# Introduction to parallel programming

*SULI seminar series*

*June 20, 2019*

*Stéphane Ethier*

*(ethier@pppl.gov)*

*Princeton Plasma Physics Lab*

# Why Parallel Computing?

## Why not run *n* instances of my code? Isn't that parallel computing?

**YES… but**

- You want to speed up your calculation because it takes a week to run!
- Your problem size is too large to fit in the memory of a single node
- Want to use those extra cores on your "multicore" processor
- **Solution:**
  - Split the work between several processor cores so that they can work in parallel
  - Exchange data between them when needed
- **How?**
  - Message Passing Interface (**MPI**) on distributed memory systems (works also on shared memory nodes)
  - **OpenMP** directives on shared memory node
  - and some other methods not as popular (pthreads, Intel TBB, Fortran Co-Arrays)

# Big Science requires Big Computers

| Rank | Site | System | Rmax (TFlop/s) | Rpeak (TFlop/s) |
|---|---|---|---|---|
| 1 | DOE/SC/Oak Ridge National Laboratory<br>United States | **Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband<br>IBM | 143,500.0 | 200,794.9 |
| 2 | DOE/NNSA/LLNL<br>United States | **Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband<br>IBM / NVIDIA / Mellanox | 94,640.0 | 125,712.0 |
| 3 | National Supercomputing Center in Wuxi<br>China | **Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway<br>NRCPC | 93,014.6 | 125,435.9 |
| 4 | National Super Computer Center in Guangzhou<br>China | **Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000<br>NUDT | 61,444.5 | 100,678.7 |
| 5 | Swiss National Supercomputing Centre (CSCS)<br>Switzerland | **Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100<br>Cray Inc. | 21,230.0 | 27,154.3 |
| 6 | DOE/NNSA/LANL/SNL<br>United States | **Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect<br>Cray Inc. | 20,158.7 | 41,461.2 |
| 7 | National Institute of Advanced Industrial Science and Technology (AIST)<br>Japan | **AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR<br>Fujitsu | 19,880.0 | 32,576.6 |
| 8 | Leibniz Rechenzentrum<br>Germany | **SuperMUC-NG** - ThinkSystem SD530, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path<br>Lenovo | 19,476.6 | 30,855.2 |
| 9 | DOE/SC/Oak Ridge National Laboratory<br>United States | **Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x<br>Cray Inc. | 17,590.0 | 27,112.5 |
| 10 | DOE/NNSA/LLNL<br>United States | **Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom<br>IBM | 17,173.2 | 20,132.7 |

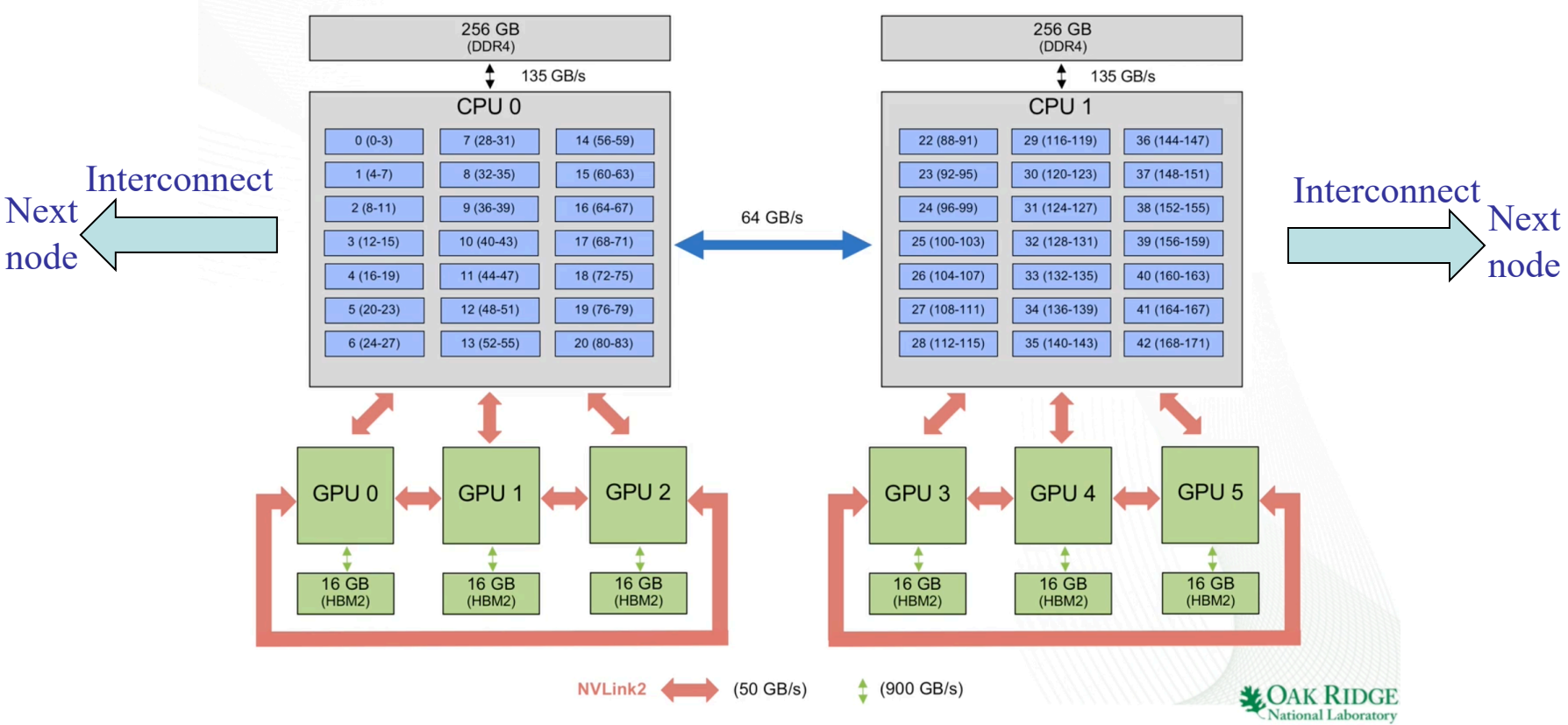
**15,312 nodes, 979,968 cores**

Nov 2018 List of the top supercomputers in the world (www.top500.org)

# SUMMIT - #1 World Supercomputer (200 PFLOPS) at the Oak Ridge Leadership Computing Facility

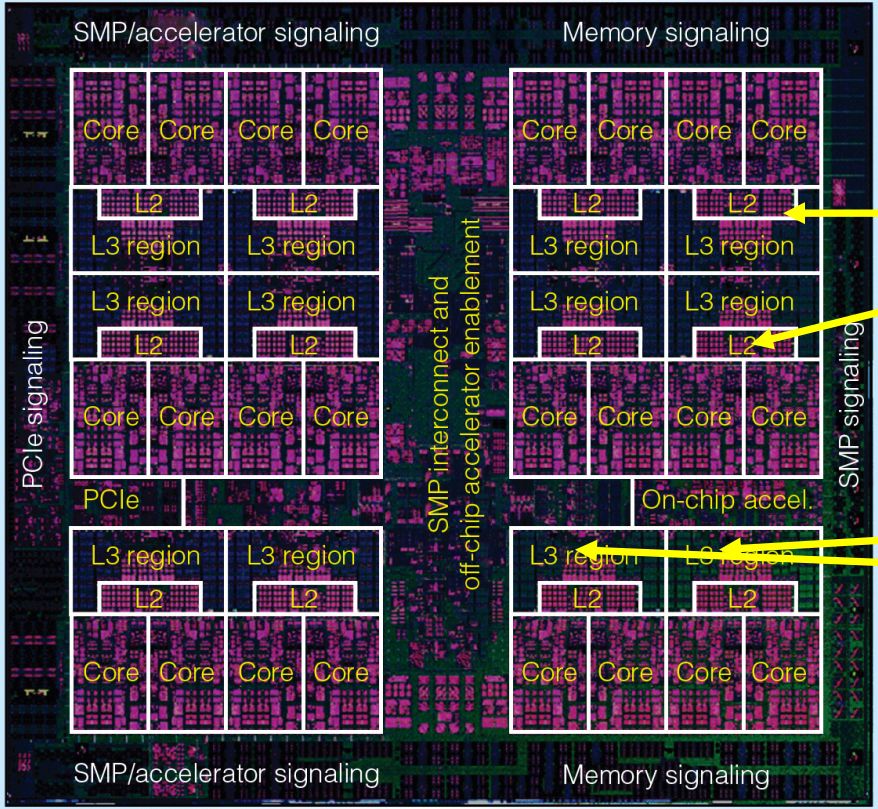
- 4,600 IBM AC922 nodes
- Each node contains:
  - 2 IBM Power 9 processors (42 cores)
  - 6 Nvidia V100 Tesla GPUs
  - 512 GB DDR4 memory for CPUs
  - 96 GB HBM2 memory for GPUs
  - 1.6 TB NVMe storage
- Power 9 processor
  - SIMD Multi-Core (21)
  - 4 hardware threads per core
  - 32 kB L1 private cache, 512 kB shared L2, 10 MB shared L3
- V100 “Volta” GPU
  - 7.8 TFLOPS double precision (X 6)
- Dual-rail EDR Infiniband interconnect between nodes

# SUMMIT node architecture




OAK RIDGE National Laboratory

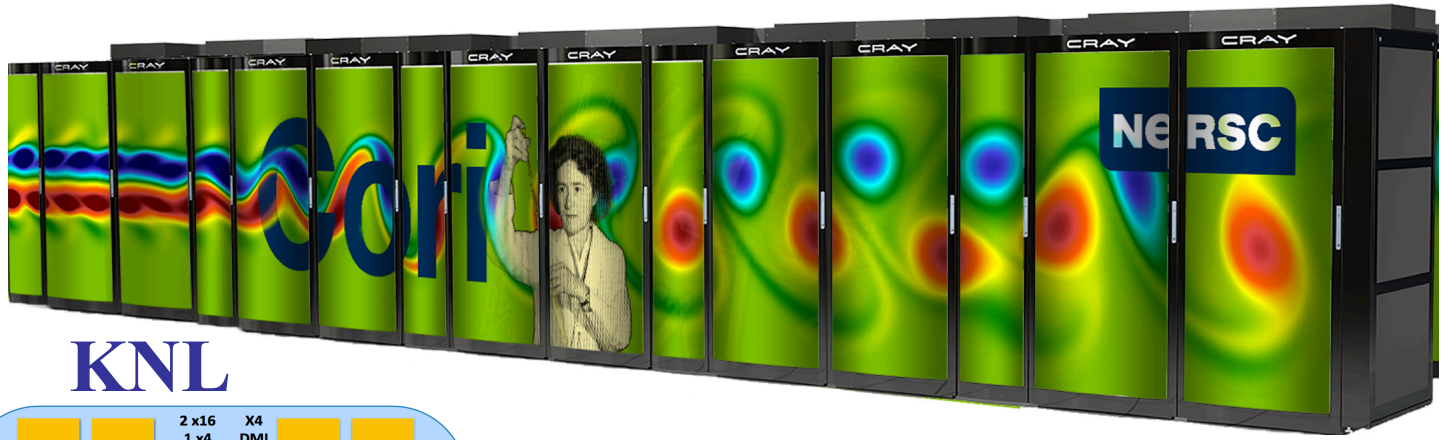
# Power9 CPU architecture



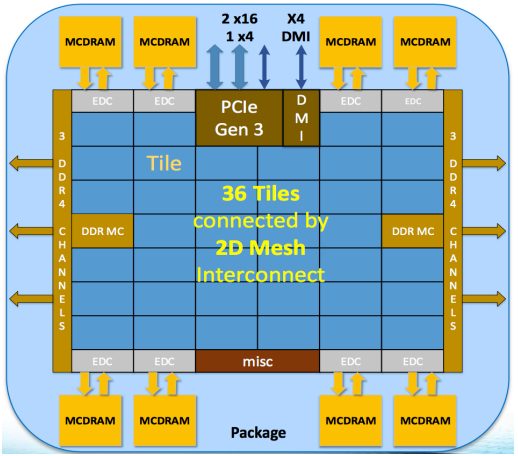

512 kB shared L2 cache

10 MB shared L3 cache

# Cray XC40 "Cori" at the National Energy Research Scientific Computing Center (NERSC)




KNL




- 9,688 Intel Xeon Phi processors "Knights Landing" nodes
- 68 cores per KNL node (single socket) with 4 hardware threads per core (total of 272 threads per KNL)
- 29.1 Pflops peak (3 Tflops/node)
- Cray Aries interconnect for communication between nodes → 36 lanes PCIe gen 3

*Let's start with the original parallelism*

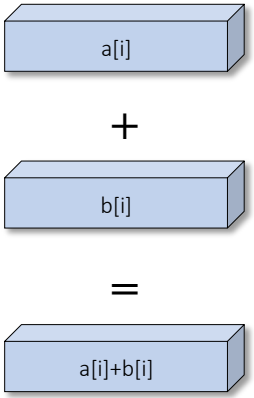
It's back in fashion!

# Hardware-level parallelism: VECTORIZATION

At the inner-most loop level

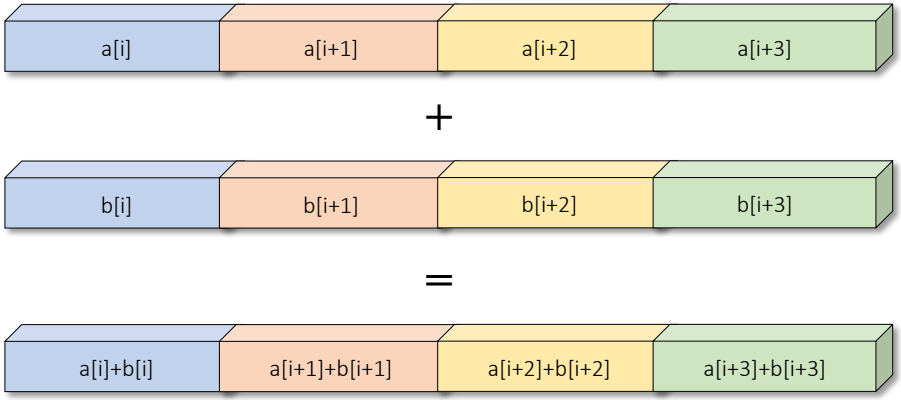
```
for (int i=0; i<N; i++) {
    c[i]=a[i]+b[i];  }
```

Scalar (SISD)




One operation
One result

Vector (SIMD)




One operation
*Multiple* results

# Vectorization terminology

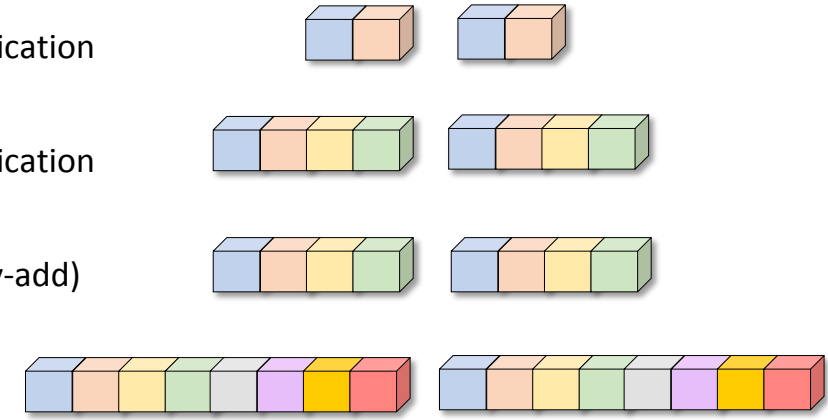
- SIMD: Single Instruction Multiple Data
- SSE: Streaming SIMD Extensions
- AVX: Advanced Vector Extension

# How to vectorize?

- The compiler will take care of it…
- You just need a high-enough level of optimization
  - ”–O3” is usually sufficient (gcc –O3, pgcc –O3, icc –O3)
  - Check your compiler’s documentation
- You just need to make sure that the loops in your code are arranged in such a way that the compiler can safely generate vector instructions
  - Iterations (loop steps) need to be independent of each other
  - Avoid branching (if-else) as much as possible
  - Don’t make the loops too large (you can always split them)

# Intel Xeon (Server) Architecture Codenames

- Number of FLOP/s depends on the chip architecture
- Double Precision (64 bit double)
  - Nehalem/Westmere (SSE):
    - 4 DP FLOPs/cycle:  128-bit addition + 128-bit multiplication
  - Ivybridge/Sandybridge (AVX)
    - 8 DP FLOPs/cycle:  256-bit addition + 256-bit multiplication
  - Haswell/Broadwell (AVX2)
    - 16 DP FLOPs/cycle: two, 256-bit FMA (fused multiply-add)
  - KNL/Skylake (AVX-512)
    - 32 DP FLOPs/cycle: two, 512-bit FMA
- FMA = (a × b + c)
- Twice as many if single precision (32-bit float)

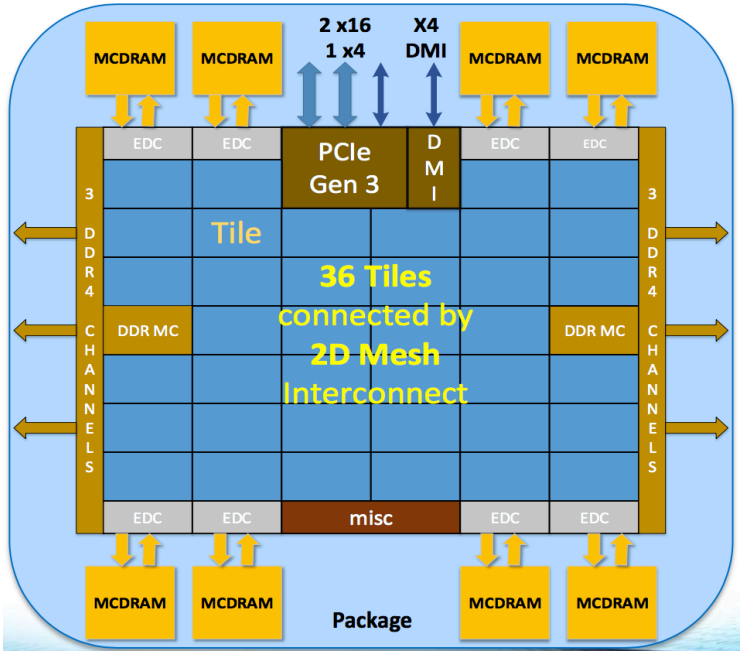


*Okay… I vectorized as much as I could*
*What's next?*

Let's say that your problem size fits perfectly within a node

# Shared memory parallelism

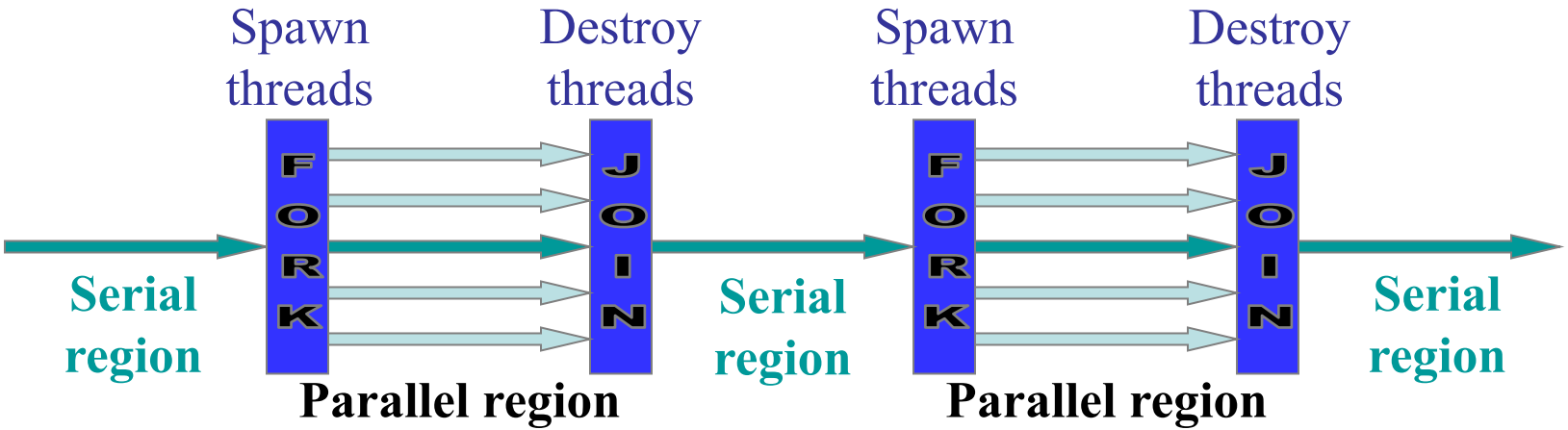
## Intel Xeon Phi architecture




- KNL "node" consists of 68 cores arranged in 4 quadrants
- All 68 cores share the memory
- Is there a way to split the work between all of these cores?

# Solution: Multi-threading

- Multi-threaded parallelism (parallelism-on-demand)
- Fork-and-Join  model (although we say “spawn” for **threads** and “fork” for **processes**).



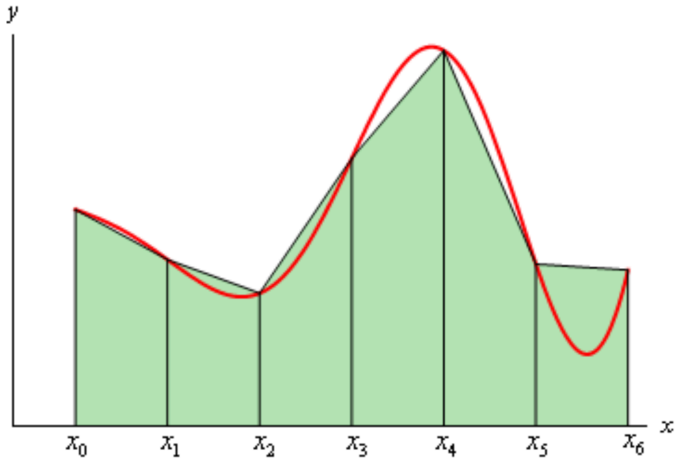
# Process and thread: what's the difference?

- You need an existing process to create a thread.
- Each process has at least one thread of execution.
- A process has its own virtual memory space that cannot be accessed by other processes running on the same or on a different processor.
- All threads created by a process share the virtual address space of that process. They read and write to the same address space in memory. They also share the same process and user ids, file descriptors, and signal handlers. However, they have their own program counter value and stack pointer, and can run independently on several processor cores.

# Example: Calculate pi by numerical integration

```
#include <stdio.h>
#include <stdlib.h>
long num_steps = 1000000;
double step = 1.0/1000000.0;
int main() {
 int i;
 double x, pi, sum = 0.0;
```

```
for(i = 0; i < num_steps; ++i) {
      x = (i-0.5)*step;
      sum += 4.0/(1.0+x*x);
}
pi = step*sum; printf("PI value =
%f\n", pi);
}
```



# Loop-level multi-threading with OpenMP

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
long num_steps = 100000;
double step = 1.0/100000.0;
int main() {
 int i;
 double x, pi, sum = 0.0;
```

```
#pragma omp parallel private(x) {
#pragma omp for reduction(+:sum)
 for(i = 0; i < num_steps; ++i) {
     x = (i-0.5)*step;
     sum += 4.0/(1.0+x*x);
 }
}
pi = step*sum; printf("PI value =
%f\n", pi);
}
```

OpenMP is a directive-based Programming model

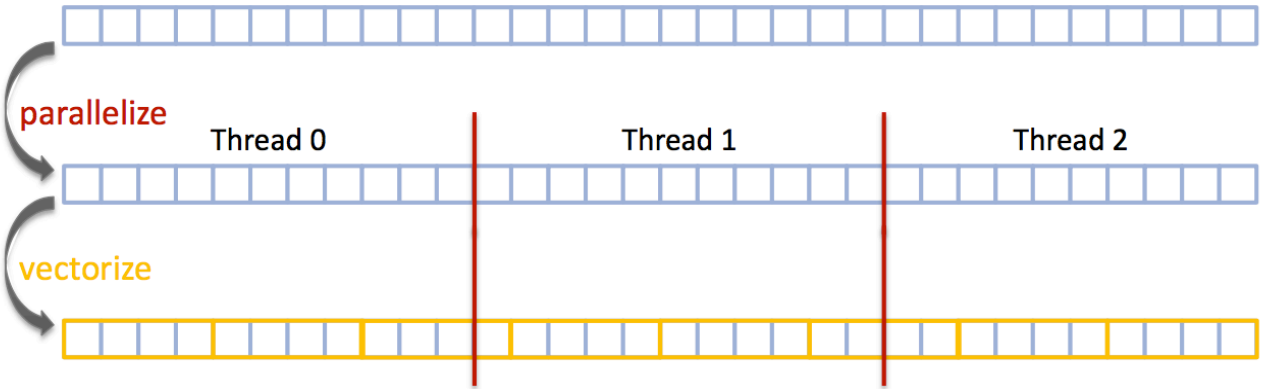
www.openmp.org

# Telling the compiler to process the directives

- Most, if not all compilers can process OpenMP directives and generate appropriate multi-threaded code.
- Be careful though. Some vendors are selling different versions of their compilers and the OpenMP support can come under a “parallel” or “high performance” version.
- This is achieved by using an option that instructs the compiler to activate and interpret all OpenMP directives. Here are a few examples:
  - PGI compiler: **pgf90 –mp** and **pgcc –mp**
  - IBM xlf: **xlf90_r -qsmp=omp** and **xlc_r –qsmp=omp**
  - Linux gcc: **gcc –fopenmp**
  - Intel (Linux): **icc –openmp** and **ifort -openmp**
- It is important to use the “thread-safe” versions of the XL compilers on the IBM systems (Blue Gene and Power systems). They have an extra “_r” added to their names (xlc_r, xlf90_r)

# Example of OpenMP *simd* directive

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```




- Make sure to use "chunk sizes" that are multiples of the SIMD length (512 bits) for best performance
- If you can't, add the simd modifier that can automatically adjust the chunk size to match the simd length

**`schedule(simd:static,5)`**
**(OpenMP 4.5)**

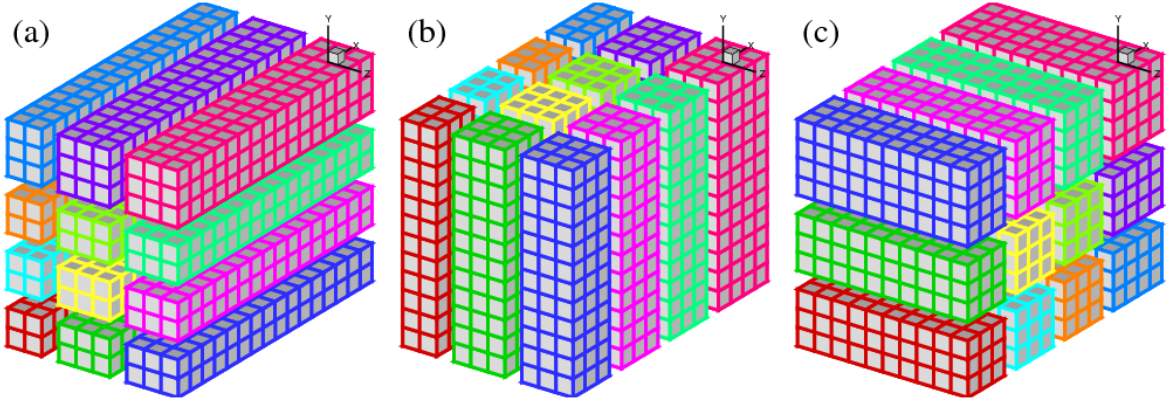
https://doc.itc.rwth-aachen.de/download/attachments/28344675/SIMD%20Vectorization%20with%20OpenMP.PDF?version=1&modificationDate=1480523704000&api=v2

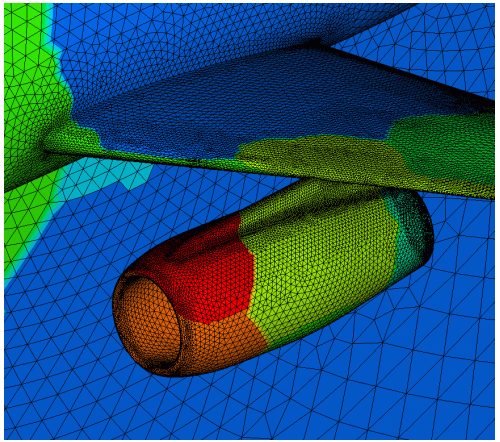
# What if my problem size is too large to fit on one node? and too slow to finish in a reasonable time?

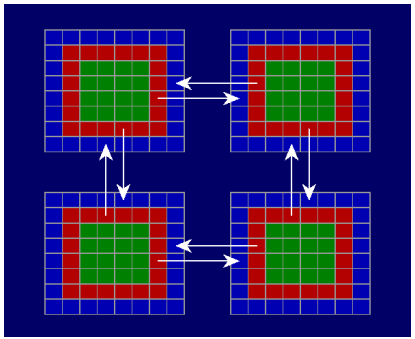
- Need to use several nodes!
- Maybe thousands of them…
- This is called ***"distributed memory parallelism"***
- How to split the work between nodes/processors?
- Memory is not shared across nodes so how will the threads(?) exchange data?

# How to split the work between processors?
# ***Domain Decomposition***

- Most widely used method for grid-based calculations

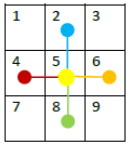




# How to split the work between processors?
## *Split matrix elements in PDE solves*

- See PETSc project: https://www.mcs.anl.gov/petsc/




(a)

| Grid Element | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 57 58 93 94 / 59 60 95 96 | | | 125 126 / 127 128 | | | | | |
| 2 | 25 26 61 62 97 98 / 27 28 63 64 99 100 | | | | 129 130 / 131 132 | | | | |
| 3 | | 29 30 65 66 101 102 / 31 32 67 68 103 104 | | | | 133 134 / 135 136 | | | |
| 4 | 1 2 / 3 4 | | 33 34 69 70 105 106 / 35 36 71 72 107 108 | | | | 137 138 / 139 140 | | |
| 5 | | 5 6 / 7 8 | | 37 38 73 74 109 110 / 39 40 75 76 111 112 | | | | 141 142 / 143 144 | |
| 6 | | | 9 10 / 11 12 | | 41 42 77 78 113 114 / 43 44 79 80 115 116 | | | | 145 146 / 147 148 |
| 7 | | | | 13 14 / 15 16 | | 45 46 81 82 117 118 / 47 48 83 84 119 120 | | | |
| 8 | | | | | 17 18 / 19 20 | | 49 50 85 86 121 122 / 51 52 87 88 123 124 | | |
| 9 | | | | | | 21 22 / 23 24 | | 53 54 89 90 / 55 56 91 92 | |

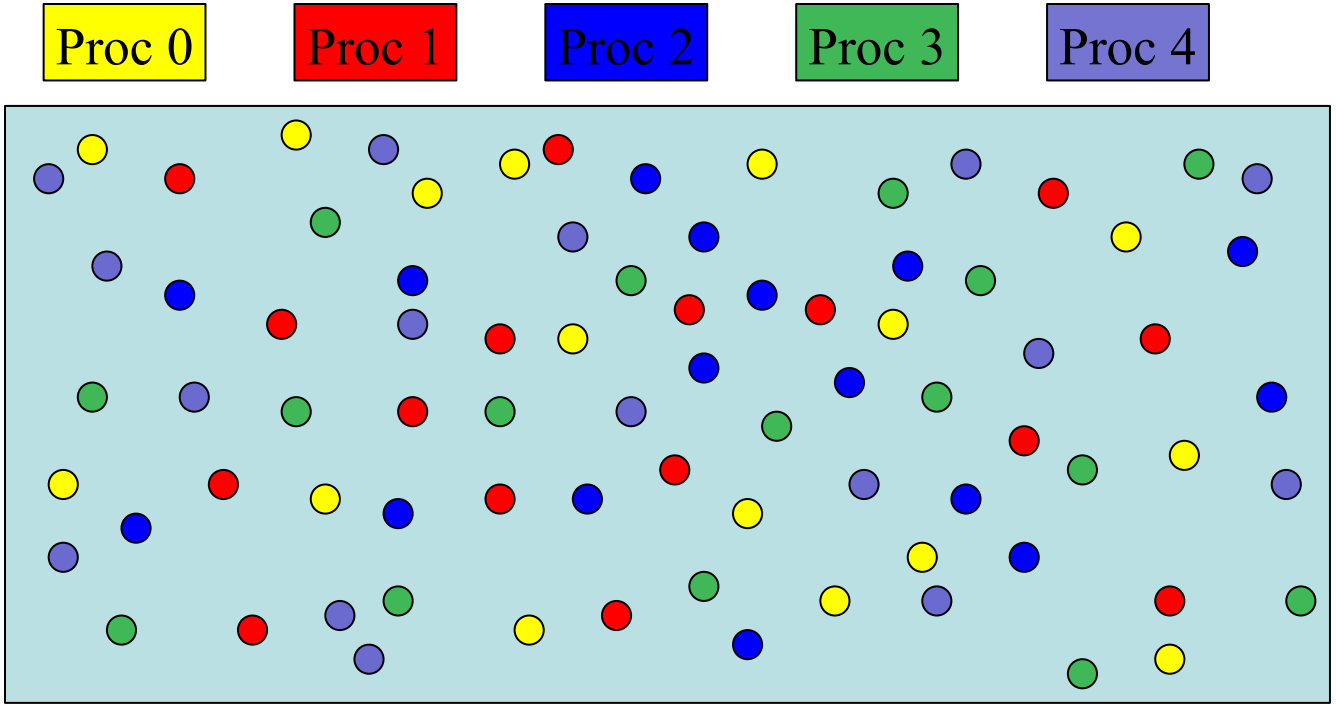
(b)

| -3 | -1 | 0 | 1 | 3 |
|---|---|---|---|---|
| 00 00 00 00 57 58 93 94 125 126 / 00 00 00 00 59 60 95 96 127 128 | | | | |
| 00 00 25 26 61 62 97 98 129 130 / 00 00 27 28 63 64 99 100 131 132 | | | | |
| 00 00 29 30 65 66 101 102 133 134 / 00 00 31 32 67 68 103 104 135 136 | | | | |
| 1 2 33 34 69 70 105 106 137 138 / 3 4 35 36 71 72 107 108 139 140 | | | | |
| 5 6 37 38 73 74 109 110 141 142 / 7 8 39 40 75 76 111 112 143 144 | | | | |
| 9 10 41 42 77 78 113 114 145 146 / 11 12 43 44 79 80 115 116 147 148 | | | | |
| 13 14 45 46 81 82 117 118 00 00 / 15 16 47 48 83 84 119 120 00 00 | | | | |
| 17 18 49 50 85 86 121 122 00 00 / 19 20 51 52 87 88 123 124 00 00 | | | | |
| 21 22 53 54 89 90 00 00 00 00 / 23 24 55 56 91 92 00 00 00 00 | | | | |

(c)

# How to split the work between processors?
## *"Coloring"*

- Useful for particle simulations



*Okay… but how do I do that???*

# MPI – *Message Passing Interface*

**Context: Distributed memory parallel computers**

- Each process has its own memory and cannot access the memory of other processes
- A copy of the same executable runs on each MPI process (processor core)
- Any data to be shared must be explicitly transmitted from one to another

**Most message passing programs use the *single program multiple data* (SPMD) model**

- Each process executes the same set of instructions
- Parallelization is achieved by letting each processor core operate on a different piece of data

# What is MPI?

- MPI stands for Message Passing Interface.
- It is a message-passing specification, a standard, for the vendors to implement.
- In practice, MPI is a set of functions (C) and subroutines (Fortran) used for exchanging data between processes.
- An MPI library exists on ALL parallel computing platforms so it is highly portable.
- The scalability of MPI is not limited by the number of processors/cores on one computation node, as opposed to shared memory parallel models.
- Also available for Python (mpi4py.scipy.org), R (Rmpi), Lua, and Julia! (if you can call C functions, you can use MPI...)

# Reasons for using MPI

- Scalability
- Portability
- ***WORKS ON SHARED MEMORY NODES AS WELL!!***

# Compiling and linking an MPI code

- Need to tell the compiler where to find the MPI include files and how to link to the MPI libraries.
- Fortunately, most MPI implementations come with scripts that take care of these issues:
  - mpicc mpi_code.c –o a.out
  - mpiCC mpi_code_C++.C –o a.out
  - mpif90 mpi_code.f90 –o a.out
- Two widely used (and free) MPI implementations on Linux clusters are:
  - MPICH (http://www-unix.mcs.anl.gov/mpi/mpich)
  - OPENMPI (http://www.openmpi.org)

# How to run an MPI executable

- The implementation supplies scripts to launch the MPI parallel calculation, for example:

```
mpirun -np #proc a.out
mpiexec -n #proc a.out
```

} MPICH, OPENMPI

```
aprun -size #proc a.out (Cray XT)
srun -n #proc a.out (SLURM batch system)
```

- A copy of the same program runs on each processor core within its own process (private address space).
- Each process works on a subset of the problem.
- Exchange data when needed
  - Can be exchanged through the network interconnect
  - Or through the shared memory on SMP machines (Bus?)
- Easy to do coarse grain parallelism = scalable

# MPI Communicators

- A communicator is an identifier associated with a group of processes
  - Each process has a unique rank within a specific communicator (the rank starts from 0 and has a maximum value of (nprocesses-1) ).
  - Internal mapping of processes to processing units
  - Always required when initiating a communication by calling an MPI function or routine.
- Default communicator MPI_COMM_WORLD, which contains all available processes.
- Several communicators can coexist
  - A process can belong to different communicators at the same time, but has a unique rank in each communicator

# A sample MPI program in C

```
#include "mpi.h"
int main( int argc, char *argv[] )
{
  int nproc, myrank;
  /* Initialize MPI */
    MPI_Init(&argc,&argv);
  /* Get the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  /* Get my process number (rank) */
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    Do work and make message passing calls…

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

# Basic MPI calls to exchange data

- Point-to-Point communications
  - Only 2 processes exchange data
  - It is the basic operation of all MPI calls
- Collective communications
  - A single call handles the communication between all the processes in a communicator
  - There are 3 types of collective communications
    - Data movement (e.g. MPI_Bcast)
    - Reduction (e.g. MPI_Reduce)
    - Synchronization: MPI_Barrier

# Point-to-point communication

**Point to point:** 2 processes at a time

```
MPI_Send(buf,count,datatype,dest,tag,comm,ierr)

MPI_Recv(buf,count,datatype,source,tag,comm,status,ierr)
```

```
MPI_Sendrecv(sendbuf,sendcount,sendtype,dest,sendtag,
    recvbuf,recvcount,recvtype,source,recvtag,comm,status,ierr)
```

where the datatypes are:

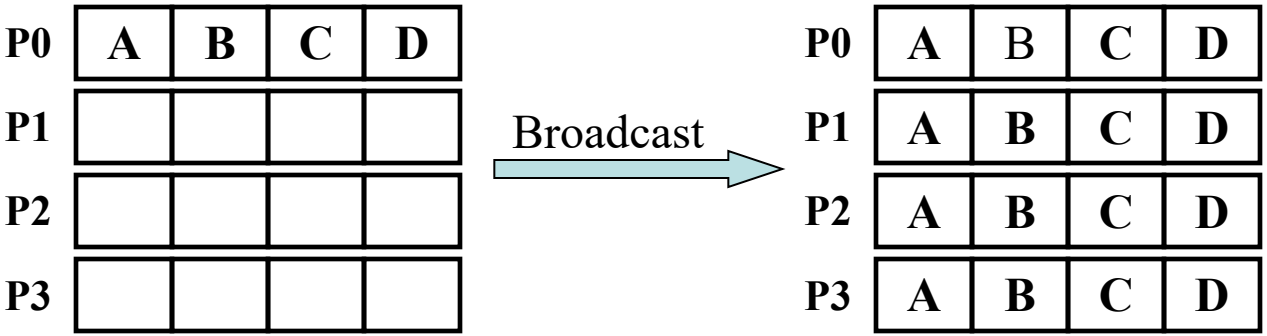
FORTRAN: MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX,MPI_CHARACTER, MPI_LOGICAL, etc…

C : MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE, etc…

Predefined Communicator: MPI_COMM_WORLD

# Collective communication: Broadcast

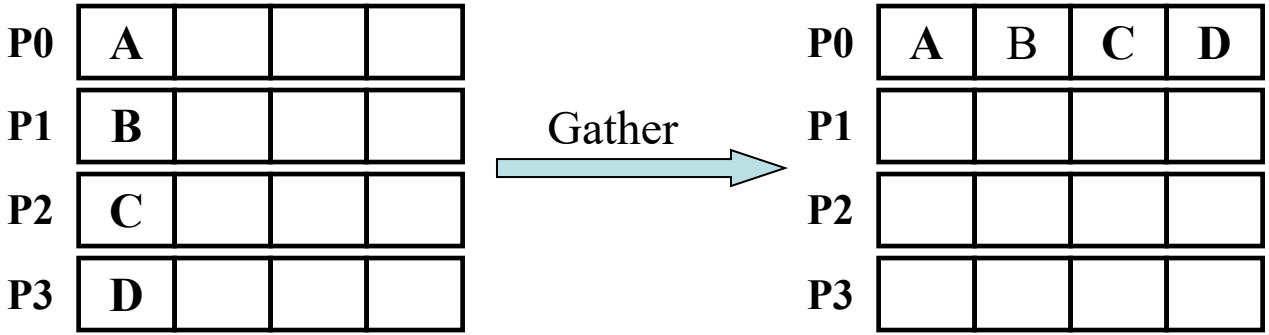
```
MPI_Bcast(buffer,count,datatype,root,comm,ierr)
```




- One process (called “root”) sends data to all the other processes in the same communicator
- Must be called by ALL processes with the same arguments

# Collective communication: Gather

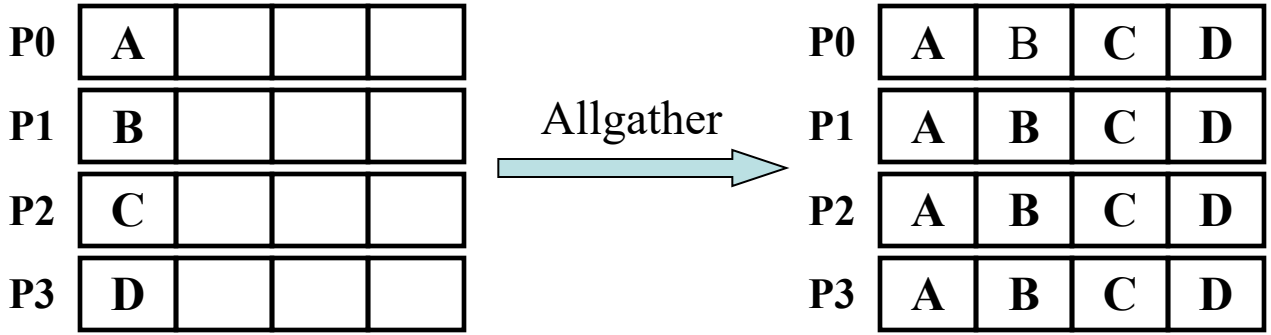
```
MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
           recvtype,root,comm,ierr)
```




- One root process collects data from all the other processes in the same communicator
- Must be called by all the processes in the communicator with the same arguments
- "sendcount" is the number of basic datatypes sent, not received (example above would be sendcount = 1)
- Make sure that you have enough space in your receiving buffer!

# Collective communication: Gather to All

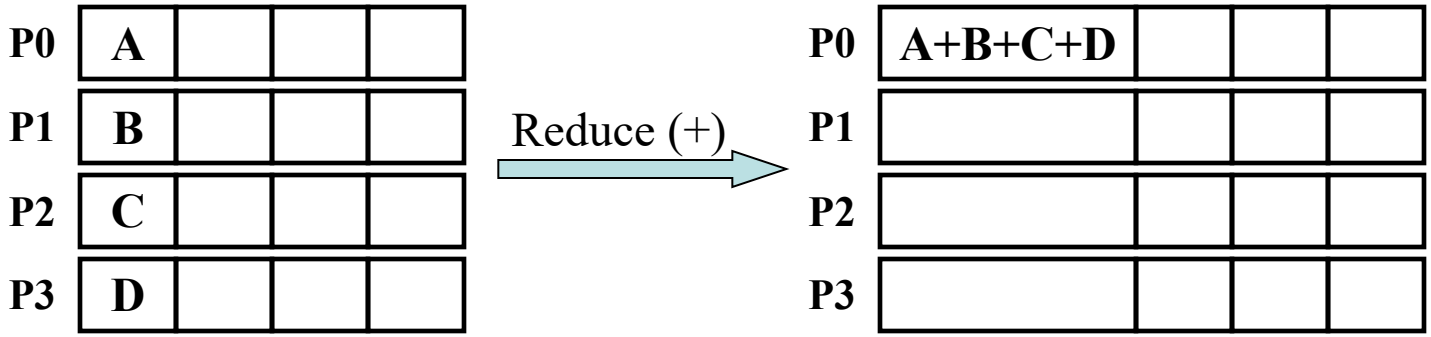
```
MPI_Allgather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
              recvtype,comm,info)
```




- All processes within a communicator collect data from each other and end up with the same information
- Must be called by all the processes in the communicator with the same arguments
- Again, sendcount is the number of elements sent

# Collective communication: Reduction

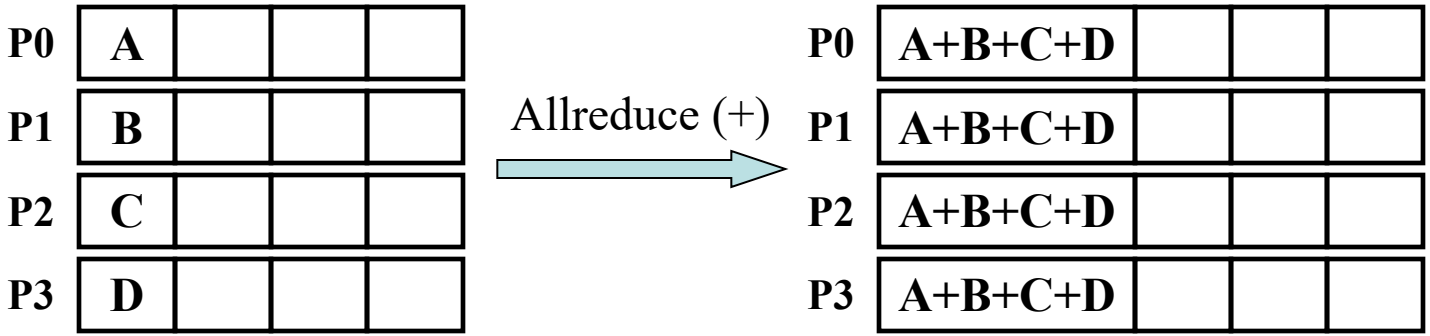
```
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```




- One root process collects data from all the other processes in the same communicator and performs an operation on the received data
- Called by all the processes with the same arguments
- Operations are: MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
- User can define own operation with MPI_Op_create()

# Collective communication: Reduction to All

```
MPI_Allreduce(sendbuf,recvbuf,count,datatype,op,comm,ierr)
```




- All processes within a communicator collect data from all the other processes and performs an operation on the received data
- Called by all the processes with the same arguments
- Operations are the same as for MPI_Reduce

# More MPI collective calls

**One "root" process send a different piece of the data to each one of the other Processes (inverse of gather)**

```
MPI_Scatter(sendbuf,sendcnt,sendtype,recvbuf,recvcnt,
            recvtype,root,comm,ierr)
```

**Each process performs a scatter operation, sending a distinct message to all the processes in the group in order by index.**

```
MPI_Alltoall(sendbuf,sendcount,sendtype,recvbuf,recvcnt,
             recvtype,comm,ierr)
```

**Synchronization: When necessary, all the processes within a communicator can be forced to wait for each other although this operation can be expensive**

```
MPI_Barrier(comm,ierr)
```

# Example: calculating π using numerical integration

```
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264643;
    double mypi, pi, h, sum, x;
    FILE *ifp;

    ifp = fopen("ex4.in","r");
    fscanf(ifp,"%d",&n);
    fclose(ifp);
    printf("number of intervals = %d\n",n);

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = 1; i <= n; i++) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;

    pi = mypi;
    printf("pi is approximately %.16f, Error is %.16f\n",
           pi, fabs(pi - PI25DT));
    return 0;
}
```

C version

# Root reads input and broadcast to all

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i, j, tag, my_n;
    double PI25DT = 3.14159265358979323846264338;
    double mypi,pi,h,sum,x,pi_frac,tt0,tt1,ttf;
    FILE *ifp;
    MPI_Status  Stat;
    MPI_Request request;

    n = 1;
    tag = 1;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    tt0 = MPI_Wtime();
    if (myid == 0) {
       ifp = fopen("ex4.in","r");
       fscanf(ifp,"%d",&n);
       fclose(ifp);
    }
 /* Global communication. Process 0 "broadcasts" n to all other processes */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

# Each process calculates its section of the integral and adds up results with MPI_Reduce

```
…
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid*n/numprocs+1; i <= (myid+1)*n/numprocs; i++) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;

    pi = 0.;  /* It is not necessary to set pi = 0 */

 /* Global reduction. All processes send their value of mypi to process 0
    and process 0 adds them up (MPI_SUM) */
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    ttf = MPI_Wtime();
    printf("myid=%d  pi is approximately %.16f, Error is %.16f  time = %10f\n",
               myid, pi, fabs(pi - PI25DT), (ttf-tt0));

    MPI_Finalize();
    return 0;
}
```

# Works with Python too!

- http://mpi4py.scipy.org/docs/usrman/tutorial.html
- mpirun -np 4 python script.py

**Script.py**

```
from mpi4py import MPI

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

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

- Uses “pickle” module to get access to C-type contiguous memory buffer
- Evolving rapidly
- On adroit.princeton.edu:
  - **module load openmpi/gcc**
  - **module load conda3**
  - **pip install --user mpi4py**

```
from mpi4py import MPI
import numpy
import time

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

N = numpy.arange(1, dtype=numpy.intc)
if rank == 0:
   N[0] = 1000*1000*100
comm.Bcast([N, 1, MPI.INT], root=0)

start = time.time()

h = 1.0 / N[0]; s = 0.0
for i in range(rank, N[0], size):
    x = h * (i + 0.5)
    s += 4.0 / (1.0 + x**2)
PI = numpy.array(s * h, dtype='d')
PI_sum = numpy.array(0.0, dtype='d')
#comm.Reduce([PI, MPI.DOUBLE], PI_sum, op=MPI.SUM, root=0)
comm.Allreduce([PI, MPI.DOUBLE], PI_sum, op=MPI.SUM)

end = time.time()
print("rank:%d  Pi with %d steps is %15.14f in %f secs" %(rank, N[0], PI_sum, end-start))
```

our PI calculation example

# What about those GPUs on SUMMIT?

- There are several ways to program for the GPUs
  - CUDA (since 2007): NVIDIA-specific programming language built as an extension of standard C language. Best approach to get the most out of your NVIDIA GPU. CUDA kernel not portable though so it won't work on the Intel Xeon Phi. Also available for FORTRAN but only for the PGI compiler.
  - **OpenMP 4.0!! ("target" directive introduced in 2013)**
  - **OpenACC** (First release in 2011) :Compiler directives similar to OpenMP. Portable code. Easy to get started. Available for a few compilers. More mature than OpenMP for GPU although OpenMP is catching up…
  - Libraries, commercial software, domain-specific environments, . . .
  - OpenCL: open standard, platform- and vendor independent
    - Works on both GPU AND CPU!!
    - Even harder than CUDA though…

# OpenACC example

```
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
   err=0.0;
#pragma acc parallel loop reduction(max:err)
   for( int j = 1; j < n-1; j++) {
      for(int i = 1; i < m-1; i++) {
         Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                              A[j-1][i] + A[j+1][i]);
         err = max(err, abs(Anew[j][i] - A[j][i]);
      }
   }
#pragma acc parallel loop
   for( int j = 1; j < n-1; j++) {
      for( int i = 1; i < m-1; i++ ) {
         A[j][i] = Anew[j][i];
      }
   }
   iter++;
}
```

# MPI References

- Just google “mpi”, or “mpi standard”, or “mpi tutorial”…
- http://www.mpi-forum.org (location of the MPI standard)
- http://www.llnl.gov/computing/tutorials/mpi/
- http://www.nersc.gov/nusers/help/tutorials/mpi/intro/
- http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html
- http://www-unix.mcs.anl.gov/mpi/tutorial/

- MPI on Linux clusters:
  - MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/)
  - Open MPI (http://www.open-mpi.org/)
- Books:
  - Using MPI “Portable Parallel Programming with the Message-Passing Interface” by William Gropp, Ewing Lusk, and Anthony Skjellum
  - Using MPI-2 “Advanced Features of the Message-Passing Interface”

# OpenMP References

- See:
  - http://www.openmp.org/resources/openmp-presentations/
  - http://www.openmp.org/resources/tutorials-articles/
- Excellent tutorial from SC'08 conference posted at:
  - http://www.openmp.org/mp-documents/omp-hands-on-SC08.pdf
  - See references within document
- More tutorials:
  - http://static.msi.umn.edu/tutorial/scicomp/general/openMP/index.html
  - https://computing.llnl.gov/tutorials/openMP/

# GPU References

- http://www.gputechconf.com/gtcnew/on-demand-gtc.php
- http://www.nvidia.com
- http://gpgpu.org
  - In particular: http://gpgpu.org/ppam2011
- http://www.olcf.ornl.gov/event/cray-technical-workshop-on-xk6-programming/
- http://www.pgroup.com/resources/index.htm
- http://www.caps-entreprise.com/products/openacc-compiler/