496 MB
node 1 cpus: 8 9 10 11 12 13 14 15
node 1 size: 16384 MB
node 1 free: 393 MB
node 2 cpus: 16 17 18 19 20 21 22 23
node 2 size: 16384 MB
node 2 free: 638 MB
node 3 cpus: 24 25 26 27 28 29 30 31
node 3 size: 16384 MB
node 3 free: 5249 MB
node distances:
node  0  1  2  3
  0:  10  13  13  13
  1:  13  10  13  13
  2:  13  13  10  13
  3:  13  13  13  10

```

- The codes are straight from the mpich examples, you may want to modify yours such that it does not prompt for the number of sub-intervals if you want to run it purely in batch-mode.
- Some people should apply OpenMP to the outer loop ( let's learn from potential mistakes ).
- Some people should apply OpenMP to the inner loop ( this would seem to make the most sense ).
- see also: [https://computing.llnl.gov/tutorials/openMP/samples/C/omp\\_reduction.c](https://computing.llnl.gov/tutorials/openMP/samples/C/omp_reduction.c) , [https://computing.llnl.gov/tutorials/openMP/samples/Fortran/omp\\_reduction.f](https://computing.llnl.gov/tutorials/openMP/samples/Fortran/omp_reduction.f)



mpi+openmp.tar

Discussion of outcomes from exercise: what worked, what didn't, and failure modes. ( 10 minutes )

- Does a bad MPI+OpenMP implementation run with OMP\_NUM\_THREADS=1 ?
- Does it fail with all MPI implementations ?

### Why bother mixing in OpenMP and MPI together ?

- reduce the number of MPI ranks
  - fewer communications (possibly offset by larger communications)
  - better scaling on a system with an imperfect network (lots of over-subscription )
- there's a good OpenMP implementation with loops containing a lot of work
- OpenMP OpenACC and access to an accelerator
- *for the algorithm under review*, OpenMP has an advantage over MPI ( shared-memory array accesses or cache effects )

### The MPI standard and threads

( 10 minutes )

MPI draft standard section 3.5 Semantics of Point-to-Point Communication:

If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multithreaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

MPI draft standard section 12.4 MPI and Threads (excerpts)

This section specifies the interaction between MPI calls and threads. The section lists minimal requirements for **thread compliant** MPI implementations and defines functions that can be used for initializing the thread environment. MPI may be implemented in environments where threads are not supported or perform poorly. Therefore, MPI implementations are not required to be thread compliant as defined in this section. Regardless of whether or not the MPI implementation is thread compliant, MPI\_INITIALIZED, MPI\_FINALIZED, MPI\_QUERY\_THREAD, MPI\_IS\_THREAD\_MAIN, MPI\_GET\_VERSION and MPI\_GET\_LIBRARY\_VERSION must always be thread-safe. When a thread is executing one of these routines, if another concurrently running thread also makes an MPI call, the outcome will be as if the calls executed in some order.

In a thread-compliant implementation, an MPI process is a process that may be multi-threaded. Each thread can issue MPI calls; however, threads are not separately addressable: a rank in a send or receive call identifies a process, not a thread. A message sent to a process can be received by any thread in this process.

*Advice to users.*

It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. ( *End of advice to users.*) The two main requirements for a thread-compliant implementation are listed below.

1. All MPI calls are thread-safe, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

Initialization and Completion The call to MPI\_FINALIZE should occur on the same thread that initialized MPI. We call this thread the **main thread**. The call should occur only after all process threads have completed their MPI calls, and have no pending communications or I/O operations.

References
Code examples: <a href="https://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/simplempi/index.html">https://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/simplempi/index.html</a>
Hybrid Programming example <a href="https://psc.edu/images/xsedetraining/BootCamp/Hybrid_Programming.pdf">https://psc.edu/images/xsedetraining/BootCamp/Hybrid_Programming.pdf</a>
Hybrid Programming course <a href="#">7 Combining MPI and OpenMP Parallelism ( NCSA CI-TUTOR )</a>
Open MP tutorial <a href="https://computing.llnl.gov/tutorials/openMP">https://computing.llnl.gov/tutorials/openMP</a> ( LLNL HPC tutorials )
Hybrid Memory discussion, see "Hybrid Model" <a href="https://computing.llnl.gov/tutorials/parallel_comp/#HybridMemory">https://computing.llnl.gov/tutorials/parallel_comp/#HybridMemory</a>
MPI Standard <a href="https://www.mpi-forum.org/docs/drafts/mpi-2018-draft-report.pdf">https://www.mpi-forum.org/docs/drafts/mpi-2018-draft-report.pdf</a>
Sample hybrid solutions <a href="https://github.com/kc9qey/mpi_openmp">https://github.com/kc9qey/mpi_openmp</a>