Introduction to mixed parallel programming using MPI and OpenMP *(and OpenACC…)*

PICSciE Mini-Course

December 5, 2013

Stéphane Ethier

(ethier@pppl.gov)

Computational Plasma Physics Group

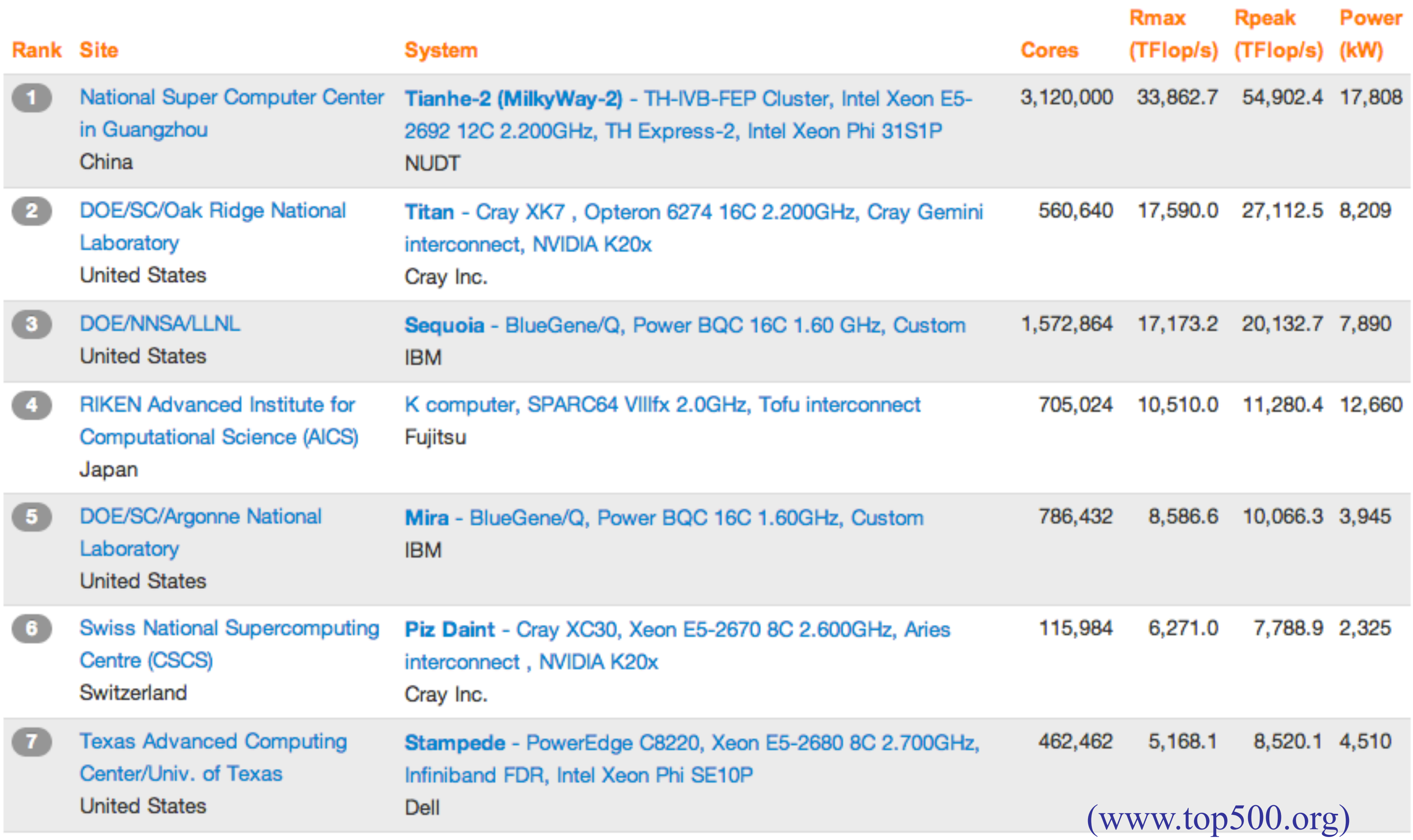
Princeton Plasma Physics Lab

Goals for this tutorial

1. Get introduced to parallel programming for shared memory and distributed memory systems
2. Learn practical knowledge of MPI communication library
3. Learn practical knowledge of OpenMP directives for shared memory parallelism
4. Learn basic knowledge of OpenACC accelerator directives
5. Learn how to use all of the above in the same parallel application

Why mixed programming?

November 2013 top500 list



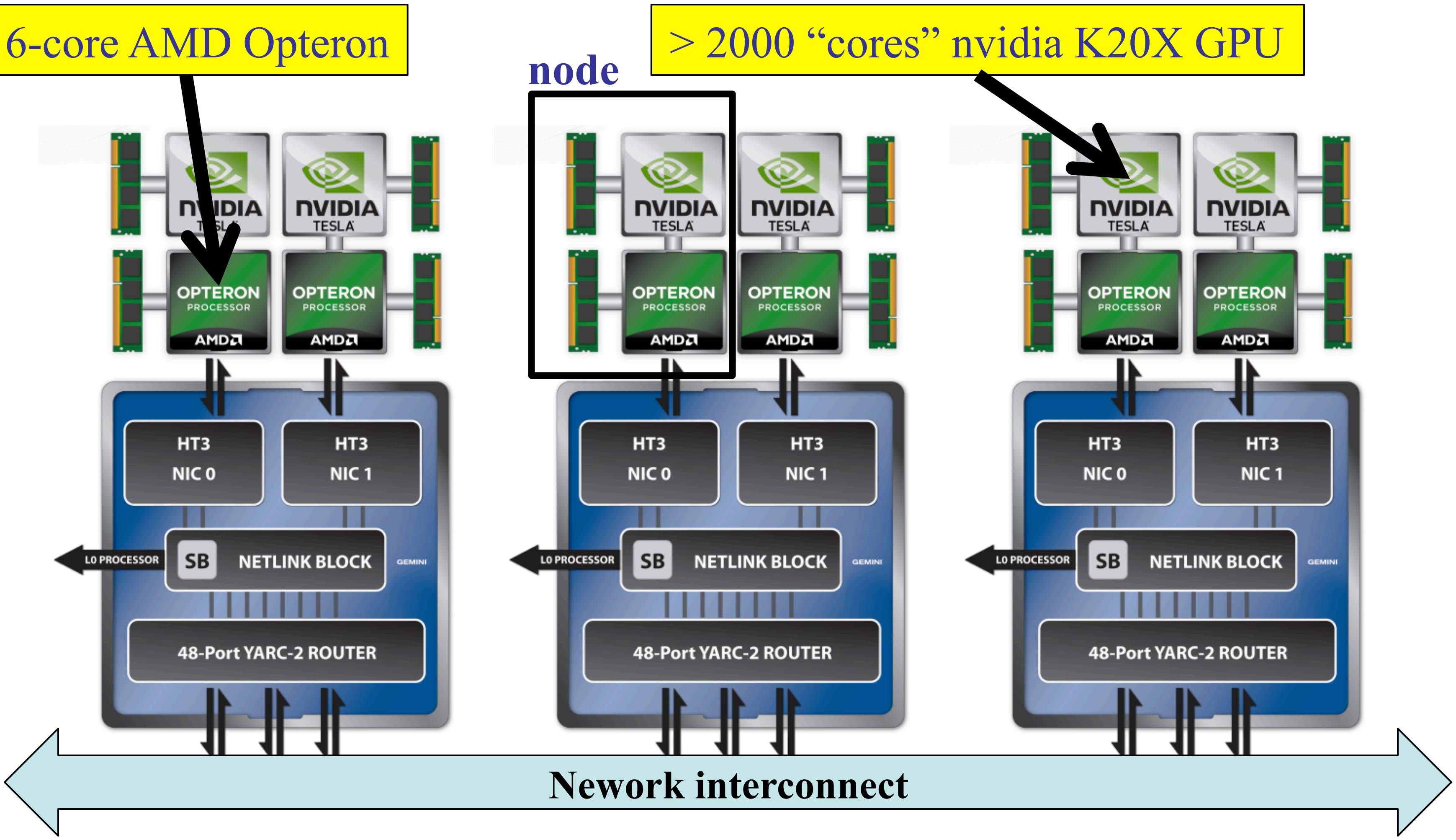
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510

(www.top500.org)

Titan Cray XK7 hybrid system

Processor: AMD Interlagos (16)	**GPUs: 18,688 Tesla K20**
Cabinets: 200	**Memory/node CPU: 32 GB**
# nodes: 18,688	**Memory/node GPU: 6 GB**
# cores/node: 16	**Interconnect: Gemini**
Total cores: 299,008	**Speed: 27 PF peak (17.6)**

Cray XK7 architecture



Why Parallel Computing?

Why not run *n* instances of my code *à la* MapReduce/Hadoop?

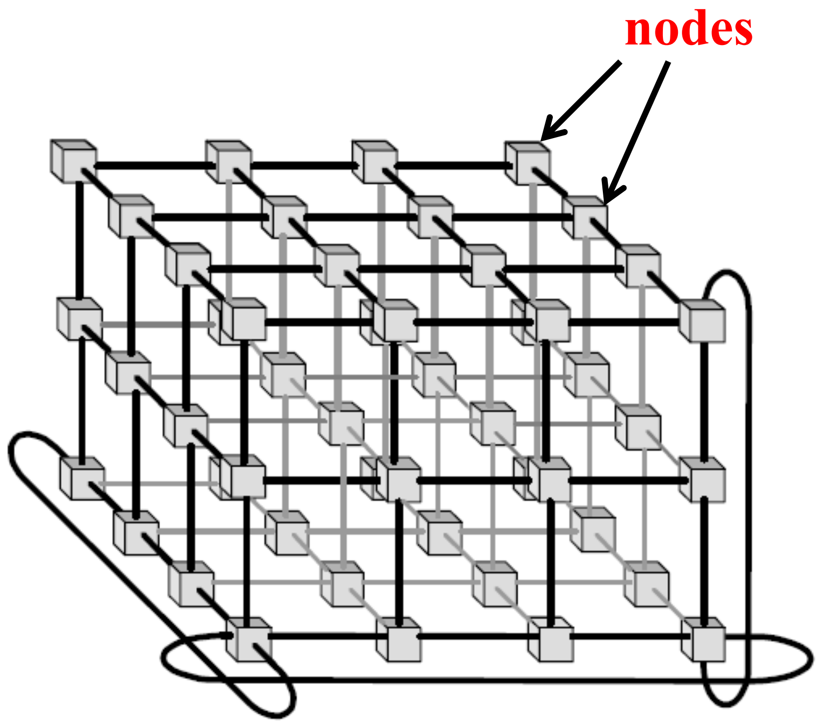
- Want to speed up your calculation.
- Your problem is too large for a single node
- Want to use those extra cores on your multicore processor
- Solution:
  - Split the work between several processor cores so that they can work in parallel
  - Exchange data between them when needed
- How?
  - Compiler auto-prallelization (only good for obvious parallelism)
  - OpenMP directives on shared memory node
  - Message Passing Interface (MPI) on distributed memory systems (works also on shared memory nodes)
  - and others…

Languages and libraries for parallel computing

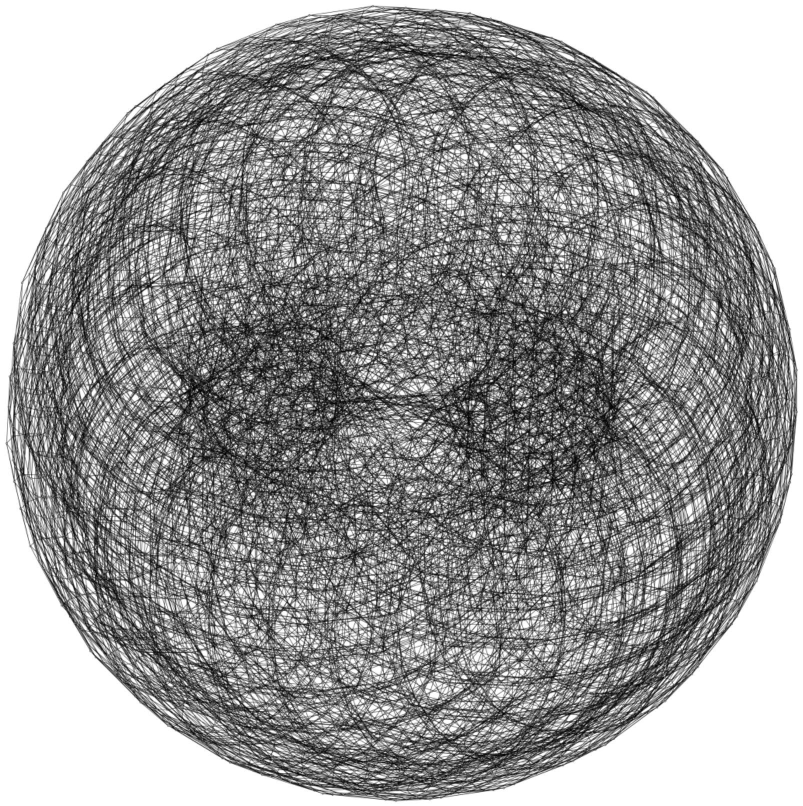
- Multithreading or “shared memory parallelism”
  - Directive-base OpenMP (deceptively easy) www.openmp.org (!$OMP DO)
  - POSIX pthread programming (explicit parallelism, somewhat harder than MPI since one needs to manage threads access to memory).
  - GPGPU (General-Purpose Graphical Processing Unit) programming with **CUDA** (nvidia) or OpenCL (similar to CUDA but more difficult) or **OpenACC!**
- PGAS global address space SPMD languages (using GASNet layer or other)
  - Efficient single-sided communication on globally-addressable memory
  - FORTRAN 2008 co-arrays
    - Example: xarray(100,200)[*] where * is a process number
    - “puts” and “gets” directly to and from remote memory via the network with little or no involvement from the CPU
    - Works best on a specialized network, such as Cray XE6 Gemini interconnect
  - UPC (http://upc.lbl.gov/): Similar to co-array Fortran but for C.
- MPI for distributed-memory parallelism (runs everywhere except GPUs)

Let’s start with MPI…

Reason to use MPI: Scalability and portability



3D torus network interconnect
(e.g. Cray XE6 or XK7)



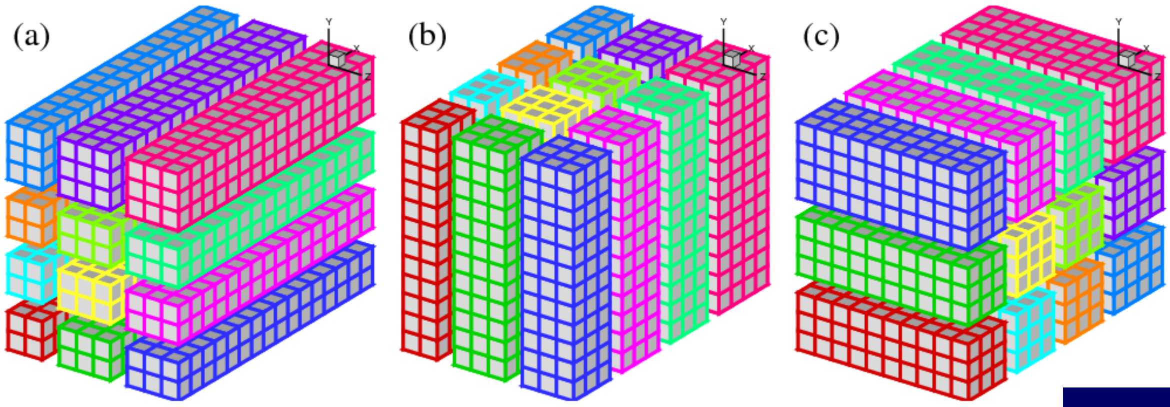
3D torus interconnect
On a large system!

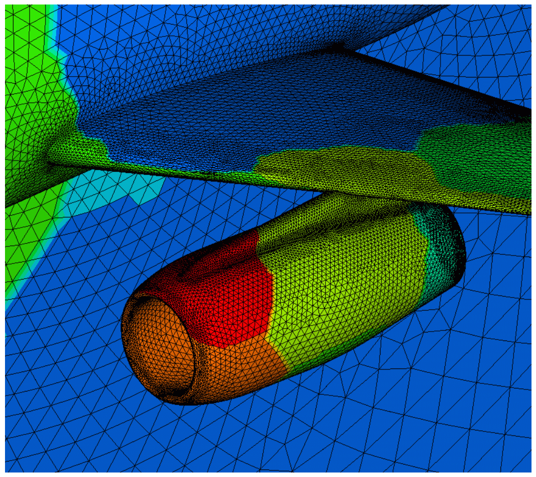
MPI

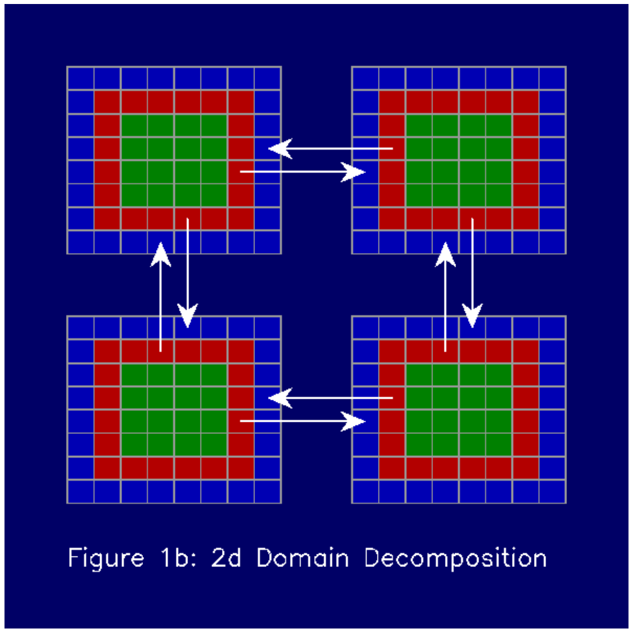
- Context: distributed memory parallel computers
  - Each processor has its own memory and cannot access the memory of other processors
  - A copy of the same executable runs on each MPI process (processor core)
  - Any data to be shared must be explicitly transmitted from one to another
- Most message passing programs use the *single program multiple data* (SPMD) model
  - Each processor executes the same set of instructions
  - Parallelization is achieved by letting each processor operate on a different piece of data
  - Not to be confused with SIMD: Single Instruction Multiple Data *a.k.a vector computing*

How to split the work between processors?
Domain Decomposition

- Most widely used method for grid-based calculations

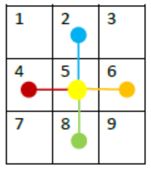




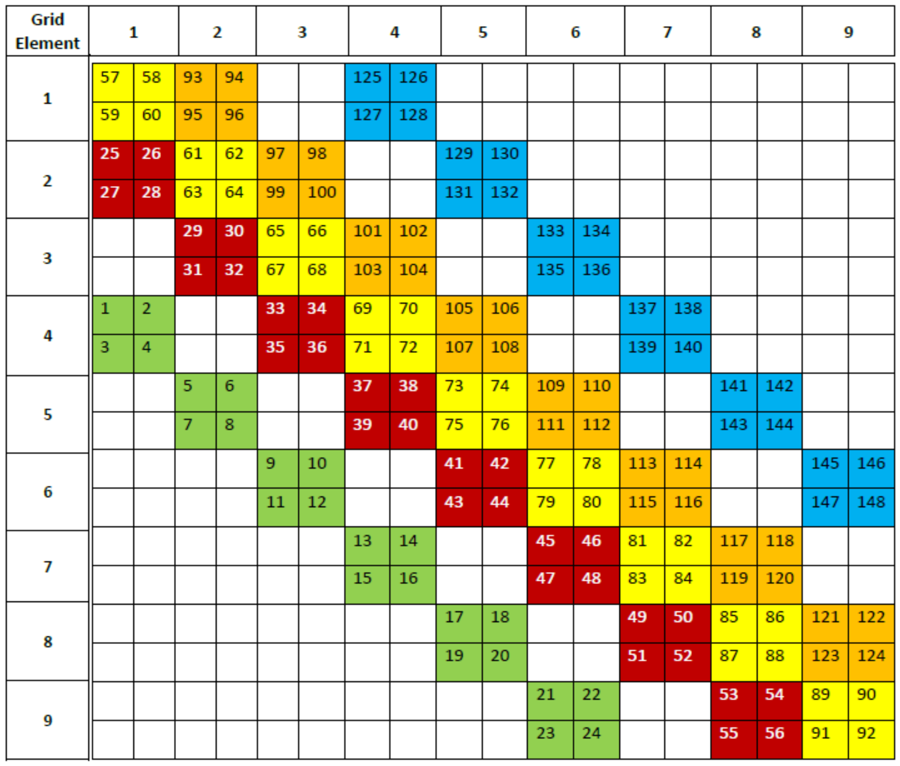


How to split the work between processors?
Split matrix elements in PDE solves

- See PETSc project:

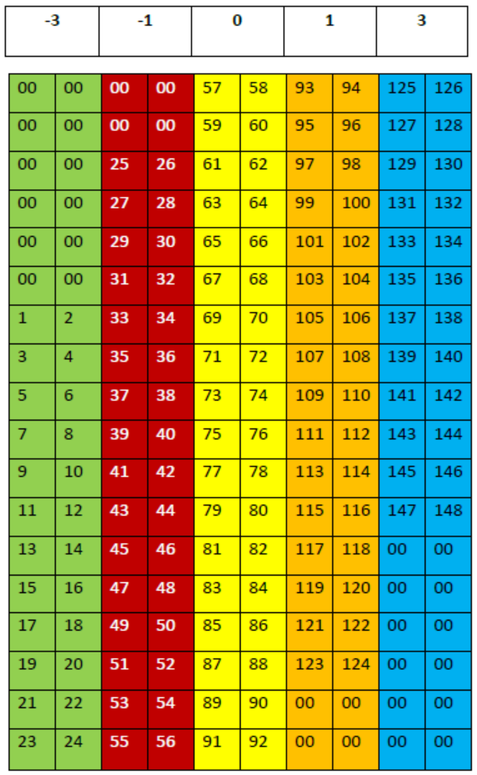


(a)



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

(b)

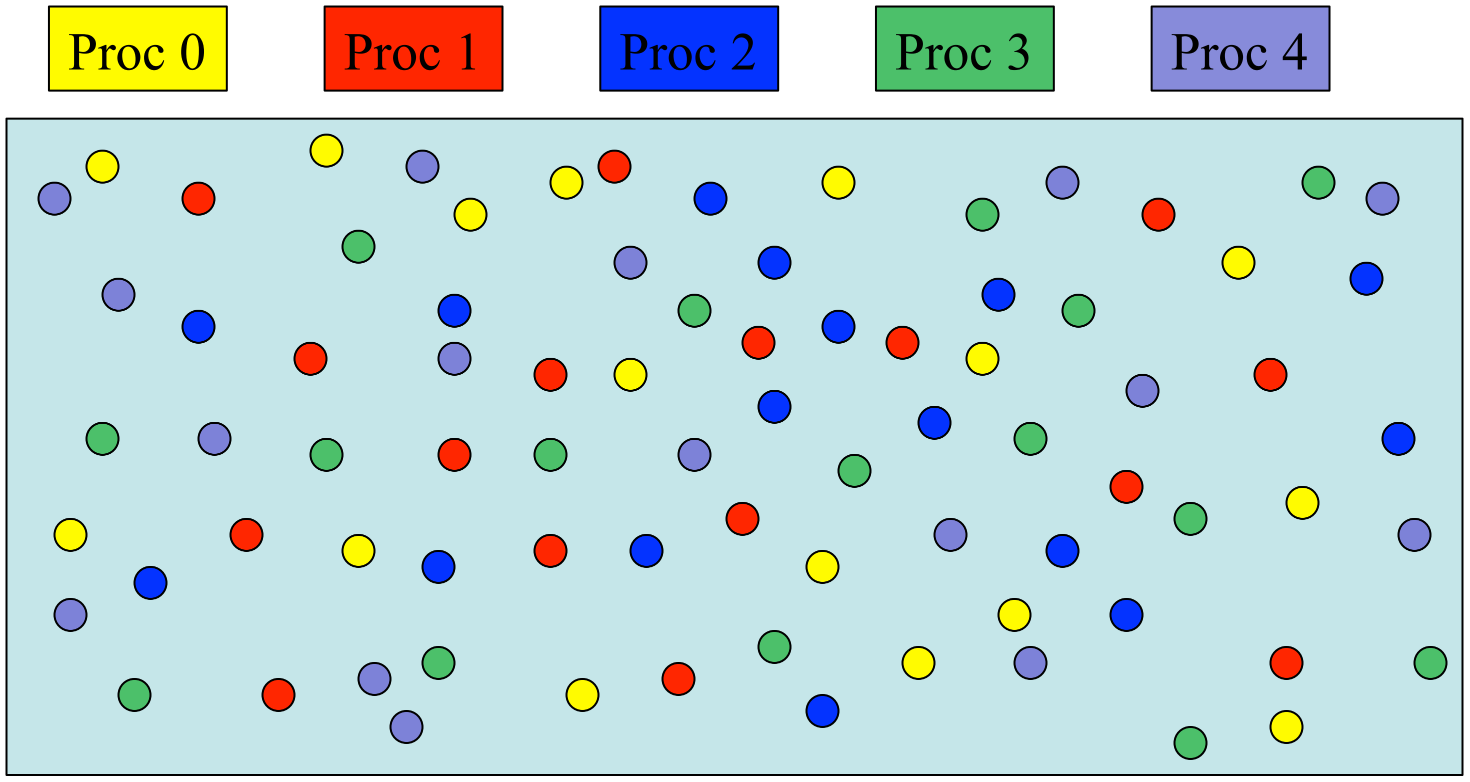


-3		-1		0		1		3	
00	00	00	00	57	58	93	94	125	126
00	00	00	00	59	60	95	96	127	128
00	00	25	26	61	62	97	98	129	130
00	00	27	28	63	64	99	100	131	132
00	00	29	30	65	66	101	102	133	134
00	00	31	32	67	68	103	104	135	136
1	2	33	34	69	70	105	106	137	138
3	4	35	36	71	72	107	108	139	140
5	6	37	38	73	74	109	110	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



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.

MPI standard

- MPI standard is a specification of what MPI is and how it should behave. Vendors have some flexibility in the implementation (e.g. buffering, collectives, topology optimizations, etc.).
- This tutorial focuses on the functionality introduced in the original MPI-1 standard
- MPI-2 standard introduced additional support for
  - Parallel I/O (many processes writing to a single file). Requires a parallel filesystem to be efficient
  - One-sided communication: MPI_Put, MPI_Get
  - Dynamic Process Management
- New MPI-3 standard starting to be implemented by compilers vendors
  - Non-blocking collectives
  - Improved one-sided communications
  - Improved Fortran bindings for type check
  - And more (see http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf)

How much do I need to know?

- MPI has over 125 functions/subroutines
- Can actually do everything with about 6 of them although I would not recommend it
- Collective functions are EXTREMELY useful since they simplify the coding and vendors optimize them for their interconnect hardware
- One can access flexibility when it is required.
- One need not master all parts of MPI to use it.

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 Fortran 90

```
Program mpi_code
  ! Load MPI definitions
    use mpi (or include mpif.h)

  ! Initialize MPI
    call MPI_Init(ierr)
  ! Get the number of processes
    call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
  ! Get my process number (rank)
    call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

    Do work and make message passing calls…

  ! Finalize
    call MPI_Finalize(ierr)

end program mpi_code
```

Header file

```
Program mpi_code
  ! Load MPI definitions
    use mpi

  ! Initialize MPI
    call MPI_Init(ierr)
  ! Get the number of processes
    call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
  ! Get my process number (rank)
    call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

    Do work and make message passing calls…

  ! Finalize
    call MPI_Finalize(ierr)

end program mpi_code
```

- Defines MPI-related parameters and functions
- Must be included in all routines calling MPI functions
- Can also use include file:

  include mpif.h

Initialization

```
Program mpi_code
  ! Load MPI definitions
    use mpi

  ! Initialize MPI
    call MPI_Init(ierr)
  ! Get the number of processes
    call MPI_Comm_size(MPI_COMM_WO
  ! Get my process number (rank)
    call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

    Do work and make message passing calls…

  ! Finalize
    call MPI_Finalize(ierr)

end program mpi_code
```

- Must be called at the beginning of the code before any other calls to MPI functions
- Sets up the communication channels between the processes and gives each one a rank.

How many processes do we have?

- Returns the number of processes available under MPI_COMM_WORLD communicator
- This is the number used on the mpiexec (or mpirun) command:

  mpiexec –n nproc a.out

```
    call MPI_Init(ierr)
  ! Get the number of processes
    call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
  ! Get my process number (rank)
    call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

    Do work and make message passing calls…

  ! Finalize
    call MPI_Finalize(ierr)

end program mpi_code
```

What is my rank?

```
Program mpi_code
   ! Load MPI definitions
```

- Get my rank among all of the nproc processes under MPI_COMM_WORLD
- This is a unique number that can be used to distinguish this process from the others

```
     call MPI_Comm_si  e(MPI_COMM_WORLD,nproc,ierr)
   ! Get my process n  ber (rank)
     call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

     Do work and make message passing calls…

   ! Finalize
     call MPI_Finalize(ierr)

end program mpi_code
```

Termination

```
Program mpi_code
  ! Load MPI definitions
    use mpi  (or include mpif.h)

  ! Initialize MPI
    call MPI_Init(ierr)
  ! Get the number of processes
    call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
  ! Get my process number (rank)
    call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

    Do work and make message passing calls…

  ! Finalize
    call MPI_Finalize(ierr)

end program mpi_code
```

- Must be called at the end of the properly close all communication channels
- No more MPI calls after finalize

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;
}
```

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)

Makefile

- Always a good idea to have a Makefile

```
%cat Makefile
CC=mpicc
CFLAGS=-O

% : %.c
	$(CC) $(CFLAGS) $< -o $@
```

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
aprun -size #proc a.out (Cray XT)
```

$\left.\begin{array}{l}\texttt{mpirun -np \#proc a.out}\\ \texttt{mpiexec -n \#proc a.out}\end{array}\right\}$ MPICH, OPENMPI

- 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

mpirun and mpiexec

- Both are used for starting an MPI job
- If you don't have a batch system, use mpirun

```
 mpirun -np #proc -machinefile mfile a.out >& out < in &

%cat mfile
 machine1.princeton.edu              machine1.princeton.edu
 machine2.princeton.edu     OR       machine1.princeton.edu
 machine3.princeton.edu              machine1.princeton.edu
 machine4.princeton.edu              machine1.princeton.edu
```

1 MPI process per host

4 MPI processes on same host

- PBS batch system usually takes care of arguments to mpiexec

Batch System

- Submit a job script: qsub script
- Check status of jobs: qstat –a   (for all jobs)
- Stop a job: qdel job_id

```
###  --- PBS SCRIPT  ---
#PBS -l nodes=4:ppn=2,walltime=02:00:00
#PBS -q dque
#PBS -V
#PBS -N job_name
#PBS -m abe
cd $PBS_O_WORKDIR
mpiexec a.out
```

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

<u>Point to point:</u> **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:

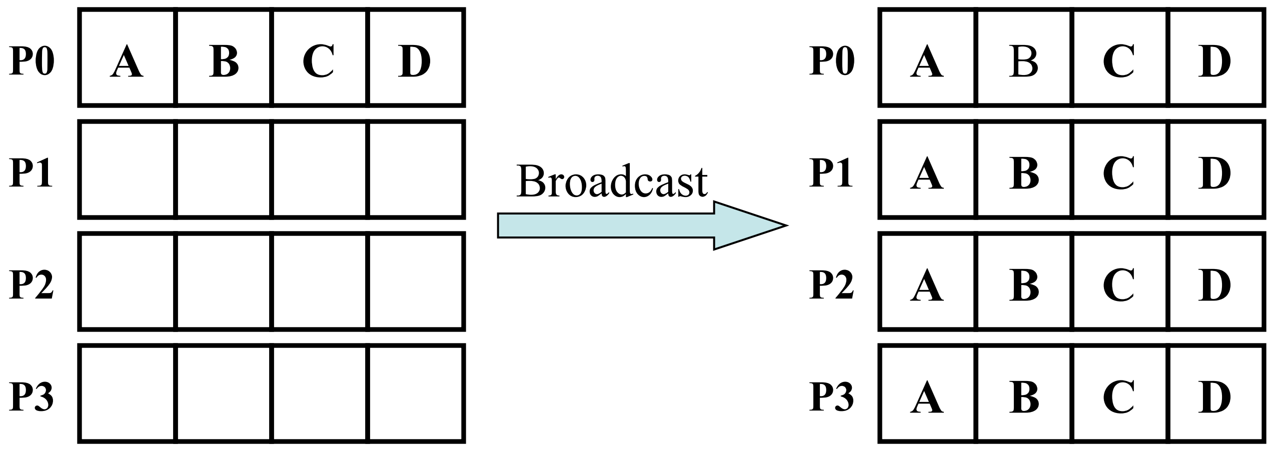
FORTRAN: `MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX,MPI_CHARACTER, MPI_LOGICAL, etc…`

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

Predefined Communicator: `MPI_COMM_WORLD`

Collective communication: Broadcast

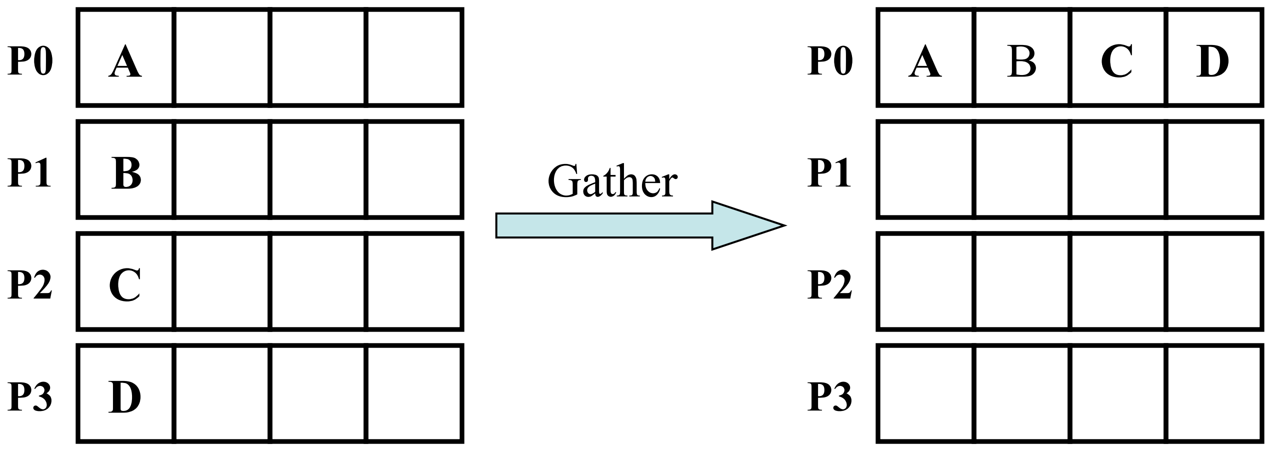
```
MPI_Bcast(buffer,count,datatype,root,comm,ierr)
```



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

Collective communication: Gather

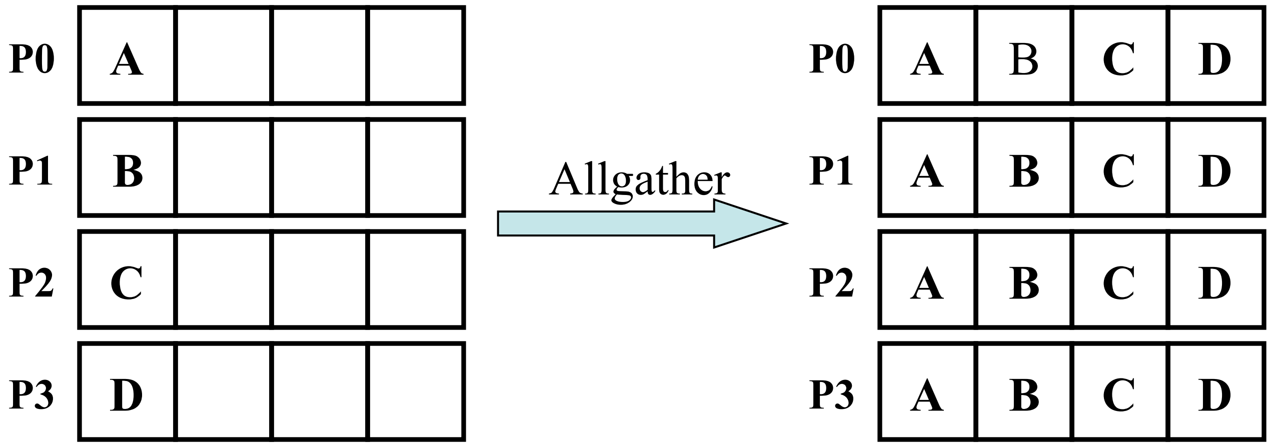
```
MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
                           recvtype,root,comm,ierr)
```



- 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

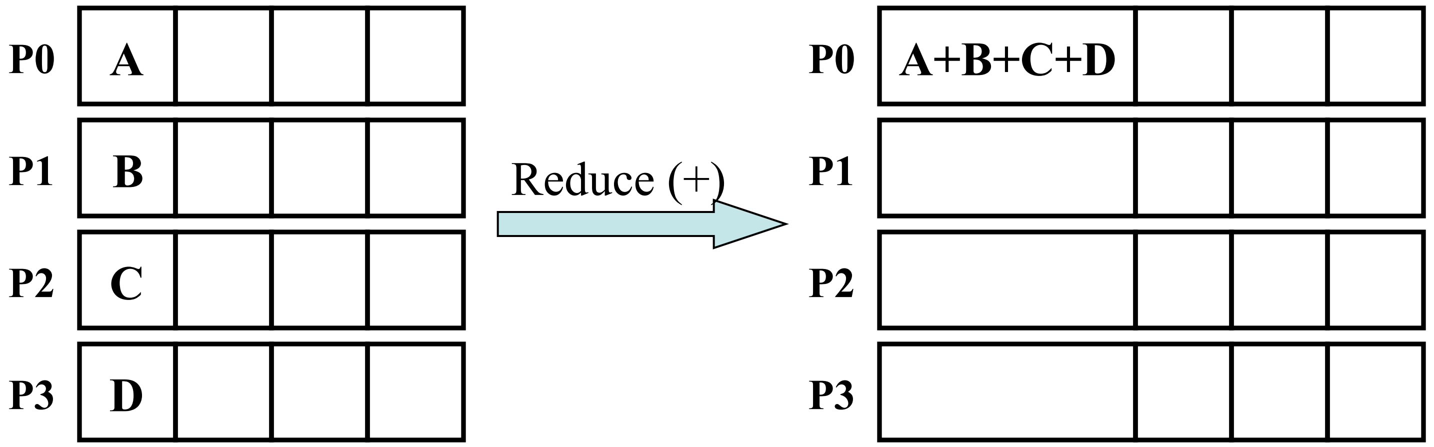
```
MPI_Allgather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
              recvtype,comm,info)
```



- 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

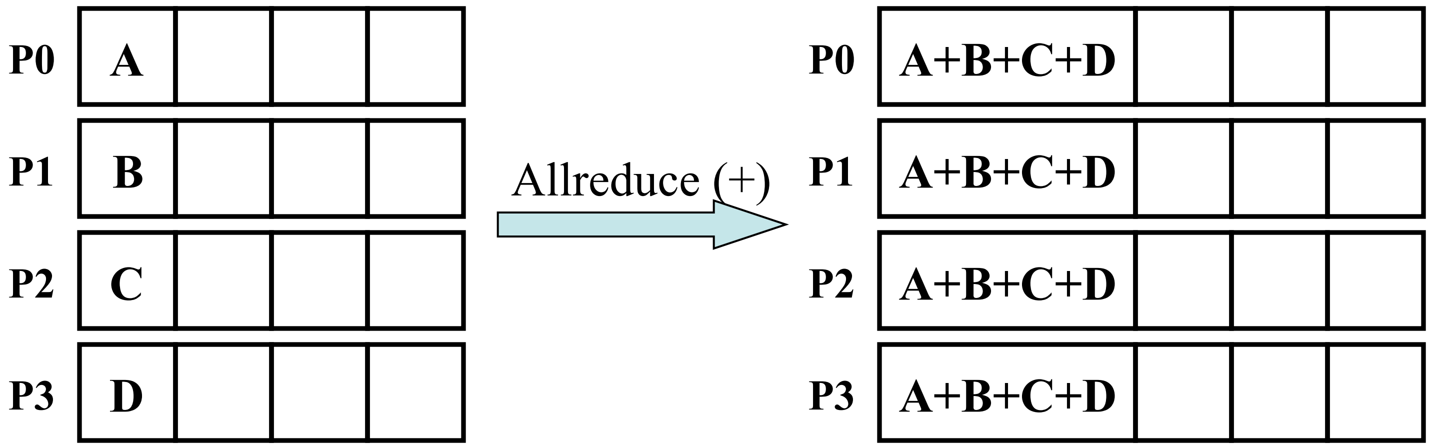
```
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```



- 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)
```

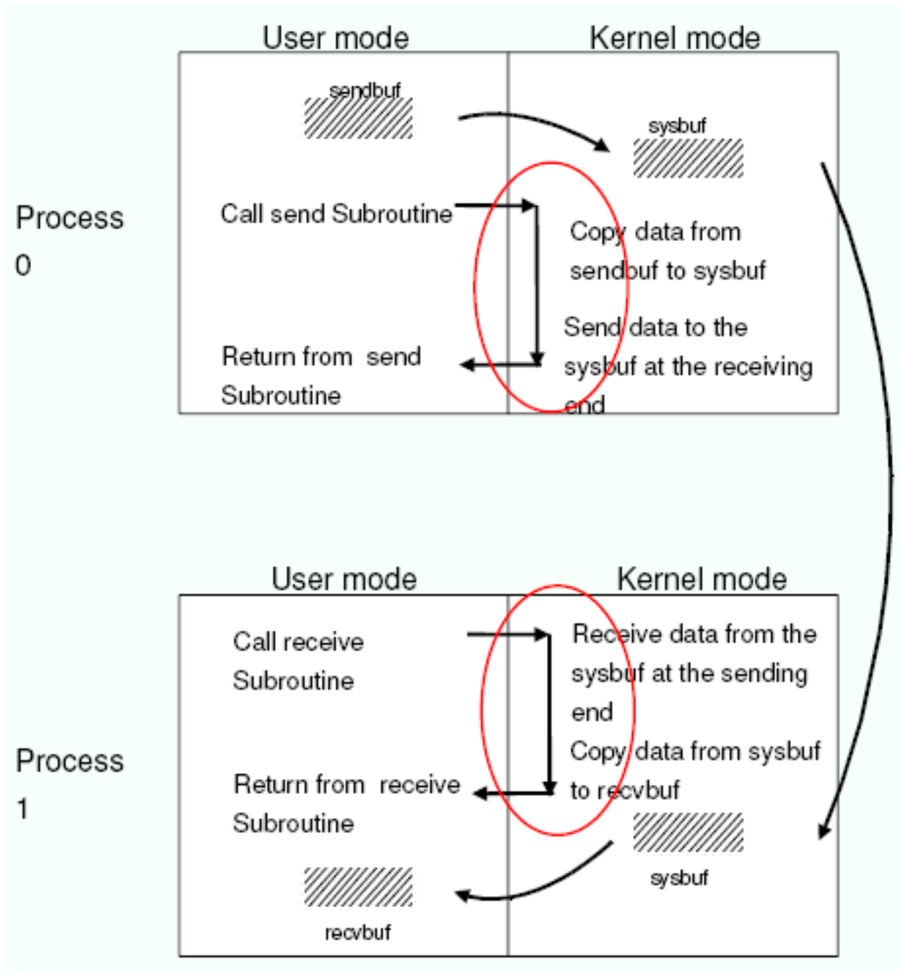
Each process performs a scatter operation, sending a distinct message to all the processes in the group in order by index.

```
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

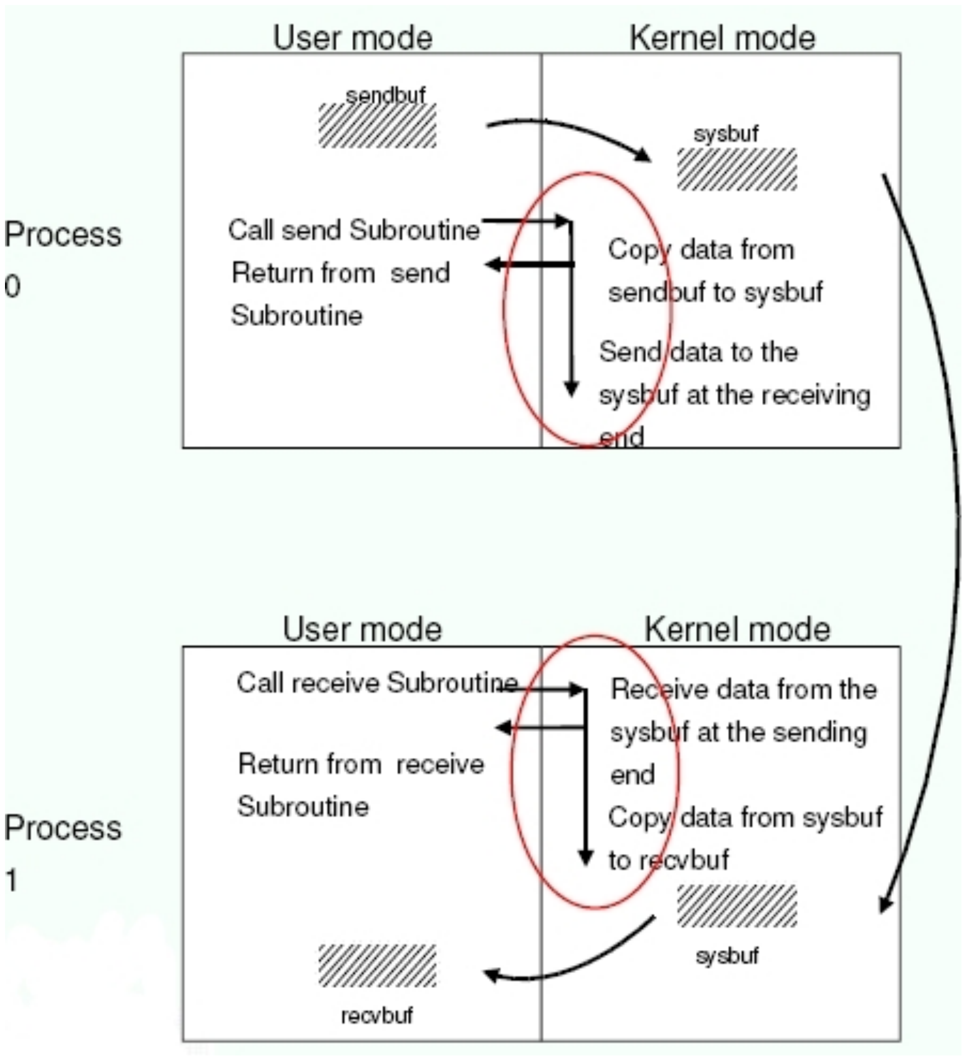
```
MPI_Barrier(comm,ierr)
```

Blocking communications


User mode
Kernel mode
sendbuf
sysbuf
Process 0
Call send Subroutine
Copy data from sendbuf to sysbuf
Send data to the sysbuf at the receiving end
Return from send Subroutine
User mode
Kernel mode
Call receive Subroutine
Receive data from the sysbuf at the sending end
Copy data from sysbuf to recvbuf
Process 1
Return from receive Subroutine
sysbuf
recvbuf

- The call waits until the data transfer is done
  - The sending process waits until all data are transferred to the system buffer
  - The receiving process waits until all data are transferred from the system buffer to the receive buffer
- All collective communications are blocking

Non-blocking


User mode
Kernel mode
sendbuf
sysbuf
Process 0
Call send Subroutine
Return from send Subroutine
Copy data from sendbuf to sysbuf
Send data to the sysbuf at the receiving end
User mode
Kernel mode
Call receive Subroutine
Return from receive Subroutine
Receive data from the sysbuf at the sending end
Copy data from sysbuf to recvbuf
Process 1
sysbuf
recvbuf

- Returns immediately after the data transferred is initiated
- Allows to overlap computation with communication
- Need to be careful though
  - When send and receive buffers are updated before the transfer is over, the result will be wrong

Non-blocking send and receive

Point to point:

```
MPI_Isend(buf,count,datatype,dest,tag,comm,request,ierr)

MPI_Irecv(buf,count,datatype,source,tag,comm,request,ierr)
```

The functions MPI_Wait and MPI_Test are used to complete a nonblocking communication

```
MPI_Wait(request,status,ierr)

MPI_Test(request,flag,status,ierr)
```

MPI_Wait returns when the operation identified by “request” is complete. This is a non-local operation.

MPI_Test returns “flag = true” if the operation identified by “request” is complete. Otherwise it returns “flag = false”. This is a local operation.

MPI-3 standard introduces “non-blocking collective calls”

How to time your MPI code

- Several possibilities but MPI provides an easy to use function called “MPI_Wtime()”. It returns the number of seconds since an arbitrary point of time in the past.

```
FORTRAN: double precision MPI_WTIME()
      C: double MPI_Wtime()
```

```
starttime=MPI_WTIME()
```

… *program body* …

```
endtime=MPI_WTIME()
elapsetime=endtime-starttime
```

Debugging tips

Use "unbuffered" writes to do "printf-debugging" and always write out the process id:

```
C:       fprintf(stderr,"%d: …",myid,…);
Fortran: write(0,*)myid,': …'
```

If the code detects an error and needs to terminate, use MPI_ABORT. The errorcode is returned to the calling environment so it can be any number.

```
C:       MPI_Abort(MPI_Comm comm, int errorcode);
Fortran: call MPI_ABORT(comm, errorcode, ierr)
```

To detect a "NaN" (not a number):

```
C:       if (isnan(var))
Fortran: if (var /= var)
```

Use a parallel debugger such as Totalview or DDT if available

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”

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 = 3.141592653589793238462643;
    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 = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, pi_frac;
    double 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

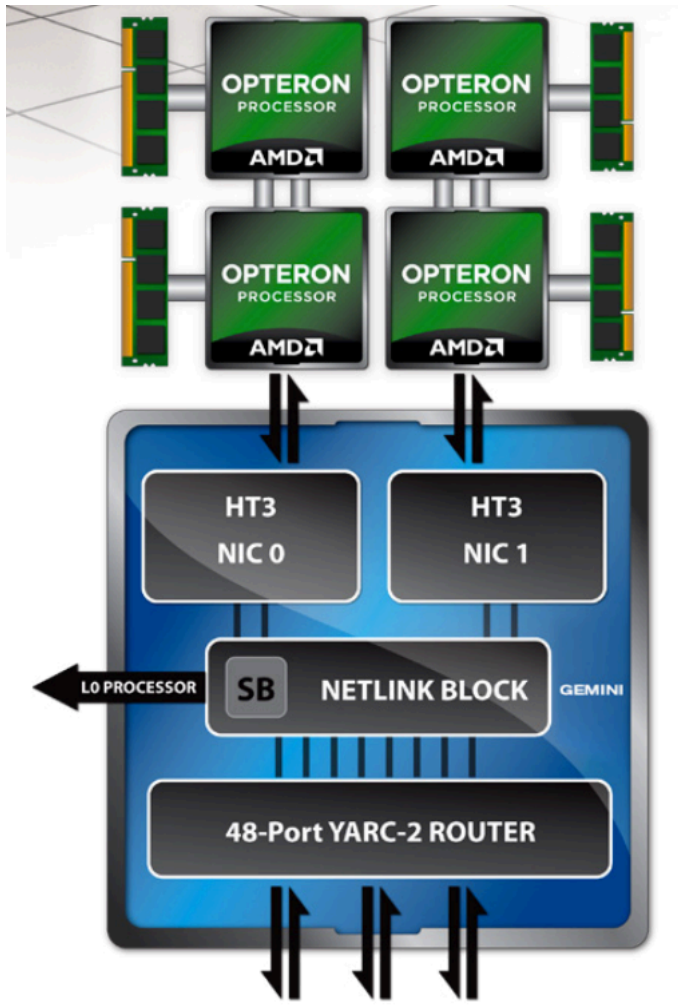
```
…
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid*n/numprocs+1; i <= (myid+1)*n/numprocs; i++) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;

    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;
}
```

OpenMP: a directive-based approach to shared memory parallelism

http://www.openmp.org

Why OpenMP?

Cray XE6 architecture



- Each node consists of 2 x 16-core (or 12-core) sockets
- The sockets are linked together and share the entire memory on the node
- All 32 cores share the memory and the single access to the network
- If each core is running an MPI process and the code issues an MPI collective call (MPI_Alltoall), all 32 processes will fight for access to the network at the same time!

What is OpenMP?

- OpenMP is:
  - An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism
  - Comprised of three primary API components:
    - Compiler Directives
    - Runtime Library Routines
    - Environment Variables
  - Portable:
    - The API is specified for C/C++ and Fortran
    - All operating systems that can handle multithreading can in principle run OpenMP codes (Linux, Unix, MacOS X, Windows)
  - Standardized:
    - Jointly defined and endorsed by a group of major computer hardware and software vendors
    - Expected to become an ANSI standard later?

What are directives?

- In C or C++, preprocessor statements ARE directives. They “direct” the preprocessing stage.
- Parallelization directives tell the compiler to add some machine code so that the next set of instructions will be distributed to several processors and run in parallel.
- In FORTRAN, directives are special purpose comments inserted right before the loop or region to parallelize.

C:

```
#pragma omp parallel for private(idx)
for (idx=1; idx <= n; idx++) {
   a[idx] = b[idx] + c[idx];
}
```

Fortran:

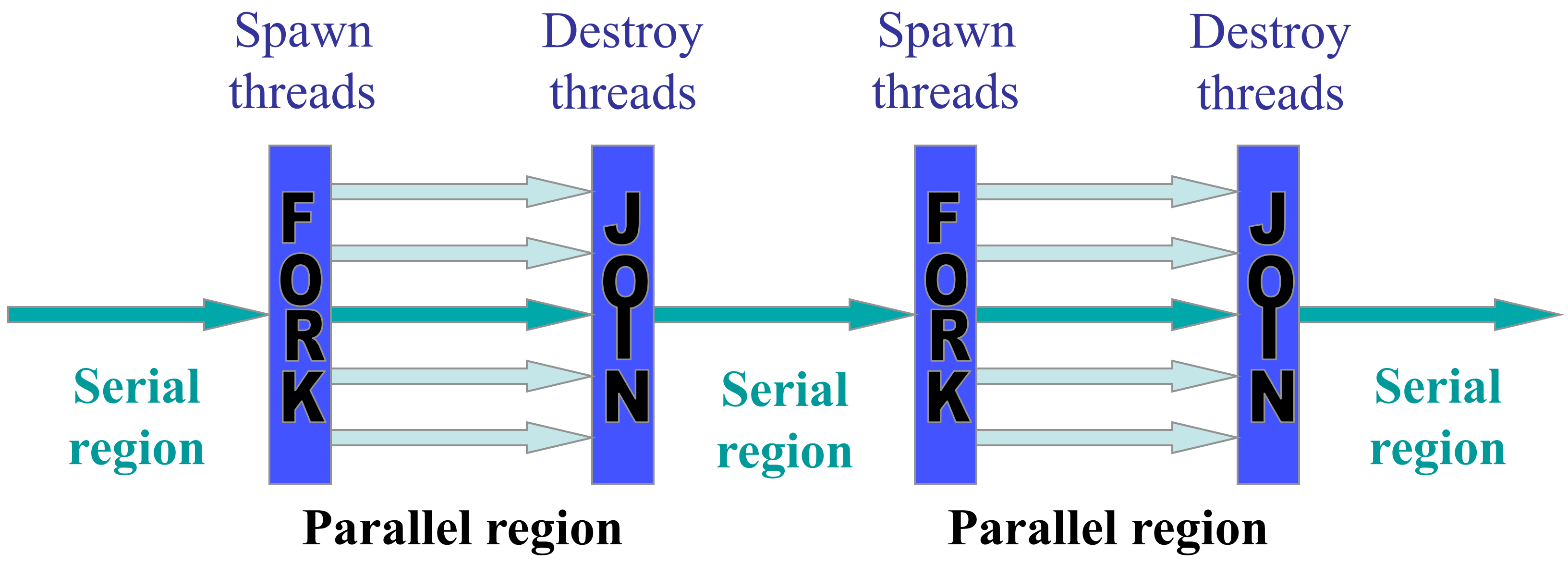
```
!$omp parallel do private(idx)
do idx=1,n
   a(idx) = b(idx) + c(idx)
enddo
```

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)

Shared memory parallelism

- 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 processors.

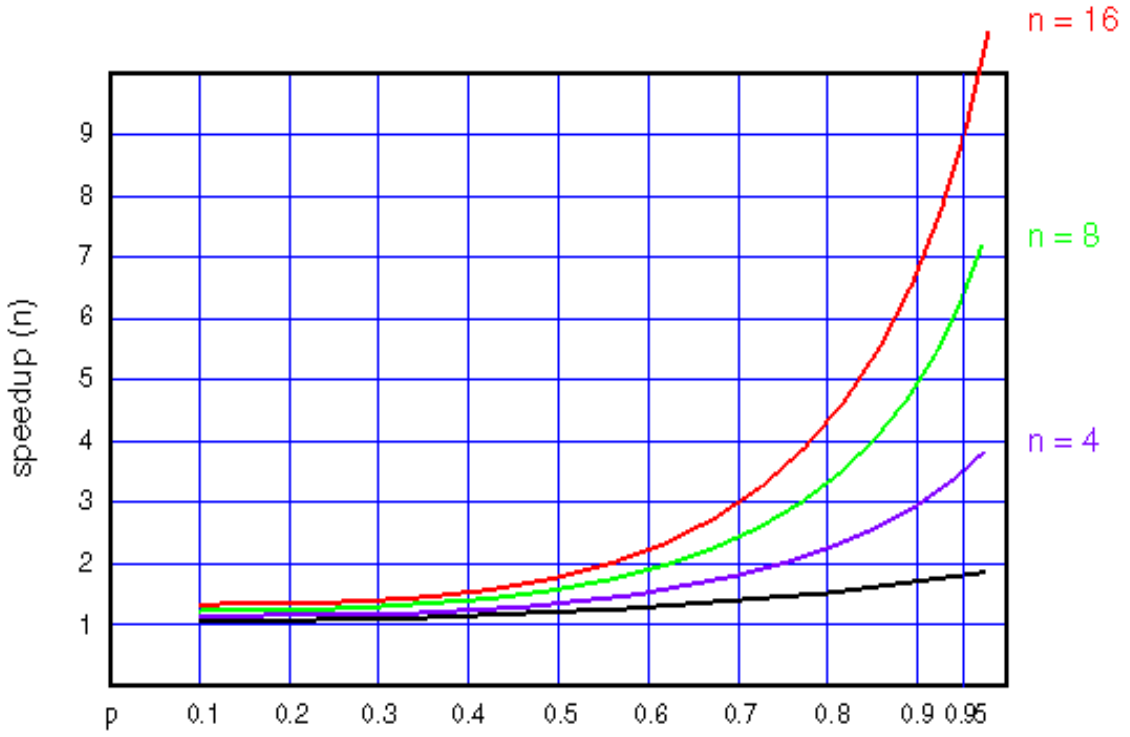
Amdahl's law of scalability

$$\text{speedup}(n) = \frac{t_1}{t_n} = \frac{1}{\left(\frac{p}{n} + (1-p)\right)}$$

where ***n*** is the number of processors and ***p*** the fraction of parallel work

$$\frac{t_1}{t_\infty} = \frac{1}{(1-p)}$$

- For *p*=0.8 the max speedup is 5!!
- The goal is to minimize the time spent in the serial regions
- **Profile your code!**



Goals of OpenMP

- Provide a standard among a variety of shared memory architectures/ platforms
- Establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
- Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
- Provide the capability to implement both coarse-grain and fine-grain parallelism.
  - Coarse-grain = domain decomposition.
  - Fine-grain = loop-level parallelism.
- Supports Fortran (77, 90, and 95), C, and C++
- Public forum for API and membership

Example of OpenMP code structure

In FORTRAN:

```
    PROGRAM HELLO
        INTEGER VAR1, VAR2, VAR3
        Serial code . . .
        Beginning of parallel section. Fork a team of threads.
       Specify variable scoping
!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
       Parallel section executed by all threads . . .
      All threads join master thread and disband
!$OMP END PARALLEL
       Resume serial code . . .
    END
```

Example of code structure in C

In C:

```
#include <omp.h>
main () {
  int var1, var2, var3;
 Serial code . . .
 Beginning of parallel section. Fork a team of threads.
 Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
 {
    Parallel section executed by all threads . . .
   All threads join master thread and disband
 }
 Resume serial code . . .
}
```

Directives format in Fortran

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)
```

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, COMP, *$OMP**
- Just use **!$OMP** and you will be fine…
- The sentinel must be followed by a valid directive name.
- Clauses are optional and can be in any order, and repeated as necessary unless otherwise restricted.
- All Fortran fixed form rules for line length, white space, continuation and comment columns apply for the entire directive line

Fortran fixed form source

- Fixed Form Source:
  - !$OMP C$OMP *$OMP are accepted sentinels and must start in column 1
  - All Fortran fixed form rules for line length, white space, continuation and comment columns apply for the entire directive line
  - Initial directive lines must have a space/zero in column 6.
  - Continuation lines must have a non-space/zero in column 6.

The following formats are equivalent:

```
!234567
!$OMP PARALLEL DO SHARED(A,B,C)
C$OMP PARALLEL DO C$OMP+SHARED(A,B,C)
```

Fortran free form source

- Free Form Source:
  - !$OMP is the only accepted sentinel. Can appear in any column, but must be preceded by white space only.
  - All Fortran free form rules for line length, white space, continuation and comment columns apply for the entire directive line
  - Initial directive lines must have a space after the sentinel.
  - Continuation lines must have an ampersand as the last non-blank character in a line. The following line must begin with a sentinel and then the continuation directives.

```
!23456789
    !$OMP PARALLEL DO &
          !$OMP SHARED(A,B,C)
  !$OMP PARALLEL &
    !$OMP&DO SHARED(A,B,C)
```

C / C++ Directives Format

```
#pragma omp parallel default(shared) private(beta,pi)
```

- #pragma omp
  - Required for all OpenMP C/C++ directives.
- **directive-name**
  - A valid OpenMP directive. Must appear after the pragma and before any clauses.
- **[clause, ...]**
  - This is optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.
- **newline**
  - Required. Precedes the structured block which is enclosed by this directive.

General rules for C/C++ format

- Directives follow conventions of the C/C++ standards for compiler directives
- Case sensitive
- Only one directive-name may be specified per directive (true with Fortran also)
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

Conditional compilation: _OPENMP

- All OpenMP-compliant implementations define a macro named _OPENMP when the OpenMP compiler option is enabled.
- This macro can be used to include extra code at the preprocessing stage.
- Valid for both C and Fortran (requires .F or .F90 extension), although one can also use simply **!$** in version 2.0 and higher for Fortran (see specification).

```
#ifdef _OPENMP
 iam = omp_get_thread_num() + index;
#endif

!$ iam = omp_get_thread_num() + &
!$&        index
```

PARALLEL Region Construct

- A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.
- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.

Fortran format of PARALLEL construct

```
!$OMP PARALLEL [ clauses …
    PRIVATE (list)
    SHARED (list)
    DEFAULT (PRIVATE | SHARED | NONE)
    FIRSTPRIVATE (list)
    REDUCTION ({operator|intrinsic_procedure}: list)
    COPYIN (list)
    IF (scalar_logical_expression)
    NUM_THREADS(scalar_integer_expression)
     ]
  block
!OMP END PARALLEL
```

C/C++ format of parallel construct

```
#pragma omp parallel [ clauses ] new-line
 { structured-block }
```

Where clauses are:

```
    private(list)
    shared(list)
    default(shared | none)
    firstprivate(list)
    reduction(operator : variable-list)
    copyin(list)
    if(scalar_expression)
    num_threads(scalar_integer_expression)
```

Data scope attribute clauses

- An important consideration for OpenMP programming is the understanding and use of data scoping.
- Because OpenMP is based on the shared memory programming model, most variables are shared by default between the threads.
- Global variables include (shared by default):
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- Private variables include (private by default):
  - Loop index variables
  - Stack variables in subroutines called from parallel regions
  - Fortran: Automatic variables within a statement block

Data scope attributes clauses

- The clauses **`private(`*`list`*`)`**, **`shared(`*`list`*`)`**, **`default`** and **`firstprivate`** ***`(list)`*** allow the user to control the scope attributes of variables for the duration of the parallel region in which they appear. The variables are listed in brackets right after the clause.
- PRIVATE variables behave as follows:
  - A new object of the same type is declared once for each thread in the team
  - All references to the original object are replaced with references to the new object
  - Variables declared PRIVATE are uninitialized for each thread
- The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list. Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.

REDUCTION clause

- **reduction(operator : variable-list)**
- This clause performs a reduction on the variables that appear in *list*, with the *operator* or the intrinsic procedure specified.
- *Operator* is one of the following:
  - Fortran: +, *, -, .AND., .OR., .EQV., .NEQV., MIN, MAX
  - C/C++: +, *, -, &, ^, |, &&, ||, min, max
- The following are only available for Fortran: IAND, IOR, IEOR
- Variables that appear in a REDUCTION clause must be SHARED in the enclosing context. A private copy of each variables in *list* is created for each thread as if the PRIVATE clause had been used.

COPYIN clause

- The COPYIN clause provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team.
  - The THREADPRIVATE directive is used to make global file scope variables (C/C++) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions.
- List contains the names of variables to copy. In Fortran, the list can contain both the names of common blocks and named variables.
- The master thread variable is used as the copy source. The team threads are initialized with its value upon entry into the parallel construct.

IF clause

- If present, it must evaluate to .TRUE. (Fortran) or non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.
- Only a single IF clause can appear on the directive.

How many threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1. If the NUM_THREADS clause appears after the directive name, the number of threads specified is used for that parallel region.
  2. Use of the **omp_set_num_threads()** library function
  3. Setting of the **OMP_NUM_THREADS** environment variable
  4. Implementation default
- The threads are numbered from 0 (master thread) to N-1
- By default, a program with multiple parallel regions will use the same number of threads to execute each region. This behavior can be changed to allow the run-time system to dynamically adjust the number of threads that are created for a given parallel section. The **`num_threads`** clause is an example of this.

Fortran example of PARALLEL construct

```
PROGRAM REDUCTION
  INTEGER tnumber,I,J,K,OMP_GET_THREAD_NUM
  I=0; J=1; K=5
  PRINT *, "Before Par Region: I=",I," J=", J," K=",K

!$OMP PARALLEL PRIVATE(tnumber) REDUCTION(+:I)&
!$OMP  REDUCTION(*:J) REDUCTION(MAX:K)
  tnumber=OMP_GET_THREAD_NUM()
  I = I + tnumber
  J = J*tnumber
  K = MAX(K,tnumber)
  PRINT *, "Thread ",tnumber," I=",I," J=", J," K=",K
!$OMP END PARALLEL

  PRINT *, ""
  print *, "Operator            +     *    MAX"
  PRINT *, "After Par Region:  I=",I," J=", J," K=",K
END PROGRAM REDUCTION
```

C example of PARALLEL construct

```
#include <omp.h>
main () {
 int nthreads, tid;  /* Fork a team of threads giving
                       them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
  {            /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    /* Only master thread does this */
    if (tid == 0){
       nthreads = omp_get_num_threads();
       printf("Number of threads = %d\n", nthreads);
    }
  } /* All threads join master thread and terminate */
}
```

Work-Sharing Constructs

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.
- Types of Work-Sharing Constructs:
  - **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
  - **WORKSHARE** - divides the execution of the enclosed structured block into separate units of work

Work-Sharing Constructs Restrictions

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all.
- Successive work-sharing constructs must be encountered in the same order by all members of a team.

DO/for directive

- Purpose:
  - The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.
- Restrictions:
  - The DO loop can not be a DO WHILE loop, or a loop without loop control. Also, the loop iteration variable must be an integer and the loop control parameters must be the same for all threads.
  - Program correctness must not depend upon which thread executes a particular iteration.
  - It is illegal to branch out of a loop associated with a DO/for directive.

Format of DO construct

```
!$OMP DO [clause ...
       SCHEDULE (type [,chunk])
       ORDERED
       PRIVATE (list)
       FIRSTPRIVATE (list)
       LASTPRIVATE (list)
       SHARED (list)
       REDUCTION (operator | intrinsic : list)
       ]
   do_loop

!$OMP END DO [ NOWAIT ]
```

(C/C++) Format of “for” construct

```
#pragma omp for [clause ...] newline
 { for-loop }
```

Where clauses are:

```
    schedule (type [,chunk])
    ordered
    private(list)
    firstprivate(list)
    lastprivate(list)
    shared(list)
    reduction(operator : variable-list)
    nowait
```

SCHEDULE clause

- Describes how iterations of the loop are divided among the threads in the team. For both C/C++ and Fortran.

- STATIC:
  - Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
- DYNAMIC:
  - Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

SCHEDULE clause

- GUIDED:
  - The chunk size is exponentially reduced with each dispatched piece of the iteration space. The chunk size specifies the minimum number of iterations to dispatch each time.. The default chunk size is 1.
- RUNTIME:
  - The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
- The default schedule is implementation dependent. Implementation may also vary slightly in the way the various schedules are implemented.

DO/for directive clauses

- ORDERED:
  - Must be present when ORDERED directives are enclosed within the DO/for directive.
  - Should be used only for debugging…
- LASTPRIVATE(*list*)
  - The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.
  - The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct. For example, the team member which executes the final iteration for a DO section, or the team member which does the last SECTION of a SECTIONS context performs the copy with its own values.

NOWAIT clause

- If specified, then the threads do not synchronize at the end of the parallel loop. Threads proceed directly to the next statements after the loop.
- In C/C++, must be in lowercase: nowait

- For Fortran, the END DO directive is optional at the end of the loop.

Fortran example DO directive

```fortran
PROGRAM VEC_ADD_DO
  INTEGER N, CHUNKSIZE, CHUNK, I
  PARAMETER (N=1000)
  PARAMETER (CHUNKSIZE=100)
  REAL A(N), B(N), C(N)
 ! Some initializations
  DO I = 1, N
     A(I) = I * 1.0
     B(I) = A(I)
  ENDDO
  CHUNK = CHUNKSIZE
 !$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)
 !$OMP DO SCHEDULE(DYNAMIC,CHUNK)
  DO I = 1, N
     C(I) = A(I) + B(I)
  ENDDO
 !$OMP END DO NOWAIT
 !$OMP END PARALLEL
END
```

C/C++ example of “for” directive

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main()
{
 int i, chunk;
 float a[N], b[N], c[N];
 /* Some initializations */
 for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
 chunk = CHUNKSIZE;

 #pragma omp parallel shared(a,b,c,chunk) private(i)
 {
   #pragma omp for schedule(dynamic,chunk) nowait
   for (i=0; i < N; i++)
       c[i] = a[i] + b[i];
 } /* end of parallel section */
}
```

SECTIONS directive

- The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Independent SECTION directives are nested within a SECTIONS directive Each SECTION is executed once by a thread in the team. Different sections will be executed by different threads.

Fortran format of SECTIONS construct

```
!$OMP SECTIONS [clause ...
       PRIVATE (list)
       FIRSTPRIVATE (list)
       LASTPRIVATE (list)
       REDUCTION (operator | intrinsic : list)
       ]
[!$OMP SECTION]
   block

[!$OMP SECTION
   block ]
  …
!$OMP END SECTIONS [ NOWAIT ]
```

(C/C++) Format of sections construct

```
#pragma omp sections [clause ...] newline
 {
 [#pragma omp section newline]
   structured-block
 [#pragma omp section newline
   structured-block ]
  …
 }
```

Where clauses are:

```
    private(list)
    firstprivate(list)
    lastprivate(list)
    reduction(operator : variable-list)
    nowait
```

Fortran example SECTIONS directive

```
PROGRAM VEC_ADD_SECTIONS
  INTEGER N, I
  PARAMETER (N=1000)
  REAL A(N), B(N), C(N)
! Some initializations
  DO I = 1, N
    A(I) = I * 1.0
    B(I) = A(I)
  ENDDO
!$OMP PARALLEL SHARED(A,B,C), PRIVATE(I), NUM_THREADS(2)
!$OMP SECTIONS
!$OMP SECTION
   DO I = 1, N/2
     C(I) = A(I) + B(I)
   ENDDO
!$OMP SECTION
   DO I = 1+N/2, N
     C(I) = A(I) + B(I)
   ENDDO
!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL
END
```

C/C++ example of “sections” directive

```
#include <omp.h>
#define N 1000
main()
{
 int i, chunk;
 float a[N], b[N], c[N];
 /* Some initializations */
 for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i) num_threads(2)
  {
   #pragma omp sections nowait
     {
     #pragma omp section
     for (i=0; i < N/2; i++)
         c[i] = a[i] + b[i];
     #pragma omp section
     for (i=N/2; i < N; i++)
         c[i] = a[i] + b[i];
     } /* end of sections */
  } /* end of parallel section */
}
```

SINGLE directive

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
- May be useful when dealing with sections of code that are not thread safe (such as I/O).
- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a nowait (C/C++) or NOWAIT (Fortran) clause is specified.
- Format:
  - Fortran: **`!$OMP SINGLE`** ***`[clause…]`*** … **`!$OMP END SINGLE`**
  - C/C++: **`#pragma omp single`** ***`[clause ...] newline`***
  - Clauses: private(list), firstprivate(list), nowait

Combined Parallel Work-Sharing Constructs: PARALLEL DO

- This is one of the simplest and most useful constructs for fine-grain parallelism.

```
PROGRAM VEC_ADD_DO
  INTEGER N, I
  PARAMETER (N=1000)
  REAL A(N), B(N), C(N)
 ! Some initializations
 !$OMP PARALLEL DO        !By default, the static schedule
  DO I = 1, N            !will be used and the loop will
     A(I) = I * 1.0      !be divided in equal chunks
     B(I) = A(I)
  ENDDO    ! No need to put the END DO directive here

 !$OMP PARALLEL DO SHARED(A,B,C) PRIVATE(I)
  DO I = 1, N
     C(I) = A(I) + B(I)
  ENDDO
END
```

Combined Parallel Work-Sharing Constructs: “parallel for”

```
#include <omp.h>
#define N 1000
main()
{
 int i;
 float a[N], b[N], c[N];

 /* Some parallel initialization */
 #pragma omp parallel for
 for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;

 #pragma omp parallel for private(i)
 for (i=0; i < N; i++)
       c[i] = a[i] + b[i];
}
```

Synchronization Constructs

- Let two threads on two different processors both trying to increment a variable x at the same time (assume x is initially 0):

```
THREAD 1:
increment(x)
 { x = x + 1; }

THREAD 1:
10 LOAD A, (x address)
20 ADD A, 1
30 STORE A, (x address)
```

```
THREAD 2:
increment(x)
 { x = x + 1; }

THREAD 2:
10 LOAD B, (x address)
20 ADD B, 1
30 STORE B, (x address)
```

One possible execution sequence:

1. Thread 1 loads the value of x into register A.
2. Thread 2 loads the value of x into register B.
3. Thread 1 adds 1 to register A
4. Thread 2 adds 1 to register B
5. Thread 1 stores register A at location x
6. Thread 2 stores register B at location x

The resultant value of x will be 1, not 2 as it should be.

Synchronization Constructs

- To avoid the situation shown on the previous slide, the increment of x must be synchronized between the two threads to insure that the correct result is produced.
- OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads:
  - OMP MASTER
  - OMP CRITICAL
  - OMP BARRIER
  - OMP ATOMIC
  - OMP FLUSH
  - OMP ORDERED

Example…

```fortran
real(8):: density(mzeta,mgrid)
density=0.

  do m=1,me
     kk=kzelectron(m)
     ij=jtelectron0(m)
     density(kk,ij) = density(kk,ij) + wzelectron(m)
  enddo
```

What happens when trying to parallelize this loop with OpenMP?

→ Need to protect updates to density array!

One solution…

```
real(8) :: density(mzeta,mgrid),dnitmp(mzeta,mgrid)

!$omp parallel private(dnitmp)
  dnitmp=0.    ! Set array elements to zero
!$omp do private(m,kk,ij)
   do m=1,me
      kk=kzelectron(m)
      ij=jtelectron0(m)
      dnitmp(kk,ij) = dnitmp(kk,ij) + wzelectron(m)
   enddo

!$omp critical
  do ij=1,mgrid
     do kk=1,mzeta
        density(kk,ij) = density(kk,ij) + dnitmp(kk,ij)
     enddo
  enddo
!$omp end critical
!$omp end parallel
```

THREADPRIVATE directive

- The THREADPRIVATE directive is used to make global file scope variables (C/C++) or Fortran common blocks and modules local and persistent to a thread through the execution of multiple parallel regions.
- The directive must appear after the declaration of listed variables/common blocks. Each thread then gets its own copy of the variable/common block, so data written by one thread is not visible to other threads.
- Format:
  - Fortran: !$OMP THREADPRIVATE (/cb/, …)
  - C/C++: #pragma omp threadprivate (list)

THREADPRIVATE directive

- On first entry to a parallel region, data in THREADPRIVATE variables, common blocks, and modules should be assumed undefined, unless a COPYIN clause is specified in the PARALLEL directive.
- THREADPRIVATE variables differ from PRIVATE variables because they are able to persist between different parallel sections of a code.
- Data in THREADPRIVATE objects are guaranteed to persist only if the dynamic threads mechanism is "turned off" and the number of threads in different parallel regions remains constant. The default setting of dynamic threads is undefined (although usually “static” in practice).

Fortran example of THREADPRIVATE

```
      PROGRAM THREADPRIV
      INTEGER ALPHA(10), BETA(10), I
      COMMON /A/ ALPHA
!$OMP THREADPRIVATE(/A/)
C     Explicitly turn off dynamic threads
      CALL OMP_SET_DYNAMIC(.FALSE.)
C     First parallel region
!$OMP PARALLEL PRIVATE(BETA, I)
      DO I=1,10
         ALPHA(I) = I
         BETA(I) = I
      END DO
!$OMP END PARALLEL
C     Second parallel region
!$OMP PARALLEL
      PRINT *, 'ALPHA(3)=',ALPHA(3), ' BETA(3)=',BETA(3)
!$OMP END PARALLEL
      END
```

C/C++ example of threadprivate construct

```
#include <omp.h>

int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)

main() {
/* Explicitly turn off dynamic threads */
 omp_set_dynamic(0);

/* First parallel region */
 #pragma omp parallel private(i,beta)
 for (i=0; i < 10; i++)
    alpha[i] = beta[i] = i;

/* First parallel region */
 #pragma omp parallel
 printf("alpha[3]= %d and beta[3]= %d\n",alpha[3],beta[3]);

}
```

OpenMP library routines

- The OpenMP standard defines an API for library calls that perform a variety of functions:
  - Query the number of threads/processors, set number of threads to use
  - General purpose locking routines (semaphores)
  - Set execution environment functions: nested parallelism, dynamic adjustment of threads.
- For C/C++, it is necessary to specify the include file "omp.h".
- For the Lock routines/functions:
  - The lock variable must be accessed only through the locking routines
  - For Fortran, the lock variable should be of type integer and of a kind large enough to hold an address.
  - For C/C++, the lock variable must have type omp_lock_t or type omp_nest_lock_t, depending on the function being used.

OMP_SET_NUM_THREADS

- Sets the number of threads that will be used in the next parallel region.
- Format:
  - Fortran
    - SUBROUTINE OMP_SET_NUM_THREADS(scalar_integer_expression)
  - C/C++
    - void omp_set_num_threads(int num_threads)
- Notes & Restrictions:
- The dynamic threads mechanism modifies the effect of this routine.
  - Enabled: specifies the maximum number of threads that can be used for any parallel region by the dynamic threads mechanism.
  - Disabled: specifies exact number of threads to use until next call to this routine.
- This routine can only be called from the serial portions of the code
- This call has precedence over the OMP_NUM_THREADS environment variable

OMP_GET_NUM_THREADS

- Purpose:
  - Returns the number of threads that are currently in the team executing the parallel region from which it is called.
- Format:
  - Fortran
    - INTEGER FUNCTION OMP_GET_NUM_THREADS()
  - C/C++
    - int omp_get_num_threads(void)
- Notes & Restrictions:
  - If this call is made from a serial portion of the program, or a nested parallel region that is serialized, it will return 1.
  - The default number of threads is implementation dependent.

OMP_GET_THREAD_NUM

- Returns the thread number of the thread, within the team, making this call. This number will be between 0 and OMP_GET_NUM_THREADS-1. The master thread of the team is thread 0
- Format:
  - Fortran
    - INTEGER FUNCTION OMP_GET_THREAD_NUM()
  - C/C++
    - int omp_get_thread_num(void)
- Notes & Restrictions:
  - If called from a nested parallel region, or a serial region, this function will return 0.

Example of omp_get_thread_num

CORRECT:

```
PROGRAM HELLO

  INTEGER TID, OMP_GET_THREAD_NUM

!$OMP PARALLEL PRIVATE(TID)
    TID = OMP_GET_THREAD_NUM()
    PRINT *, 'Hello World from thread = ', TID ...
!$OMP END PARALLEL

END
```

INCORRECT:

- **TID must be PRIVATE**

```
PROGRAM HELLO

  INTEGER TID, OMP_GET_THREAD_NUM

!$OMP PARALLEL
    TID = OMP_GET_THREAD_NUM()
    PRINT *, 'Hello World from thread = ',
       TID ...
!$OMP END PARALLEL

END
```

Other functions and subroutines

- OMP_GET_MAX_THREADS()
  – Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function.

- OMP_GET_NUM_PROCS()
  – Returns the number of processors that are available to the program.

- OMP_IN_PARALLEL
  – May be called to determine if the section of code which is executing is parallel or not.

More functions…

- OMP_SET_DYNAMIC()
  - Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions.

- OMP_GET_DYNAMIC()
  - Used to determine if dynamic thread adjustment is enabled or not.

- OMP_SET_NESTED()
  - Used to enable or disable nested parallelism.

- OMP_GET_NESTED
  - Used to determine if nested parallelism is enabled or not.

OpenMP “locking” functions

- OMP_INIT_LOCK()
  - This subroutine initializes a lock associated with the lock variable.
- OMP_DESTROY_LOCK()
  - This subroutine disassociates the given lock variable from any locks.
- OMP_SET_LOCK()
  - This subroutine forces the executing thread to wait until the specified lock is available. A thread is granted ownership of a lock when it becomes available.
- OMP_UNSET_LOCK()
  - This subroutine releases the lock from the executing subroutine.
- OMP_TEST_LOCK()
  - This subroutine attempts to set a lock, but does not block if the lock is unavailable.

OpenMP timing functions

- OMP_GET_WTIME()
  - This function returns a double precision value equal to the elapsed wallclock time in seconds since some arbitrary time in the past.
  - The times returned are “per-thread times”.
- OMP_GET_WTICK()
  - This function returns a double precision value equal to the number of seconds between successive clock ticks.

- Consults the OpenMP specification for more details on all the subroutines and functions:
  - http://www.openmp.org/specs

OpenMP environment variables

- OpenMP provides four environment variables for controlling the execution of parallel code.
- All environment variable names are uppercase. The values assigned to them are not case sensitive.

- **OMP_SCHEDULE**
  - Applies only to DO, PARALLEL DO (Fortran) and for, parallel for (C/C++) directives which have their schedule clause set to RUNTIME. The value of this variable determines how iterations of the loop are scheduled on processors. For example:
    - **setenv OMP_SCHEDULE "guided, 4"**
    - **setenv OMP_SCHEDULE "dynamic"**

OpenMP environment variables…

- **OMP_NUM_THREADS** (the most used…)
  - Sets the maximum number of threads to use during execution. For example: **setenv OMP_NUM_THREADS 8**

- **OMP_DYNAMIC**
  - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE. For example: **setenv OMP_DYNAMIC TRUE**

- **OMP_NESTED** (rarely implemented on current systems)
  - Enables or disables nested parallelism. Valid values are TRUE or FALSE. For example: **setenv OMP_NESTED TRUE**

Example: parallelize our π test code with OpenMP

```
      program fpi
      double precision  PI25DT
      parameter        (PI25DT = 3.141592653589793238462643d0)
      double precision  mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, j, ierr


      open(12,file='ex4.in',status='old')
      read(12,*) n
      close(12)
      write(*,*)'  number of intervals=',n
c
      h = 1.0d0/n
      sum  = 0.0d0

!omp parallel do private(i,x) shared(h,n) reduction(+:sum)
      do i = 1, n
         x = h * (dble(i) - 0.5d0)
         sum = sum + 4.d0/(1.d0 + x*x)
      enddo
      mypi = h * sum
c
      pi = mypi
      write(*,*)' pi=',pi,'  Error=',abs(pi - PI25DT)

      end
```

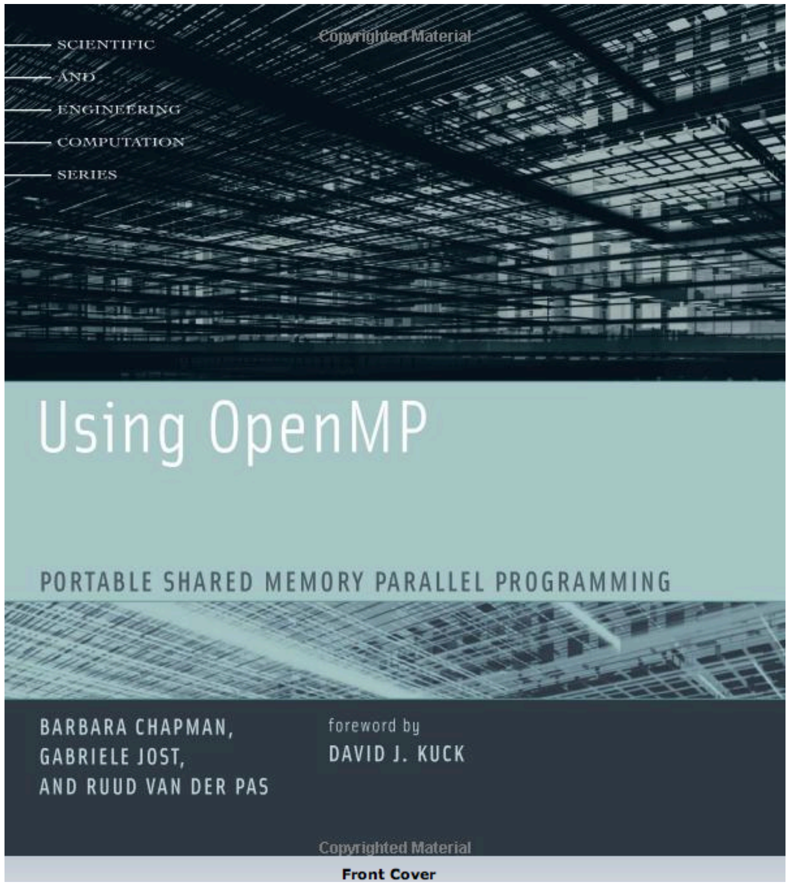
Summation ordering issues

- What happens when we use a different integrand? such as:

$$f(x) = \frac{1}{e^{\alpha(x-1)} + 1}$$

- If the numbers that you add up together are separated by a large order of magnitude (very small and very large numbers together), the order of the summation matters. Ideally you need to sort all the numbers and add them in ascending order (smallest to largest) otherwise you will get a different answer. If the variation is in the last 3 or 4 digits of a double precision number it might not matter.
- During an OpenMP reduction, the order in which the threads add up the numbers varies from one run to another. Something to keep in mind…

Books on OpenMP



SCIENTIFIC AND ENGINEERING COMPUTATION SERIES

Copyrighted Material

Using OpenMP

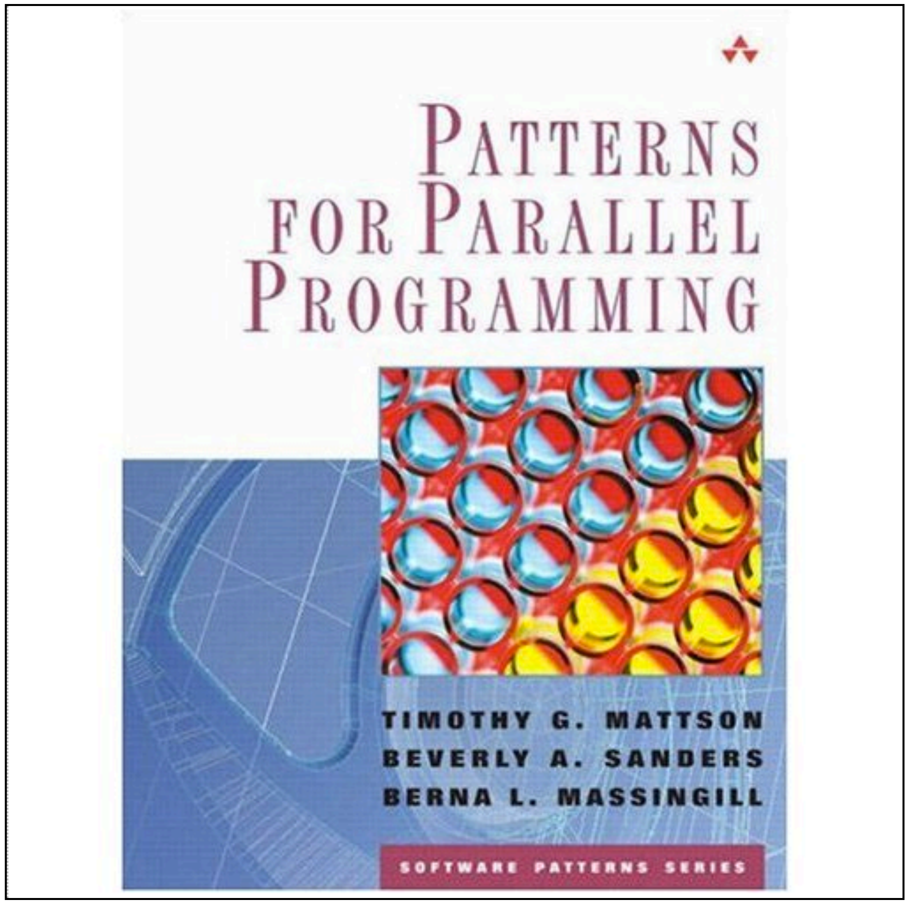
PORTABLE SHARED MEMORY PARALLEL PROGRAMMING

BARBARA CHAPMAN, GABRIELE JOST, AND RUUD VAN DER PAS

foreword by DAVID J. KUCK

Copyrighted Material

Front Cover



PATTERNS FOR PARALLEL PROGRAMMING

TIMOTHY G. MATTSON

BEVERLY A. SANDERS

BERNA L. MASSINGILL

SOFTWARE PATTERNS SERIES

References

- 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/
- See http://openmp.org/wp/resources/

Mixing MPI and OpenMP together in the same application

Why use both MPI and OpenMP in the same code?

- To save memory by not having to replicate data common to all processes, not using ghost cells, sharing arrays, etc.
- To optimize interconnect bandwidth usage by having only one MPI process per NUMA node.
- Although MPI generally scales very well it has its limit, and OpenMP gives another avenue of parallelism.
- Some compilers have now implemented OpenMP-like directives to run sections of a code on general-purpose GPU (GPGPU). Fine-grain parallelism with OpenMP directives is easy to port to GPUs.
- Processes are “heavy” while threads are “light” (fast creation and context switching).

Implementing mixed-model

- Easiest and safest way:
  - Coarse grain MPI with fine grain loop-level OpenMP
  - All MPI calls are done outside the parallel regions
  - This is always supported
- Allowing the master thread to make MPI calls inside a parallel region
  - Supported by most if not all MPI implementations
- Allowing ALL threads to make MPI calls inside the parallel regions
  - Requires MPI to be fully thread safe
  - Not the case for all implementations
  - Can be tricky…

Find out the level of support of your MPI library

MPI-2 “Init” functions for multi-threaded MPI processes:

```
     int MPI_Init_thread(      int * argc, char ** argv[],
                         int thread_level_required,
                        int * thead_level_provided);
    int MPI_Query_thread(     int * thread_level_provided);
              int MPI_Is_main_thread(int * flag);
```

- “Required” values can be:
  - **MPI_THREAD_SINGLE:** **<u>Only one thread</u> will execute**
  - **MPI_THREAD_FUNNELED:** **<u>Only master thread will make MPI-calls</u>**
  - **MPI_THREAD_SERIALIZED:** **Multiple threads may make MPI-calls, but <u>only one at a time</u>**
  - **MPI_THREAD_MULTIPLE:** **Multiple threads may call MPI,without <u>restrictions</u>**
- “Provided” returned value can be less than “required” value

Compiling and linking mixed code

- Just add the “openmp” compiler option to the compile AND link lines (if separate from each other):
  - mpicc –openmp mpi_omp_code.c –o a.out
  - mpif90 –openmp mpi_omp_code.f90 –o a.out

Launching a mixed job

- Set OMP_NUM_THREADS and then launch job with “mpirun” or “mpiexec”
- If you have an “UMA-type” node where all the cores have the same level of access to memory, use a single MPI process per node by using a hostfile (or machinefile) on a cluster

```
 export OMP_NUM_THREADS=4
 mpiexec -n 4 -hostfile mfile a.out >& out < in &

%cat mfile
 machine1.princeton.edu
 machine2.princeton.edu
 machine3.princeton.edu
 machine4.princeton.edu
```

- This job will use 16 cores: 4 MPI processes with 4 OpenMP threads each

How to deal with the batch system

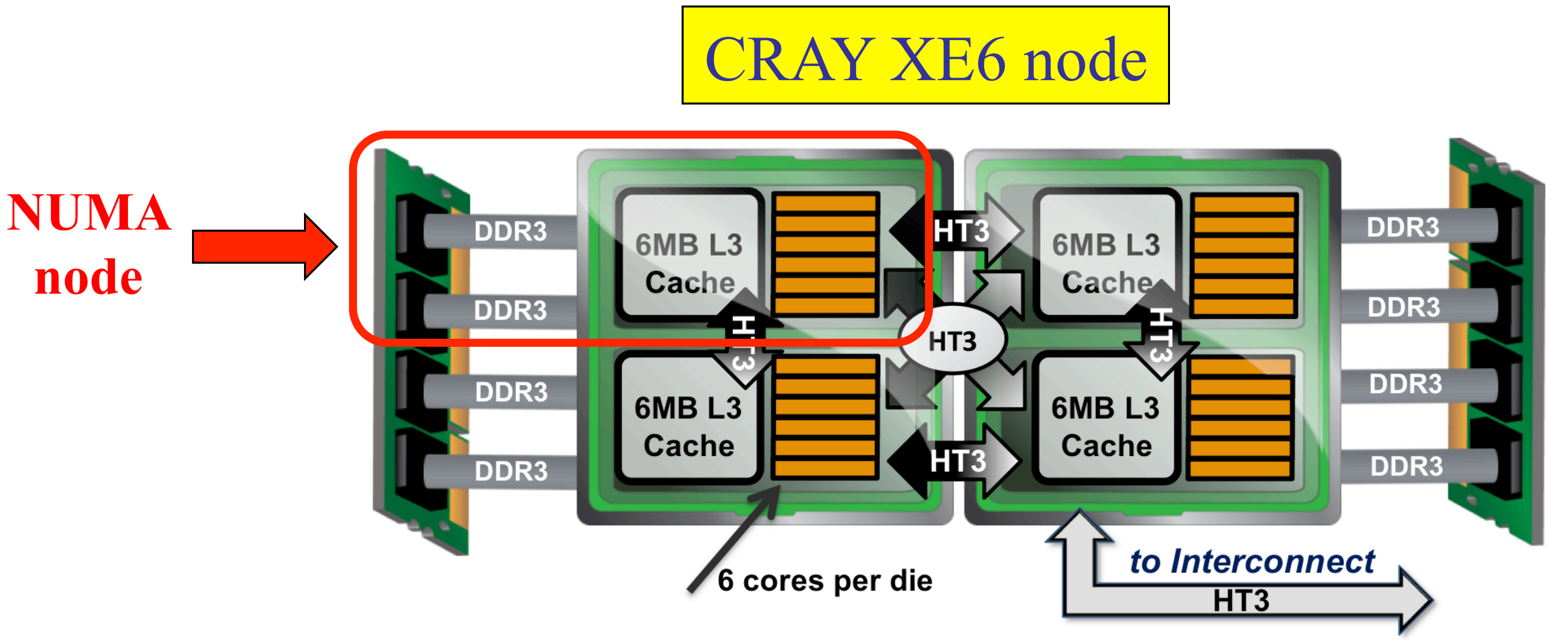
- When submitting a job to a batch system, such as PBS, you do not know in advance which nodes you will get.
- However, that information is stored in $PBS_NODEFILE
- If you launch a single MPI process per node you can do:

```
/bin/cat $PBS_NODEFILE | uniq > mfile
nprocs=`wc -l mfile | awk '{print $1}'`

export OMP_NUM_THREADS=8
mpiexec -n $nprocs -hostfile mfile ./mpi_omp_code
```

- Make sure that the number of cores that you are asking for equals nprocs*OMP_NUM_THREADS

Watch out for NUMA

- On non-uniform memory access (NUMA) nodes, one needs to be careful how the memory gets assigned to reach good performance.
- When in doubt, use 1 MPI process per “NUMA node”



Launching a NUMA-aware job on the CRAY XE6

- Commands to launch a job that uses 1 MPI process per NUMA node on the XE6:

```
#PBS -l mppwidth=96
#PBS -V
#PBS -l walltime=1:00:00
cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=6
aprun -n 16 -d 6 -N 4 -S 1 -ss ./hybrid.x
```

- You can still use a single MPI process per node but should use a “first-touch” method to allocate memory close to the threads

What about OpenACC???

Running codes on GPUs? What's the big deal?

- NVIDIA GeForce GTX 690
- Based on latest Kepler technology
- 2X GK104 GPU chips
- **3072 CUDA cores**
- 384 GB/s memory bandwidth
- **4.58 Teraflops single precision performance**
- 0.19 Tflops double precision
- Frequency 1.019 GHz
- 4 GB GDDR5 memory
- **For awesome double precision performance, go for the Tesla K20/K40 hardware**

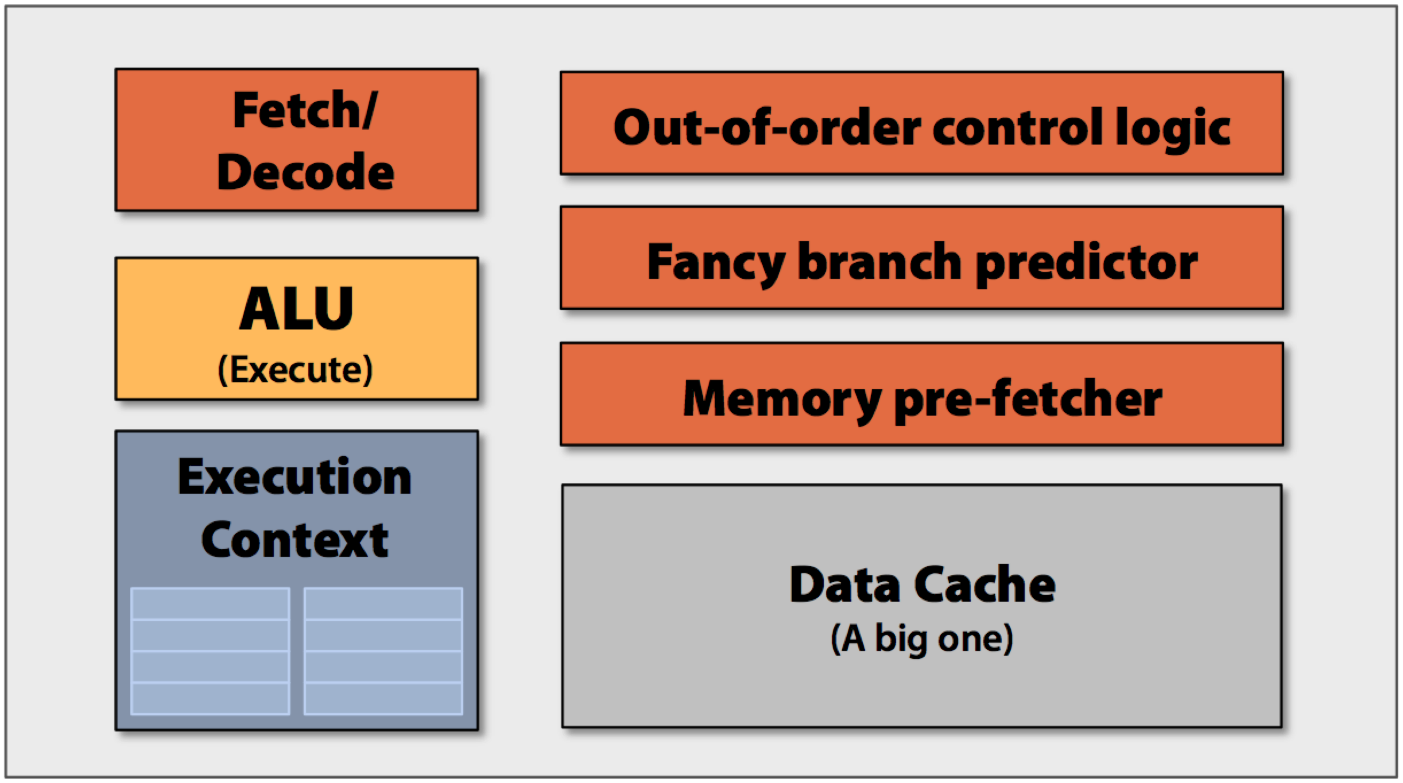
Side-by-side comparison of CPU vs. GPU

Processor	Intel Xeon E5-2690 "Sandy Bridge"	NVIDIA K20 Tesla GPU "Kepler" (Fermi)
No. of cores per "node"	16 cores (2x 8/chip)	**2,880** (512)
Frequency	2.9 GHz	~1 GHz
Memory bandwidth	51.2 GB/sec	**~320 GB/s (177)**
Peak Flops	371 GFlops	~**2000 Gflops (665)**
Memory (shared)	32 – 64 GB	**6 GB**

What is the difference between a GPU and a CPU?

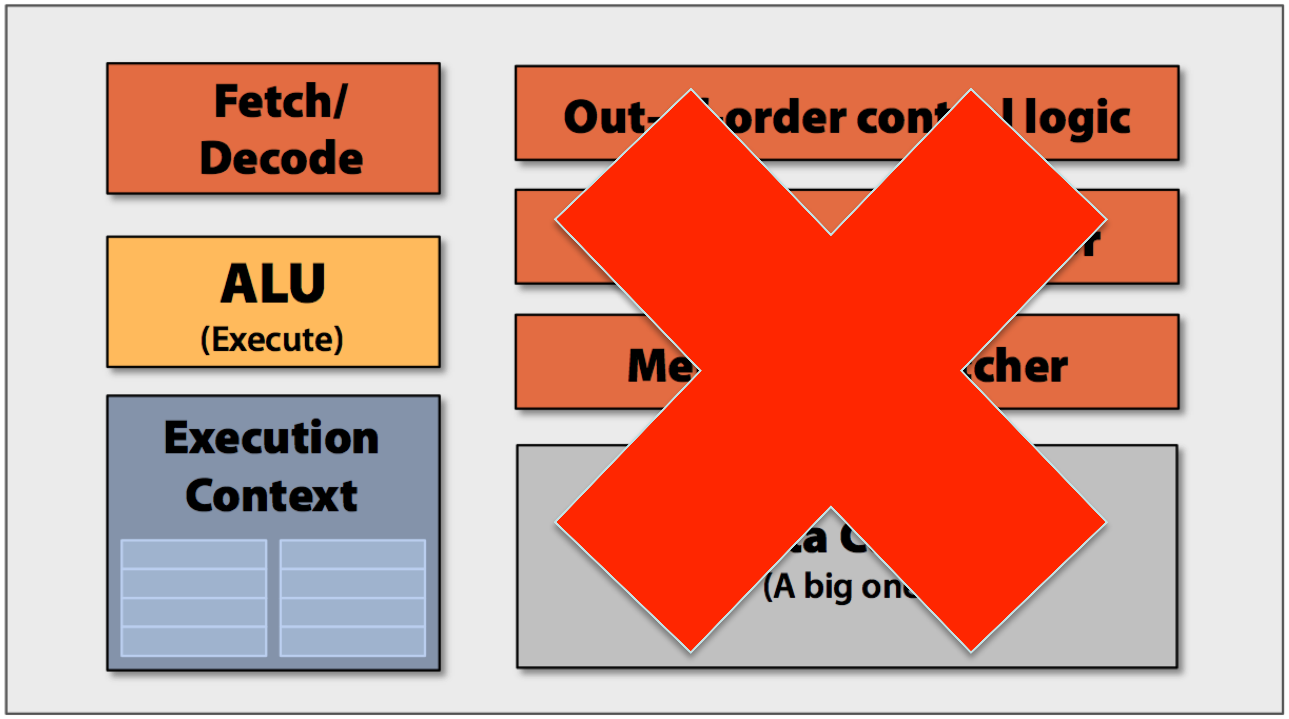
- How different is the GPU in terms of hardware architecture?
  - NVIDIA has been introducing more “scientific-code-friendly” hardware in its GPUs, such as more and faster double precision units, memory error correction (ECC), IEEE standard operations, etc.
  - The introduction of the “TESLA” model of GPU by NVIDIA has taken the hardware beyond the “gaming” business
- What makes it achieve such high performance compared to CPU?
- Why do we need CPUs at all then?!!?

Start with a "CPU-style" core…



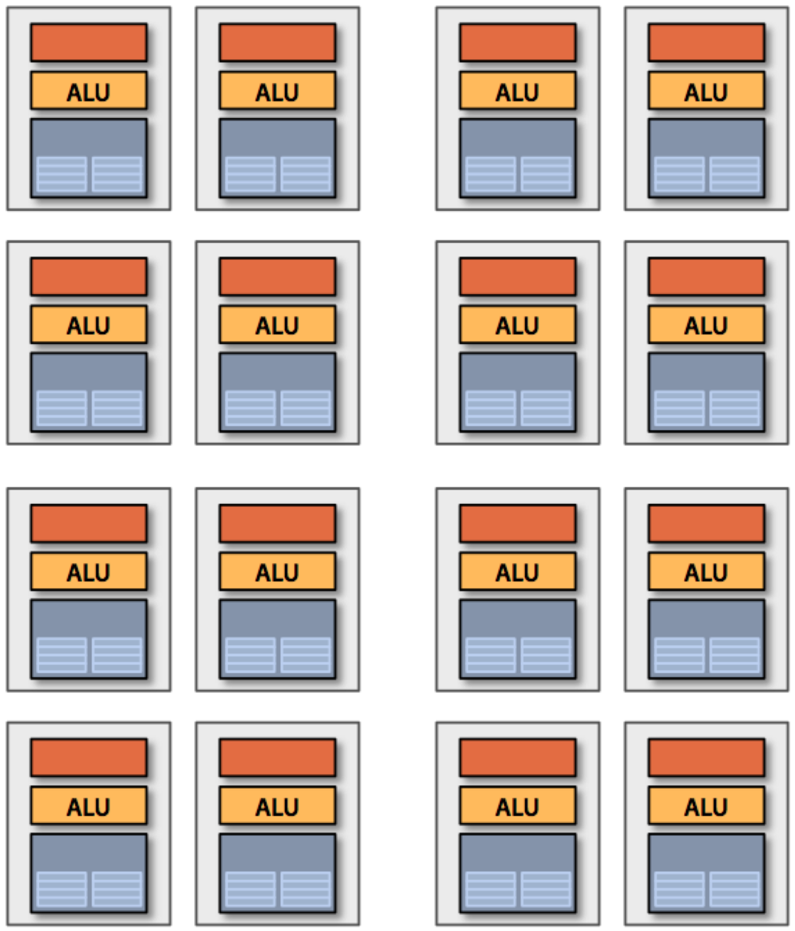
http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

Remove everything that makes a single instruction stream go fast



http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

Put many of these simple cores together



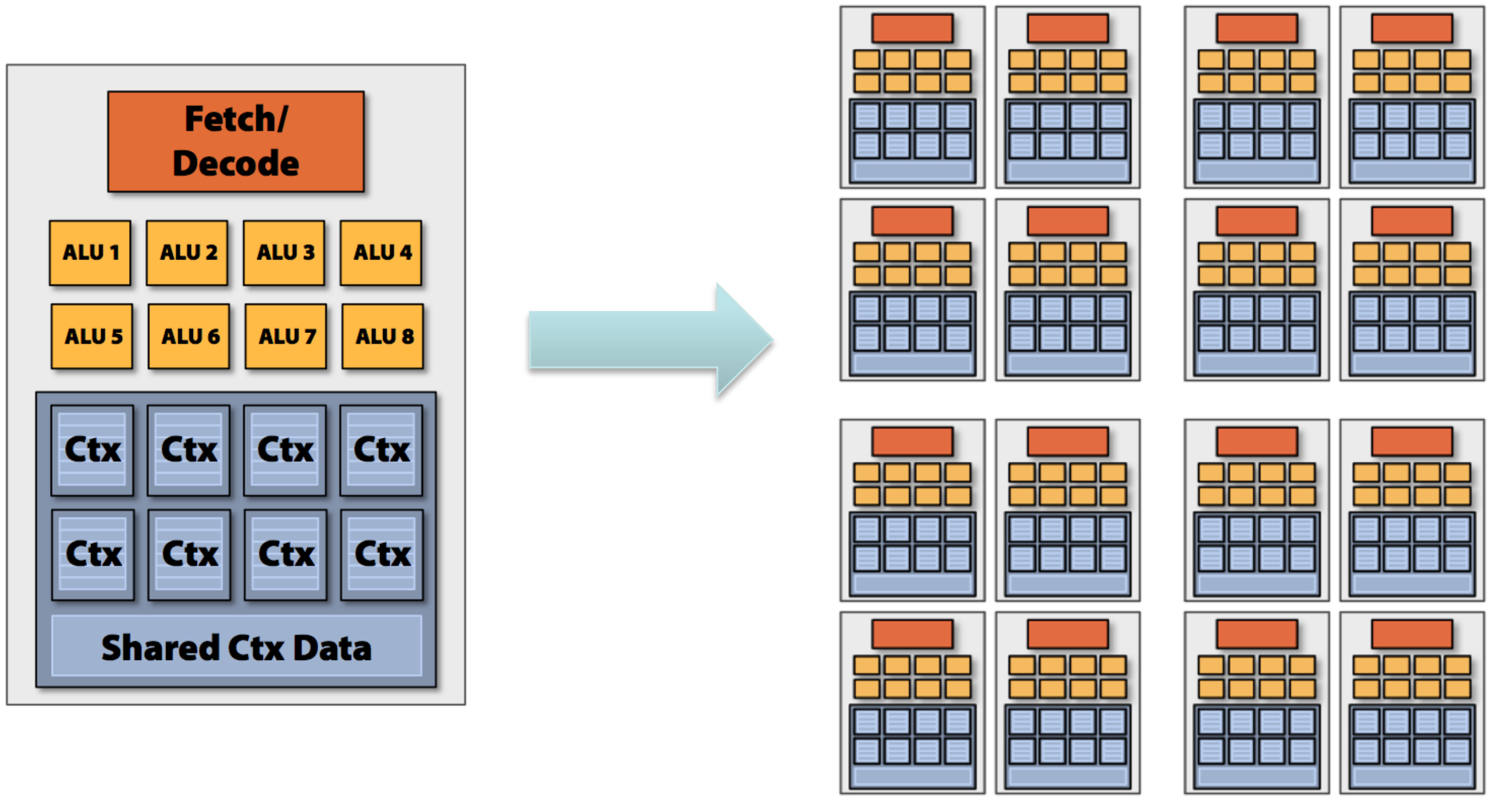
- Let’s not forget the original purpose of the GPU though:
  - Same operations applied to a large number of vertices, pixels, polygons, …

= DATA PARALLEL or SIMD (Single Instruction Multiple Data)

Better known in our community as **VECTOR PROCESSING**

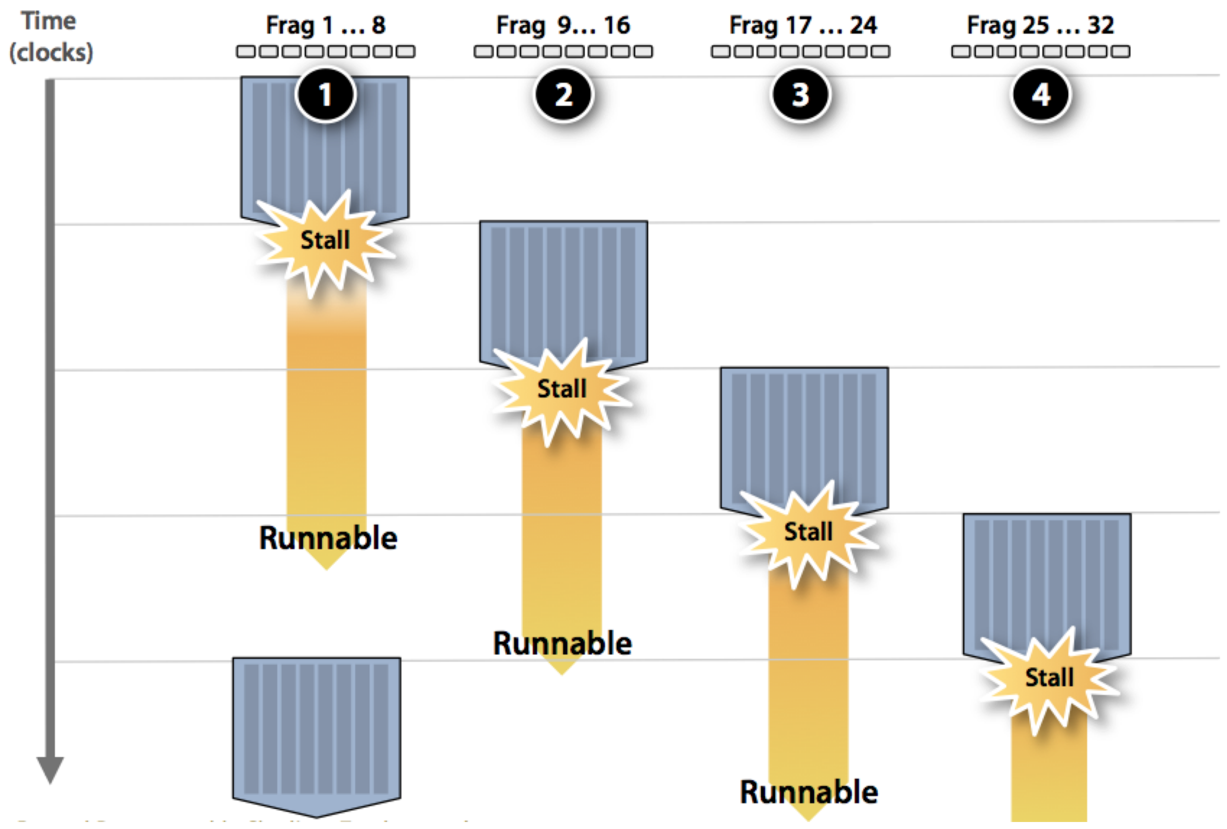
http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

So let's add some arithmetic units and have them share a stream of instructions



http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

The secret to GPU high throughput: massive multi-threading + interleaving



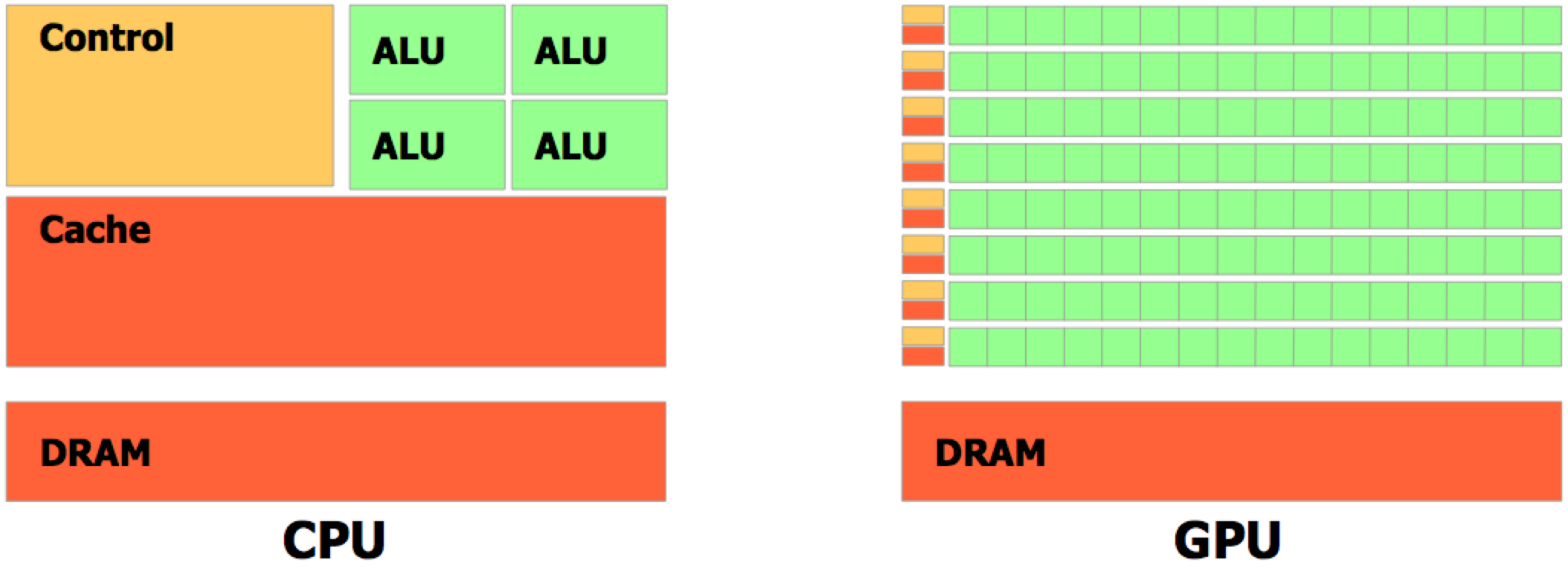
NOT SIMD
But rather

<u>SIMT</u>!
Single
Instruction
Multiple
Threads

255 registers per thread!!

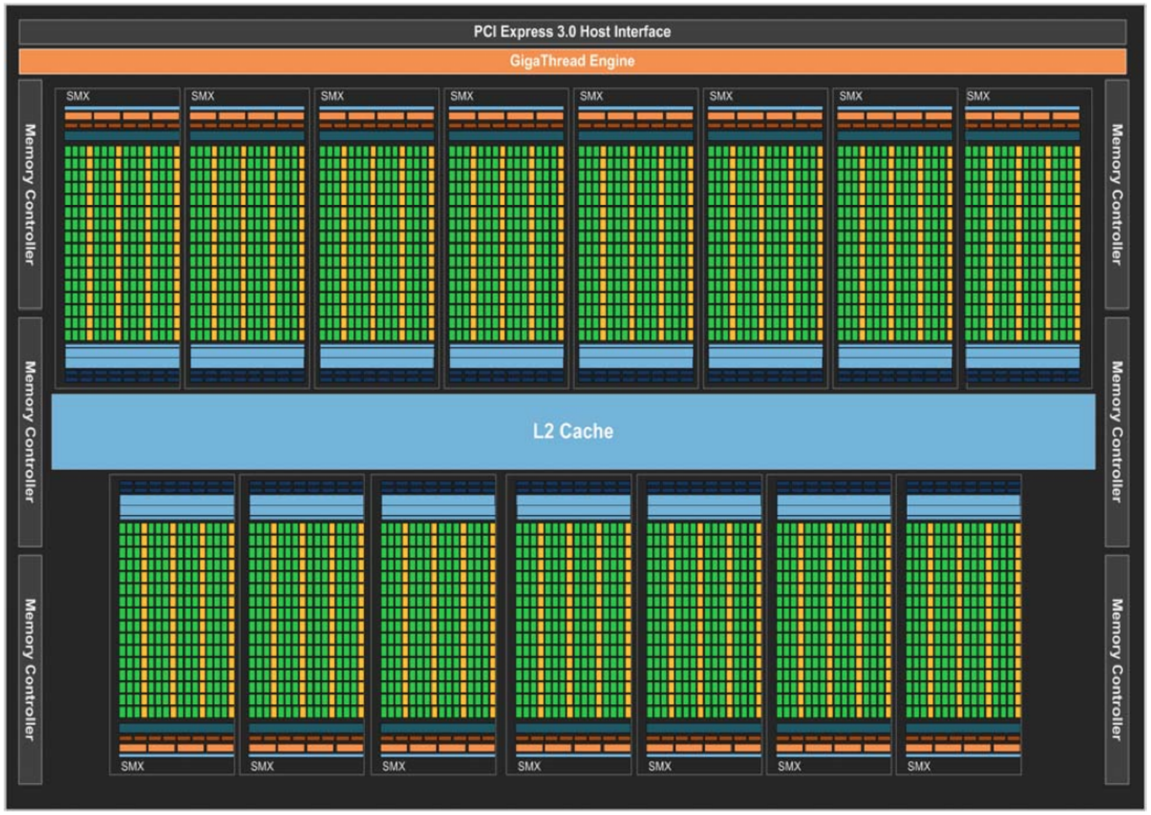
http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

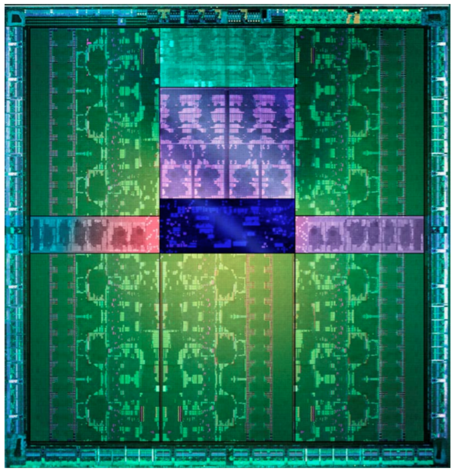
Classic picture of CPU vs. GPU



NVIDIA Kepler GPU

GK110 chip (15 SMX, 2048 threads/SMX, 7.1 billion transistors)

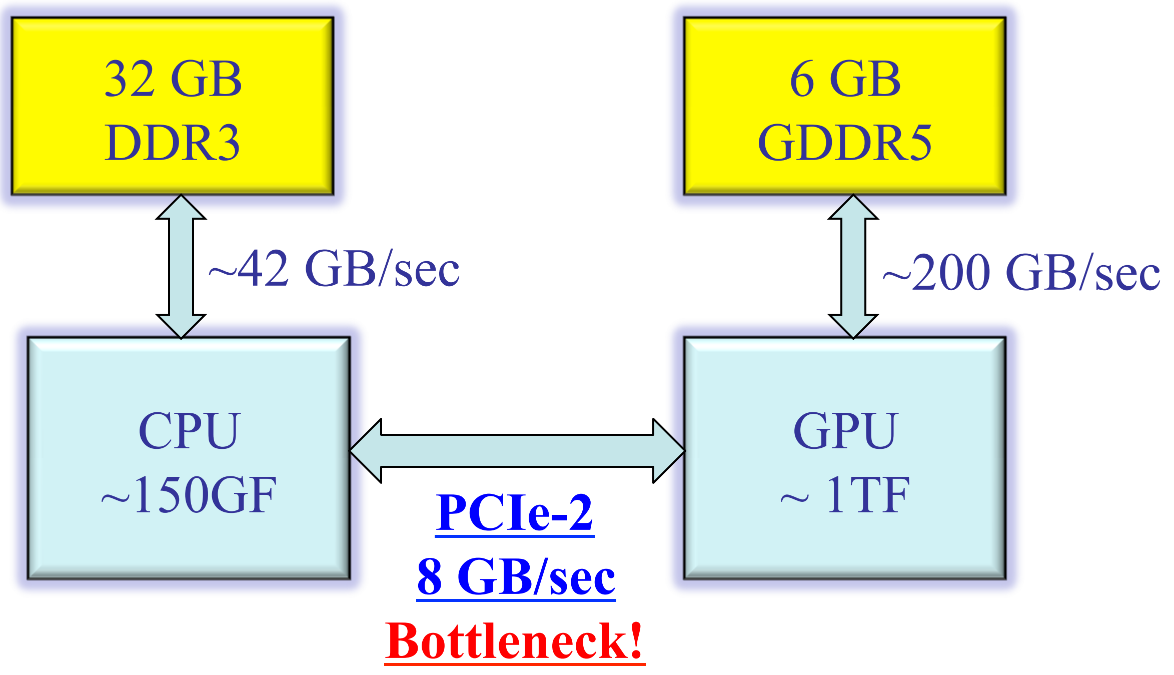




Using and programming GPUs

- Several possibilities
  - OpenGL: computer graphics functions used by game developers. NOT a good idea for scientific codes!!
  - CUDA: NVIDIA-specific programming language built as an extension of standard C language. Best approach to get the most out of your GPU. CUDA kernels not portable though. Also available for FORTRAN but only through the PGI compiler.
  - OpenACC compiler directives similar to OpenMP. Portable code. Easy to get started. Available for a few compilers.
  - Libraries, commercial software, domain-specific environments, . . .
  - OpenCL: open standard, platform- and vendor independent
    - Works on both GPU AND CPU!!
    - Even harder than CUDA though…

Hardware considerations affecting GPU programming



- Keep "kernel" data resident on GPU memory as much as possible
- Avoid frequent copying between CPU and GPU
- Use asynchronous, non-blocking, communication, multi-level overlapping

What to do first…

- MOST IMPORTANT:
  - Find and expose as much parallelism as you can in your code.
  - You need LOTS of parallel operations to keep the GPU busy!
  - Try to remove as many dependencies as you can between successive iterations in a loop.
  - The ideal case is when each iteration is completely independent from the others ➔ VECTORIZATION

Goals of OpenACC

- Same idea as OpenMP
  - Simplify the programming of complex hardware by using directives (hides a lot of the complexity)
  - Least changes to your code
  - Portable across different accelerators (nvidia, ATI, Intel Xeon Phi) as opposed to CUDA, which works only on nvidia GPUs
- Works for Fortran, C, C++
- When running the same code on a multi-core CPU you can use OpenMP directives instead of OpenACC and run in parallel on the CPU as well
  - Use preprocessor macros "_OPENACC" and "_OPENMP" to direct compilation according to available hardware
- http://www.openacc-standard.org
- http://www.pgroup.com/doc/openACC_gs.pdf
- Version 1.0 was limited in terms of features but 2.0 is much, much better

Example of OpenACC directive

It can be as simple as the following:

```
subroutine smooth( a, b, w0, w1, w2, n, m)
 real, dimension(:,:) :: a,b
 real :: w0, w1, w2
 integer :: n, m
 integer :: i, j
!$acc parallel loop
   do i = 2,n-1
    do j = 2,m-1
     a(i,j)= w0 * b(i,j) + &
        w1 * (b(i-1,j) + b(i,j-1) + b(i+1,j) + b(i,j+1)) + &
        w2 * (b(i-1,j-1) + b(i-1,j+1) + b(i+1,j-1) + b(i+1,j+1))
    enddo
   enddo
```

CUDA *vs* OpenACC

- Simple example: **REDUCTION**

```
a=0.0
do i = 1,n
   a = a + b(i)
end do
```

“Simple” CUDA implementation

```
__global__ void reduce0(int *g_idata, int *g_odata)
{
extern __shared__ int sdata[];

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();

for(unsigned int s=1; s < blockDim.x; s *= 2) {
if ((tid % (2*s)) == 0) {
sdata[tid] += sdata[tid + s];
}
__syncthreads();
}

if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}


extern "C" void reduce0_cuda_(int *n, int *a, int *b)
{
int *b_d,red;
const int b_size = *n;


cudaMalloc((void **) &b_d , sizeof(int)*b_size);
cudaMemcpy(b_d, b, sizeof(int)*b_size,
cudaMemcpyHostToDevice);
```

```
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(2048, 1, 1);
dim3 small_dimGrid(16, 1, 1);

int smemSize = 128 * sizeof(int);
int *buffer_d, *red_d;
int *small_buffer_d;

cudaMalloc((void **) &buffer_d , sizeof(int)*2048);
cudaMalloc((void **) &small_buffer_d ,
sizeof(int)*16);
cudaMalloc((void **) &red_d , sizeof(int));


reduce0<<< dimGrid, dimBlock, smemSize >>>(b_d,
buffer_d);

reduce0<<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d);

reduce0<<< 1, 16, smemSize >>>(small_buffer_d,
red_d);

cudaMemcpy(&red, red_d, sizeof(int),
cudaMemcpyDeviceToHost);

*a = red;

cudaFree(buffer_d);
cudaFree(small_buffer_d);
cudaFree(b_d);
}
```

Ref: SC13 OpenACC tutorial, Luiz DeRose, Alistair Hart, Heidi Poxon, & James Beyer

Simple "CUDA" implementation (cont.)

```
template<class T>
struct SharedMemory
{
    __device__ inline operator       T*()
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }

    __device__ inline operator const T*() const
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }
};

template <class T, unsigned int blockSize, bool nIsPow2>
__global__ void
reduce6(T *g_idata, T *g_odata, unsigned int n)
{
    T *sdata = SharedMemory<T>();

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    T mySum = 0;
    while (i < n)
    {
        mySum += g_idata[i];
        if (nIsPow2 || i + blockSize < n)
            mySum += g_idata[i+blockSize];
        i += gridSize;
    }
sdata[tid] = mySum;
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = mySum
+ sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = mySum
+ sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] = mySum = mySum
+ sdata[tid +  64]; } __syncthreads(); }
```

```
  if (tid < 32)
   {
      volatile T* smem = sdata;
      if (blockSize >=  64) { smem[tid] = mySum = mySum + smem[tid + 32];  }
      if (blockSize >=  32) { smem[tid] = mySum = mySum + smem[tid + 16];  }
      if (blockSize >=  16) { smem[tid] = mySum = mySum + smem[tid + 8];  }
      if (blockSize >=   8) { smem[tid] = mySum = mySum + smem[tid + 4];  }
      if (blockSize >=   4) { smem[tid] = mySum = mySum + smem[tid + 2];  }
      if (blockSize >=   2) { smem[tid] = mySum = mySum + smem[tid + 1];  }
   }

   if (tid == 0)
      g_odata[blockIdx.x] = sdata[0];
}
extern "C" void reduce6_cuda_(int *n, int *a, int *b)
{
   int *b_d;
   const int b_size = *n;

   cudaMalloc((void **) &b_d , sizeof(int)*b_size);
   cudaMemcpy(b_d, b, sizeof(int)*b_size, cudaMemcpyHostToDevice);

   dim3 dimBlock(128, 1, 1);
   dim3 dimGrid(128, 1, 1);
   dim3 small_dimGrid(1, 1, 1);
   int smemSize = 128 * sizeof(int);
   int *buffer_d;
   int small_buffer[4],*small_buffer_d;

   cudaMalloc((void **) &buffer_d , sizeof(int)*128);
   cudaMalloc((void **) &small_buffer_d , sizeof(int));
   reduce6<int,128,false><<< dimGrid, dimBlock, smemSize >>>(b_d,buffer_d, b_size);
   reduce6<int,128,false><<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d,128);
   cudaMemcpy(small_buffer, small_buffer_d, sizeof(int),
cudaMemcpyDeviceToHost);

   *a = *small_buffer;

   cudaFree(buffer_d);
   cudaFree(small_buffer_d);
   cudaFree(b_d);
}
```

OpenACC version of the Reduction code

```
!$acc data present(a,b,n)
   a = 0.0
!$acc update device(a)
!$acc parallel
!$acc loop reduction(+:a)
   do i = 1,n
      a = a + b(i)
   end do
!$acc end parallel
!$acc end data
```

Another example and why it won’t work well

```
PROGRAM main
 INTEGER :: a(N)
 <stuff>
!$acc parallel loop
  DO i = 1,N
     a(i) = I
  ENDDO
!$acc end parallel loop
!$acc parallel loop
  DO i = 1,N
     a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop
<stuff>
END PROGRAM main
```

- Two accelerator parallel region
- Compiler creates two kernels
  - Loop iterations automatically divided across gangs,workers,vectors
  - Breaking parallel region acts as barrier
- First kernel initialises array
  - Compiler will determine copyout(a)
- Second kernel updates array
  - Compiler will determine copy(a)
- Breaking parallel region=barrier
- **Array a(:) unnecessarily moved from and to GPU between kernels**
  - **"data sloshing"**

Another version that works better…

```
PROGRAM main
 INTEGER :: a(N)
 <stuff>
!$acc data copyout(a)
!$acc parallel loop
  DO i = 1,N
     a(i) = I
  ENDDO
!$acc end parallel loop
!$acc parallel loop
  DO i = 1,N
     a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop
!$acc end data
<stuff>
END PROGRAM main
```

- Now added a data region
- Specified arrays only moved at boundaries of data region
- Unspecified arrays moved by each kernel
- No compiler-determined movements for data regions
- Data region can contain host code and accelerator regions
- Copies of arrays independent
- **No automatic synchronisation of copies within data region**
  - **User-directed synchronisation via update directive**

Most important OpenACC directives

http://www.openacc.org/sites/default/filesOpenACC_API_QuickRefGuide.pdf

- parallel loop reduction(op, var)   where op = +, *, min, max, …
- copyin: a shared variable that is used read-only in the loopnest
- copyout: a shared variable that is used write-only in the loopnest
- copy: a shared variable that is used read-write in the loopnest
- create: a shared variable that is a temporary in the loopnest
- present: tells the compiler that the data is already on GPU. No need to copy
- present_or_copy: if data not present on GPU copy from host

  Data clauses can accept array section arguments
- Data scoping:
  - shared: all loop iterations all process the same version of the variable
  - private: each loop iteration uses variable separately
  - firstprivate: same as private except for initialization

- **Compile with pgcc or pgf95 with "-acc" option!**

That’s all for now…
Please check `www.openacc.org`

Thanks and

Happy parallel programming!