Parallel Computing Using MPI, OpenMP, CUDA with examples and debugging, tracing, profiling of parallel programs on the Discovery Cluster.

The session is structured as follows:

- Using MPI on Discovery and brief MPI features (10 minutes)
- OpenMP – Programming with Shared Memory (10 minutes)
- NVIDIA CUDA 6.5 SDK (10 minutes)
- MPI/OpenMP brief examples (15 minutes)
- CUDA/OpenCL brief examples (15 minutes)
- Hybrid MPI, OpenMP example runs on non-GPU nodes (15 minutes)
- Hybrid MPI, OpenMP, CUDA example runs on GPU nodes (15 minutes)
- Debugging, Tracing and Profiling with MPE (10 minutes)
- Debugging, Tracing and Profiling with HPCToolKit (10 minutes)
- Questions (10 minutes)

Northeastern University Research Computing: Nilay K Roy, MS Computer Science, Ph.D Computational Physics

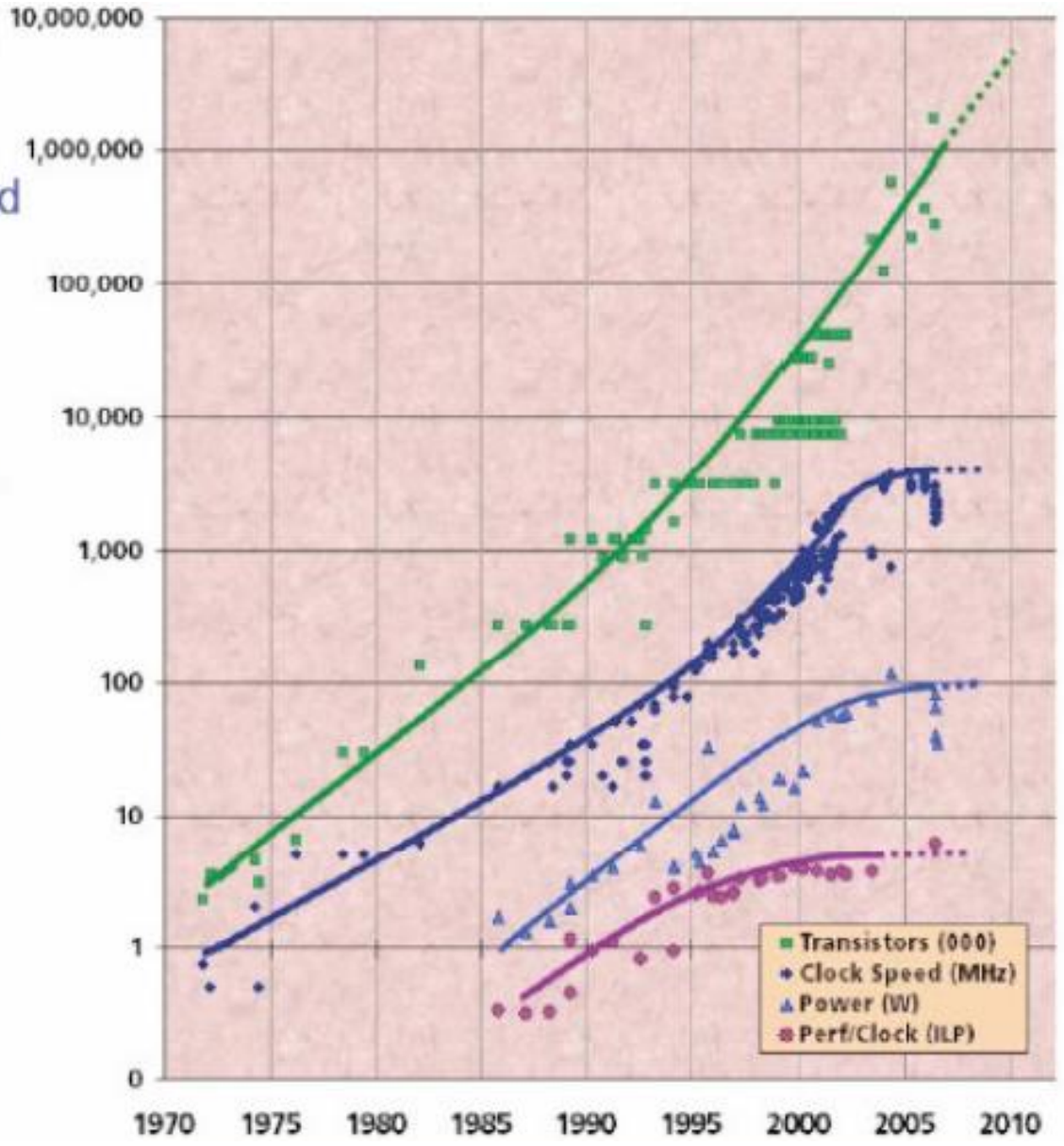
Why Parallelization …?

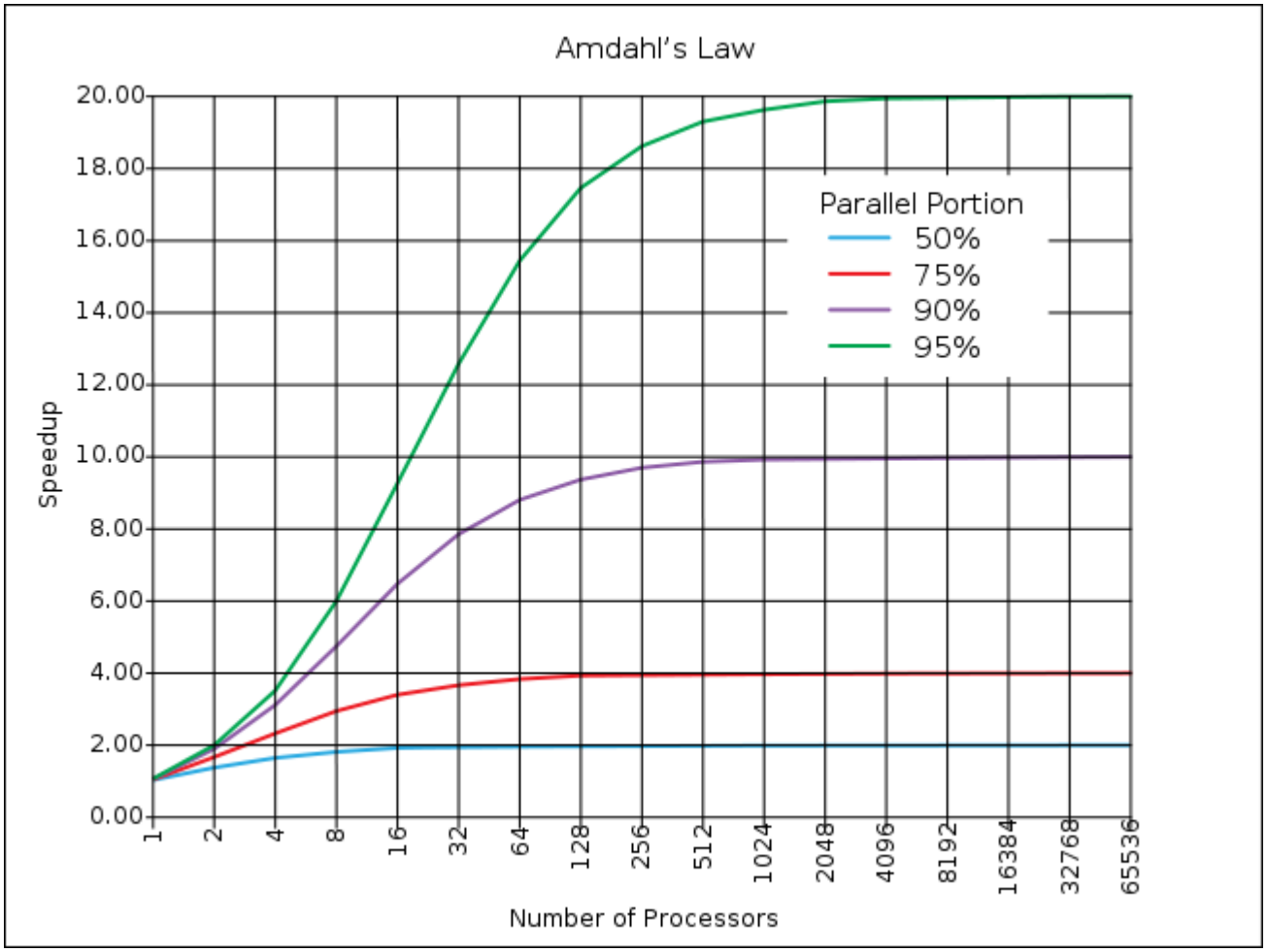
New Constraints

- 15 years of *exponential* clock rate growth has ended

But Moore's Law continues!

- How do we use all of those transistors to keep performance increasing at historical rates?
- Industry Response: #cores per chip doubles every 18 months *instead* of clock frequency!

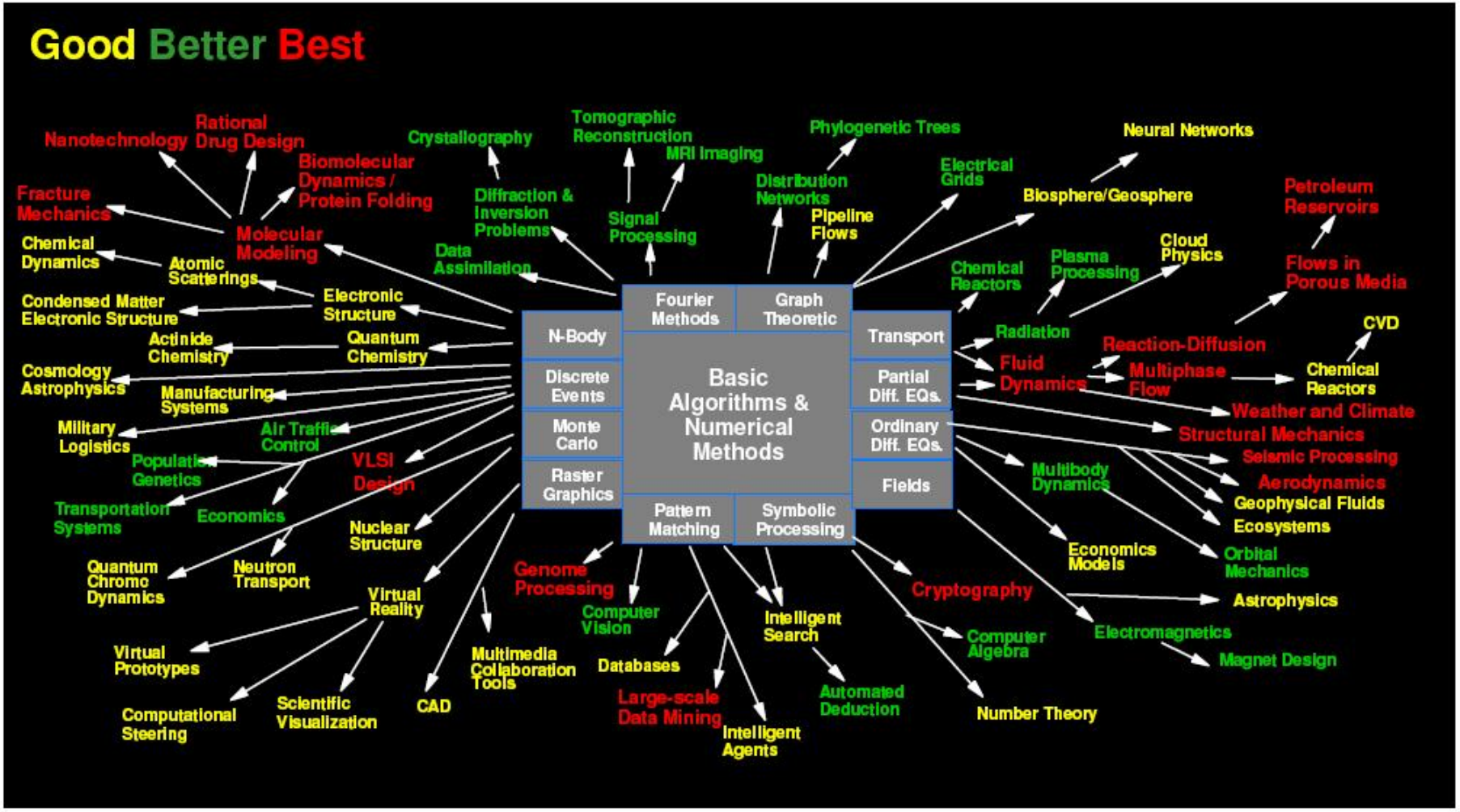




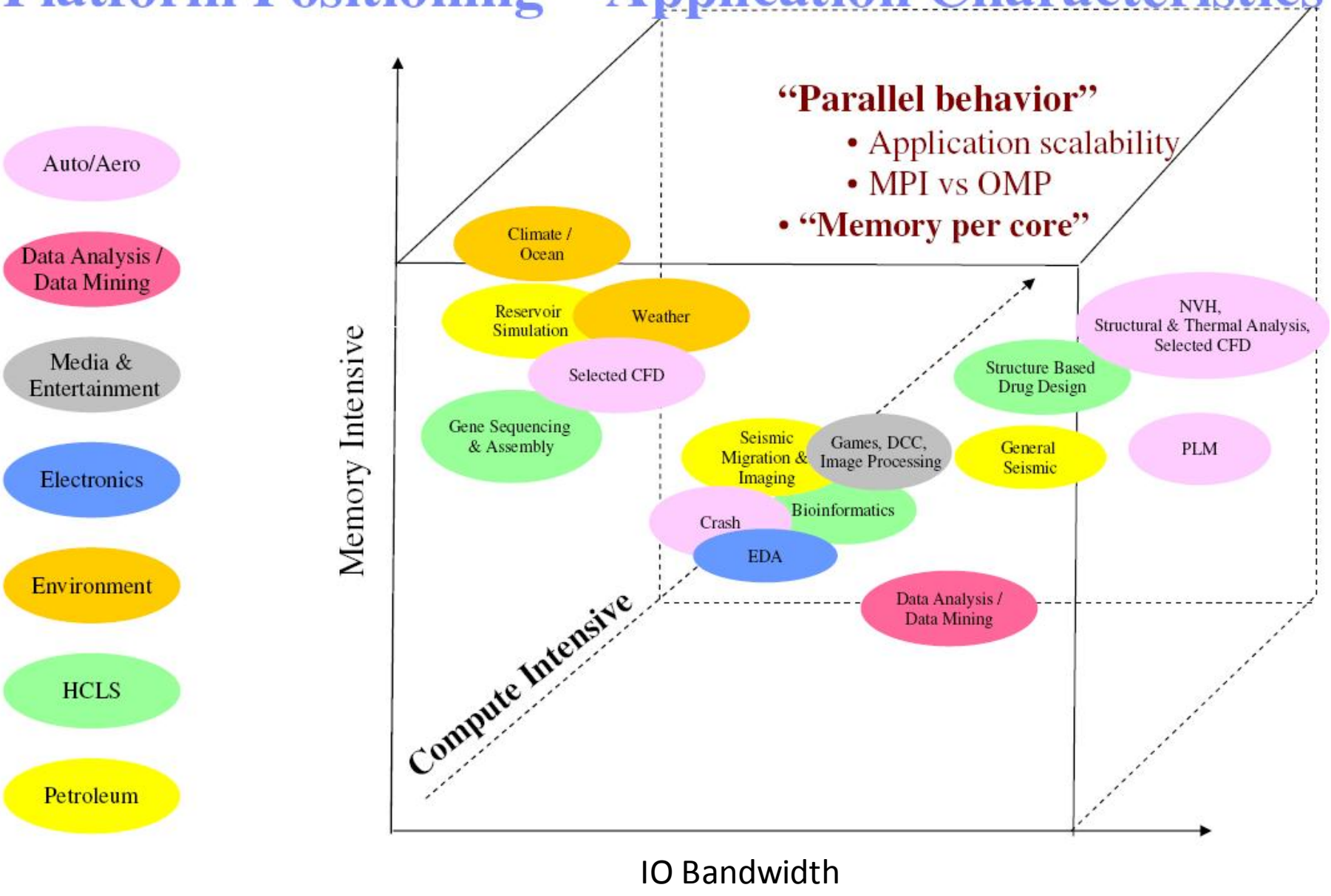
•Amdahl's law states that if *P* is the proportion of a program that can be made parallel, and (1–*P*) is the proportion that cannot be parallelized, then the maximum speedup that can be achieved by using *N* processors = P/[(1-P) + (P/N)]

•95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20×, no matter how many processors are used

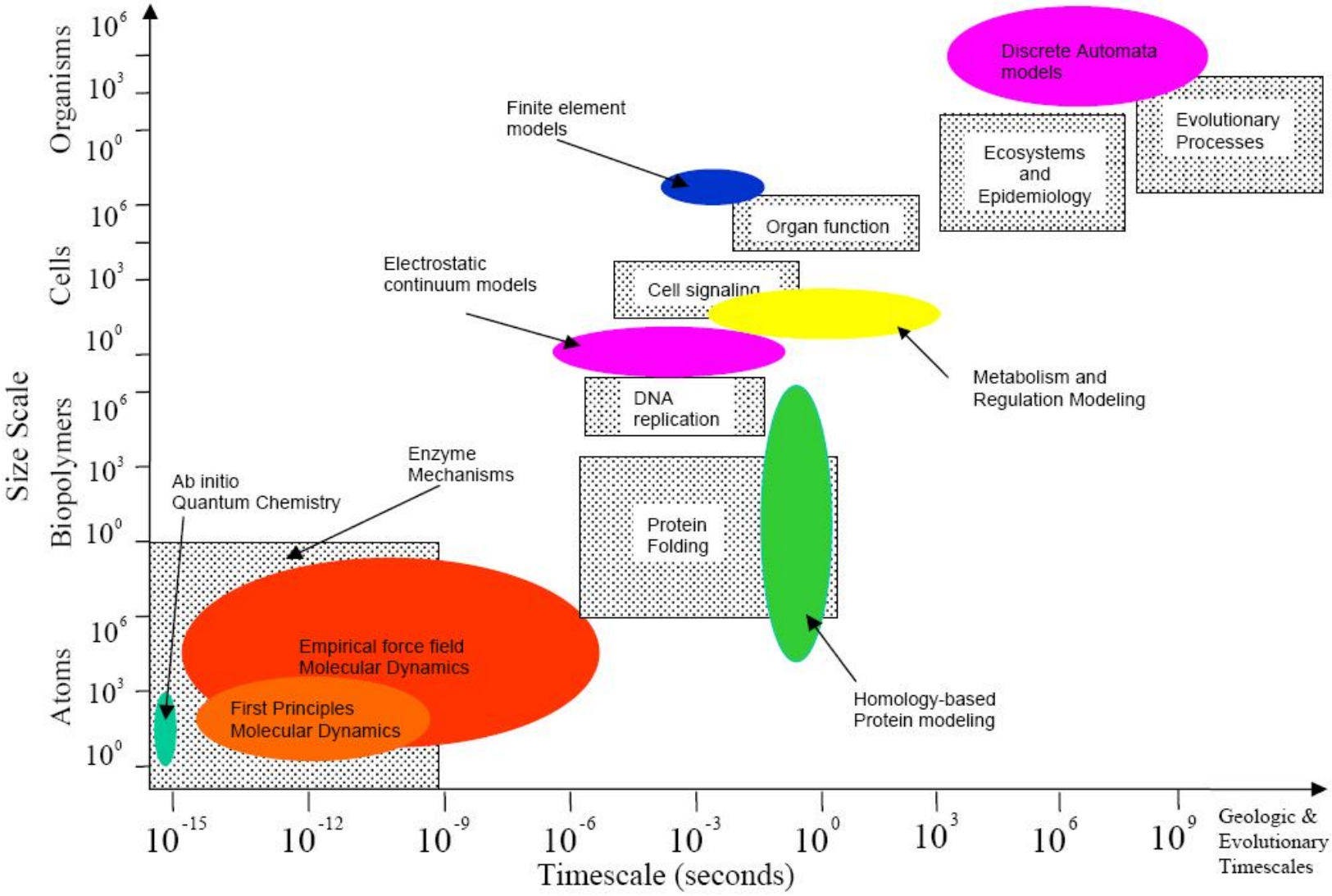
Many Computational Science Modeling and Simulation Algorithms and Numerical Methods are Massively Parallel

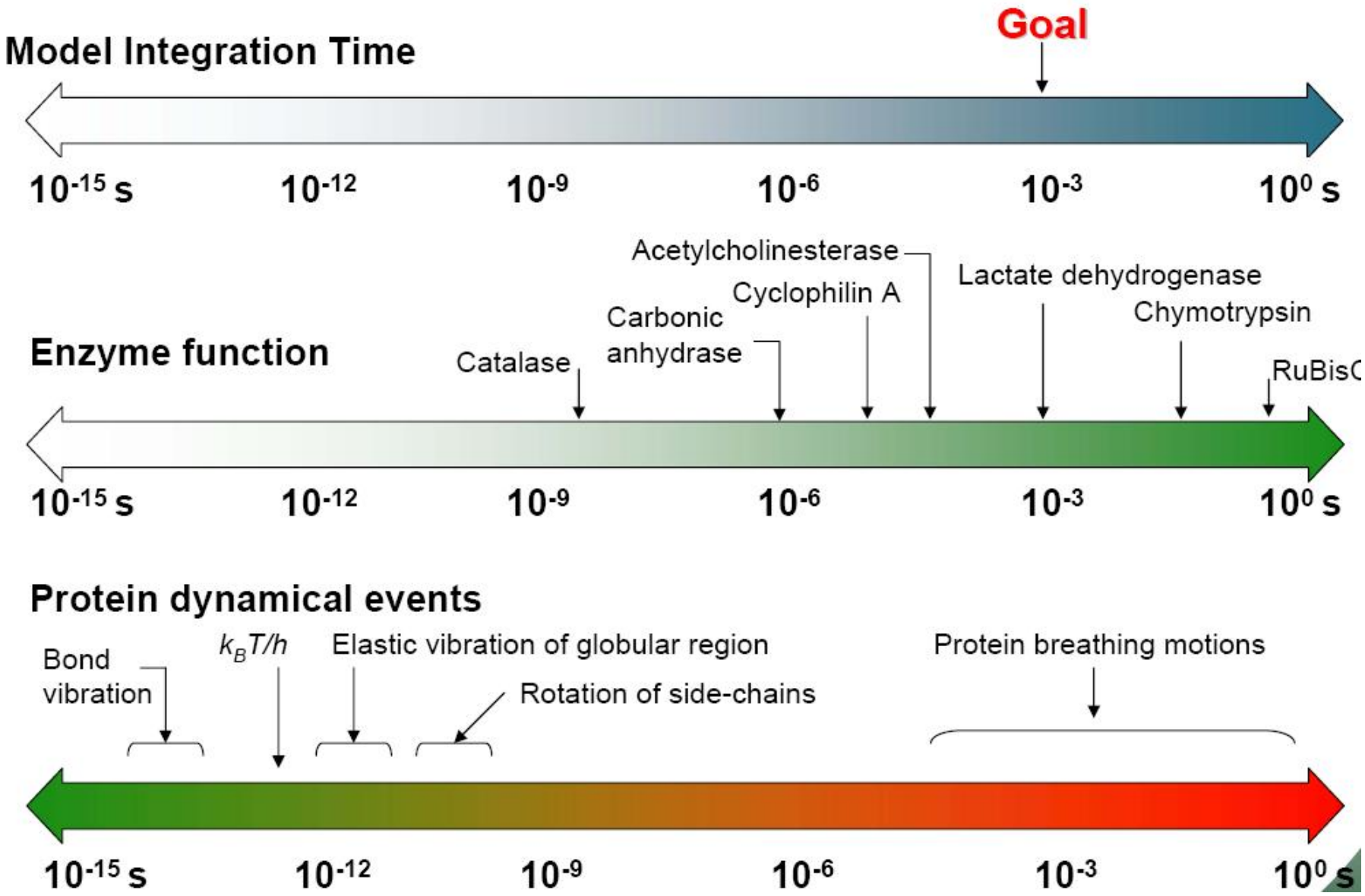


Platform Positioning – Application Characteristics

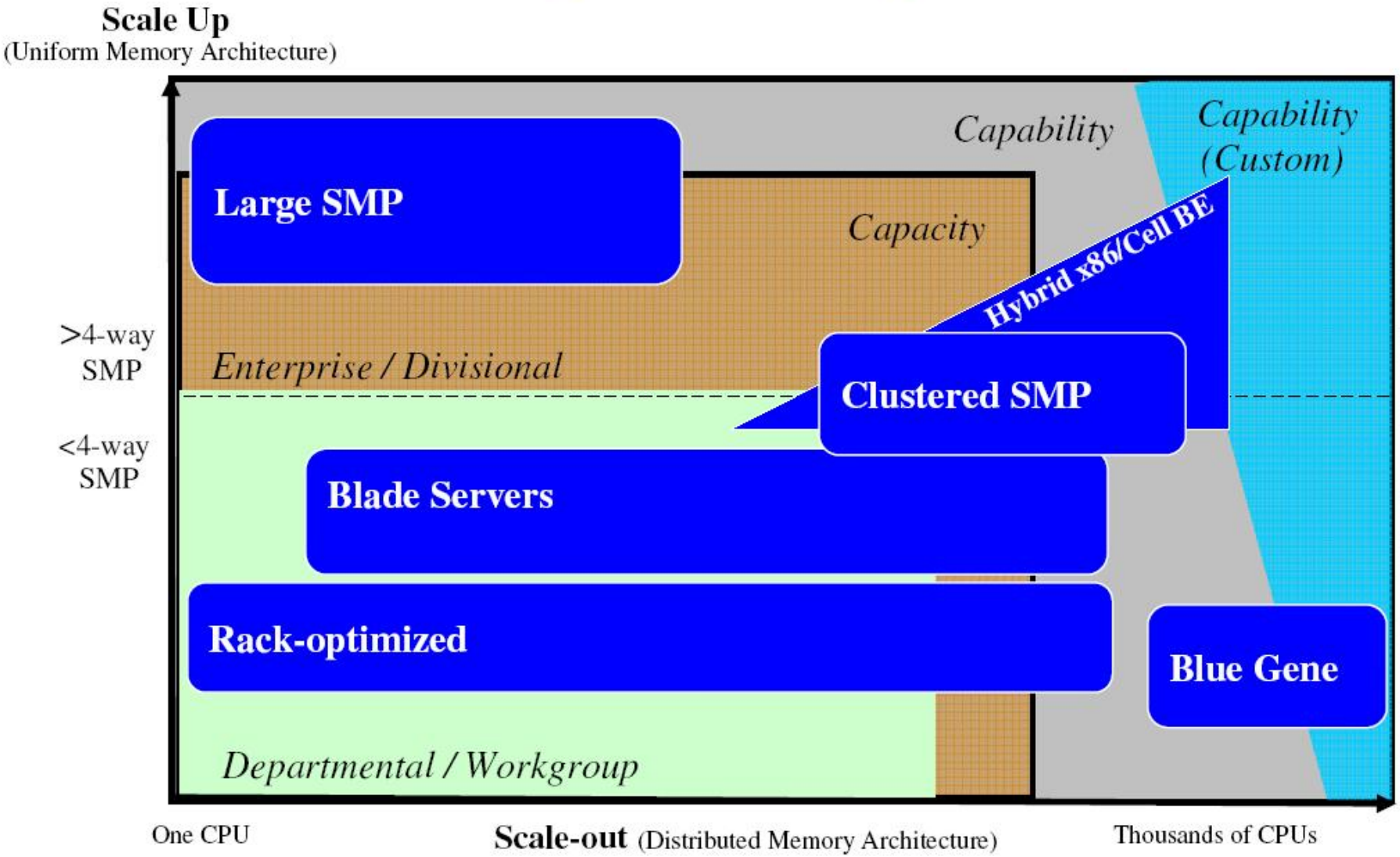


24 Orders Magnitude of Spatial and Temporal Range

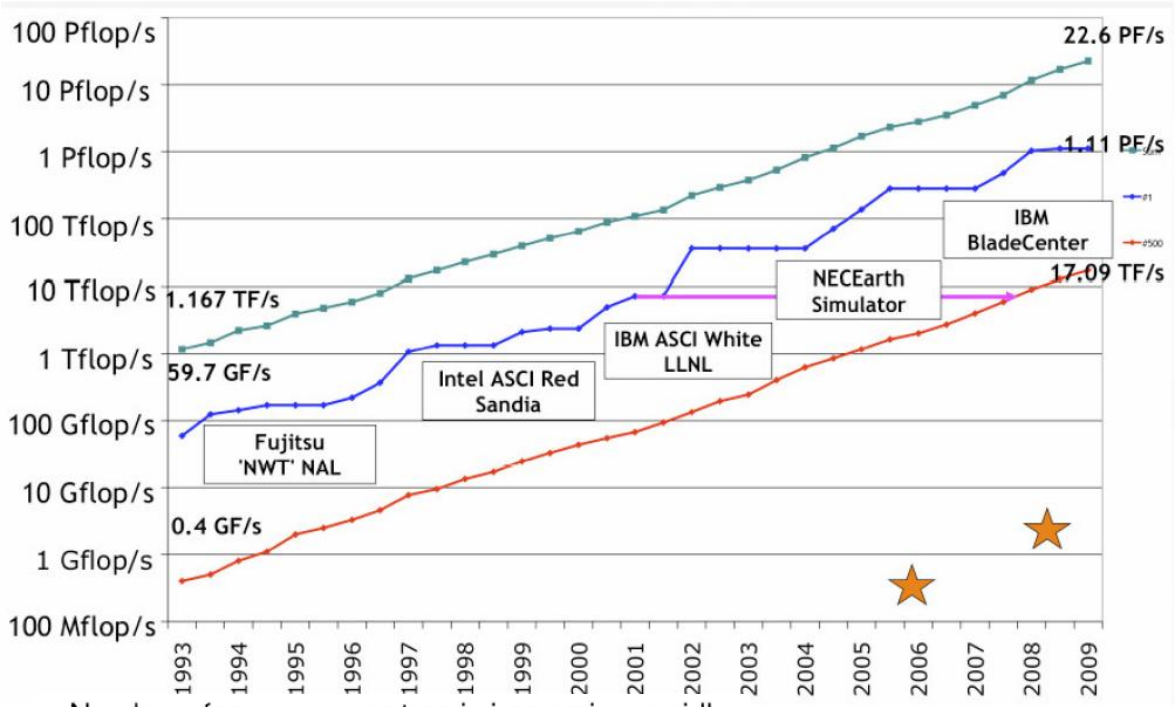



Model Integration Time
Goal
10-15 s
10-12
10-9
10-6
10-3
100 s
Enzyme function
Acetylcholinesterase
Cyclophilin A
Carbonic anhydrase
Catalase
Lactate dehydrogenase
Chymotrypsin
RuBisC
10-15 s
10-12
10-9
10-6
10-3
100 s
Protein dynamical events
Bond vibration
kBT/h
Elastic vibration of globular region
Rotation of side-chains
Protein breathing motions
10-15 s
10-12
10-9
10-6
10-3
100 s

Platform Positioning - Scalability

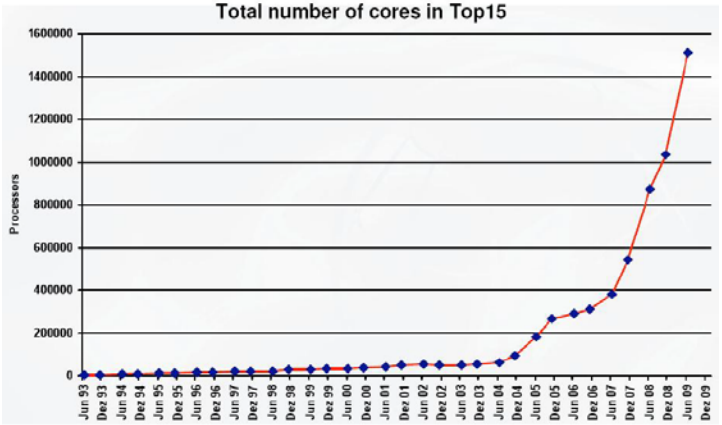


HPC Top500 Performance Evolution



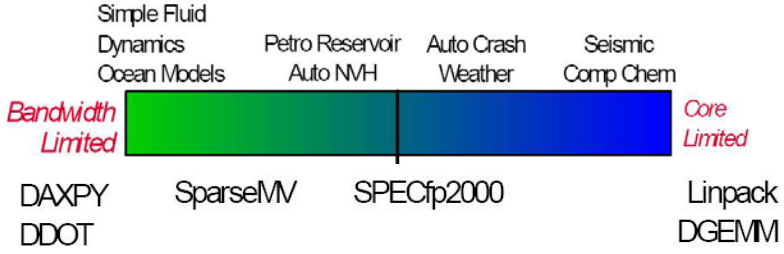
Number of cores per system is increasing rapidly

Petascale computers have tens to hundreds of thousands of processors

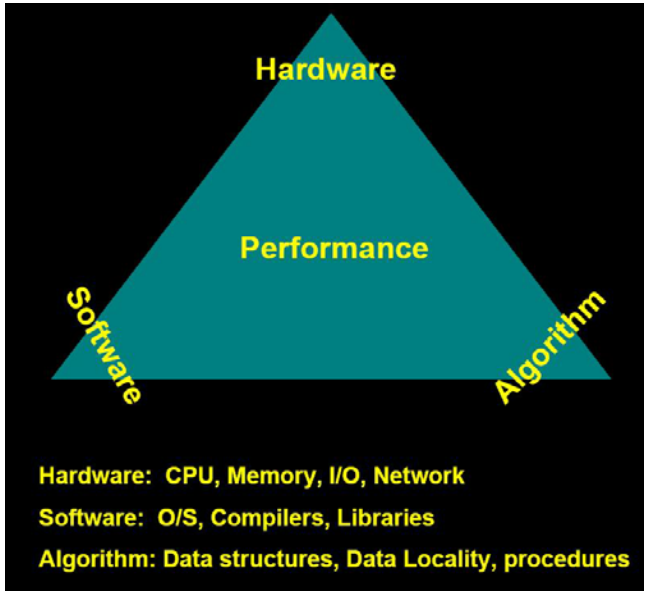


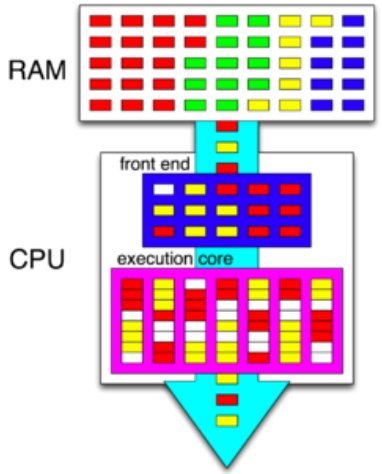
HPC Application Spectrum

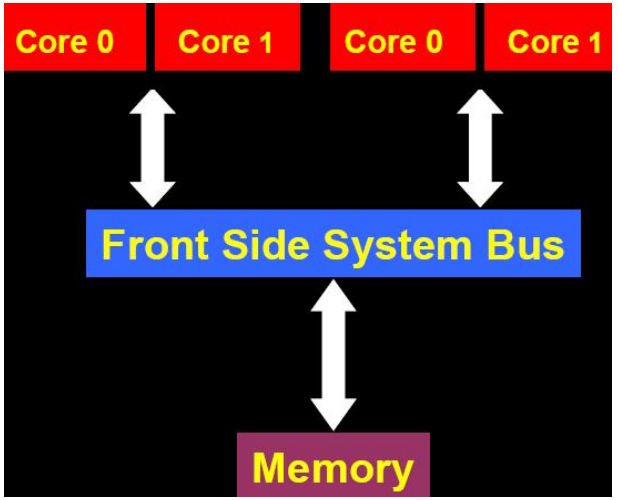
- **Bandwidth and processor compute capability assessed**
  - Applications span the spectrum
  - No single industry accepted metric exists



CPU design, scalability, memory, i/o, overall architectures

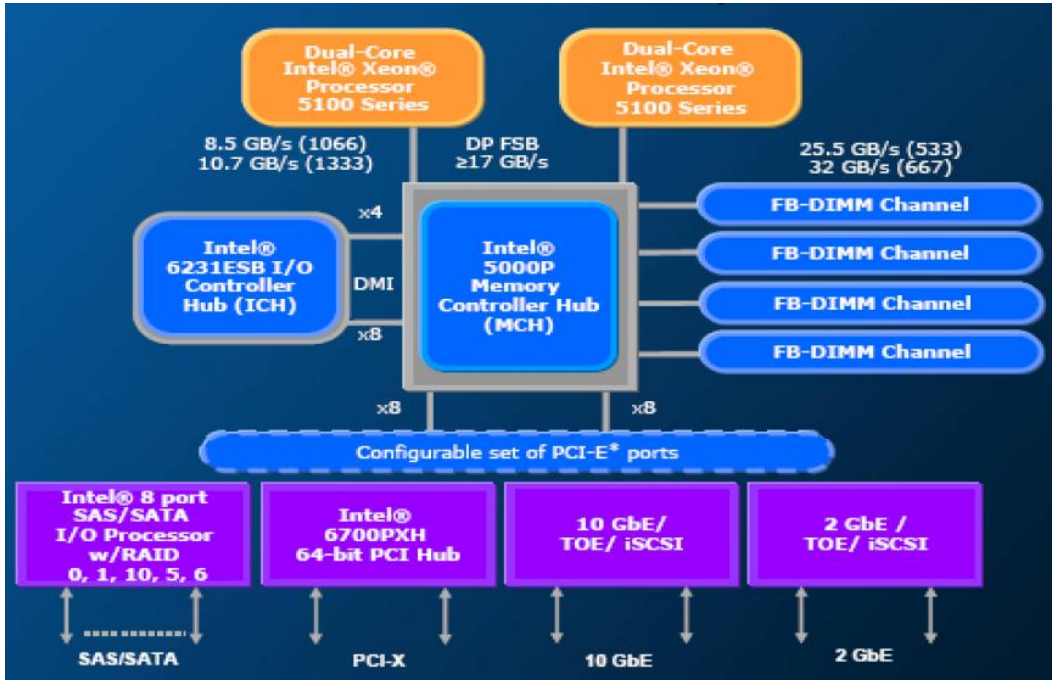






What is important to Software Performance As far as CPU is concerned?

- CPU Speed
- L1/L2 cache size
- L1/L2 Latency
- Execution rate (keeping the processor busy)
- Taking advantage of the Instruction Set
- Support for Threading



- Instructions per cycle
- Pipeline depth
- SMP and Hyperthreading
- Branch prediction
- Caches with hardware prefetch
- Out of order execution cores
- Instruction level parallelism
- Intra-register vectorization

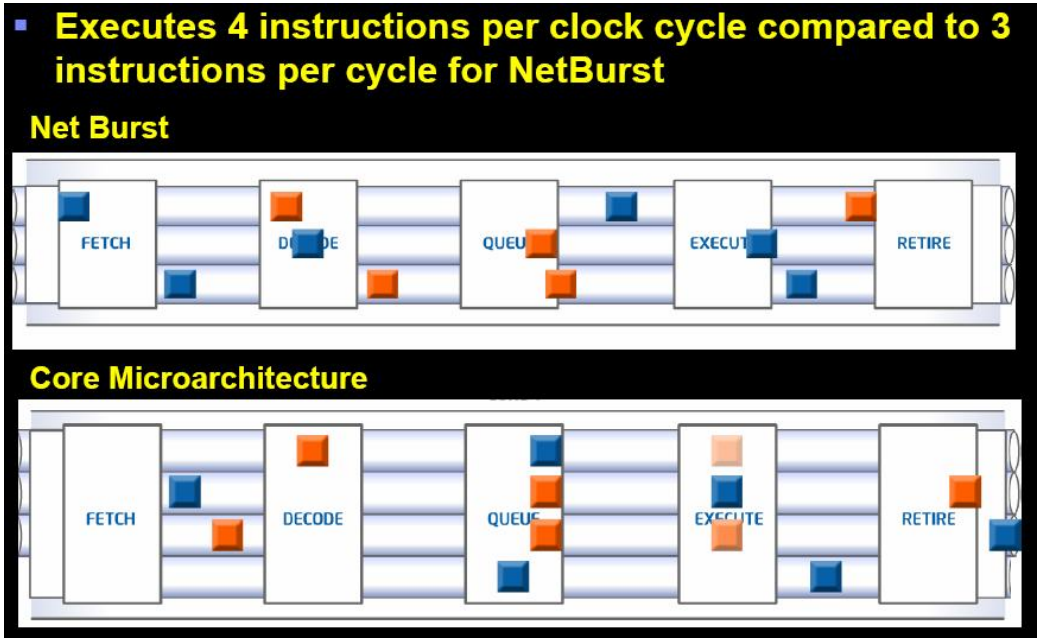
Hyper-Threading

To improve Single Core performance of

– Multi-threaded Application

– Multi-threaded Operating System

– Single-threaded App in Multi-tasking env



Many levels of parallelism

- Node
- Socket
- Chip
- Core
- Thread
- Register/SIMD
- Multiple instruction pipelines

MPI
Message Passing Interface

Outline

- Background
- Message Passing
- MPI
  - Group and Context
  - Communication Modes
  - Blocking/Non-blocking
  - Features
  - Programming / issues

Message Passing

- A process is a program counter and address space.
- Message passing is used for communication among processes.
- Inter-process communication:
  - Type:
    Synchronous / Asynchronous
  - Movement of data from one process's address space to another's

Synchronous Vs. Asynchronous

- A synchronous communication is not complete until the message has been received.

- An asynchronous communication completes as soon as the message is on the way.

What is message passing?

- Data transfer.
- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

What is MPI?

- A message-passing library specifications:
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks.
- Communication modes: *standard*, *synchronous*, *buffered,* and *ready*.
- Designed to permit the development of parallel software libraries.
- Designed to provide access to advanced parallel hardware for
  - End users
  - Library writers
  - Tool developers

Group and Context (cont.)

- Are two important and indivisible concepts of MPI.
- Group: is the set of processes that communicate with one another.
- Context: it is somehow similar to the frequency in radio communications.
- Communicator: is the central object for communication in MPI. Each communicator is associated with a group and a context.

Communication Modes

- Based on the type of send:
  - Synchronous: Completes once the acknowledgement is received by the sender.
  - Buffered send: completes immediately, unless if an error occurs.
  - Standard send: completes once the message has been sent, which may or may not imply that the message has arrived at its destination.
  - Ready send: completes immediately, if the receiver is ready for the message it will get it, otherwise the message is dropped silently.

Blocking vs. Non-Blocking

- Blocking, means the program will not continue until the communication is completed.

- Non-Blocking, means the program will continue, without waiting for the communication to be completed.

Features of MPI

- General
  - Communications combine context and group for message security.
  - Thread safety can’t be assumed for MPI programs.

Features that are NOT part of MPI

- Process Management
- Remote memory transfer
- Threads
- Virtual shared memory

Why to use MPI?

- MPI provides a powerful, efficient, and portable way to express parallel programs.
- MPI was explicitly designed to enable libraries which may eliminate the need for many users to learn (much of) MPI.
- Portable !!!!!!!!!!!!!!!!!!!!!!!!!
- Good way to learn about subtle issues in parallel computing

Parallelism can be found/exists at different granularities

- Instruction Level

Ex: add instruction executes with multiply instruction

Compiler good at finding this

- Thread Level

Ex: screen redraw function executes with recalculate in spreadsheet

Programmers good at finding this

- Process Level

Ex: Simulation job runs on same machines as spreadsheet

Users good at creating this

Thread Level Parallelism

Programmer generally makes TLP explicit

Compilers can extract threads in regular programs

```
for (i = 0; i < 200; i++)
for(j = 1; j < 20000; j++)
val[i,j] = val[i,j-1] + 1;

forall(i = 0; i < 200; i++)
for(j = 1; j < 20000; j++)
val[i,j] = val[i,j-1] + 1;
```

Thread Level Parallelism

Synchronization

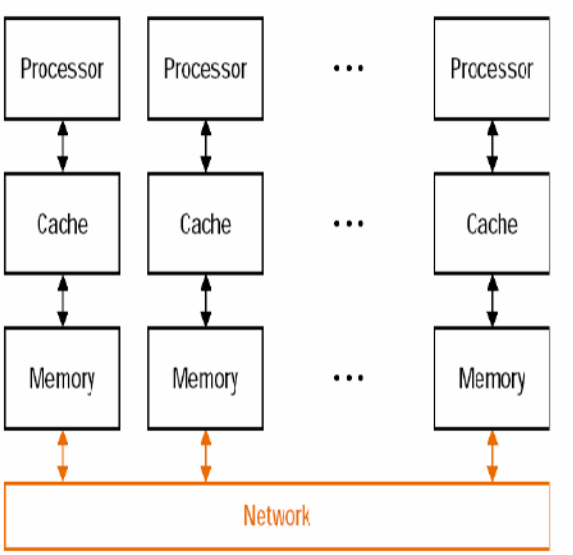
- Unlike in ILP, flow of data/dependences must be explicit

```
while(ptr = ptr->next)
sum += ptr->val;

while(ptr = ptr->next)
produce(ptr);
produce(NULL);

while(ptr = consume(ptr))
sum += ptr->val;
```

Communication and Synchronization… (order and flow)

MPI

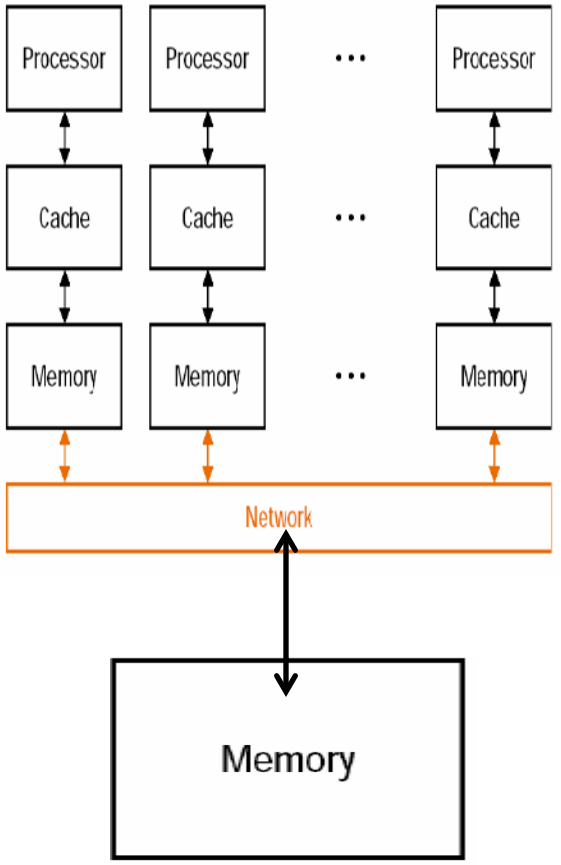
Multiple Processor Organization
Message Passing/Private Memory



- Threads communicate directly (send, receive)
- Scales relatively well
- No memory coherence problem (for the hardware at least)

OpenMP

Multiple Processor Organization
May exist on single chip



MPI Structure

- All MPI/C/C++ programs must include a header file mpi.h
- All MPI programs must call MPI INT as the first MPI call, to initialize themselves.
- Most MPI programs call MPI COMM SIZE to get the number of processes that are running
- Most MPI programs call MPI COMM RANK to determine their rank, which is a number between 0 and size-1.
- Conditional process and general message passing can take place. For example, using the calls MPI SEND and MPI RECV.
- All MPI programs must call MPI FINALIZE as the last call to an MPI library routine.

MPI – Point to Consider

- Data types
- Communication - point-to-point and collective
- Timing
- Grouping data for communications
- Communicators and Topologies
- I/O in parallel
- Debugging parallel program
- Performance

MPI Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with

Process 0	Process 1
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

- This is called “unsafe” because it depends on the availability of system buffers

MPI – Deadlock Solutions

- Order the operations more carefully:

Process 0	Process 1
Send(1)	**Recv(0)**
Recv(1)	**Send(0)**

- Use non-blocking operations:

Process 0	Process 1
Isend(1)	**Isend(0)**
Irecv(1)	**Irecv(0)**
Waitall	**Waitall**

RECAP

- MPI Communications
- Most important Part of MPI
- RDMA v/s TCP/IP

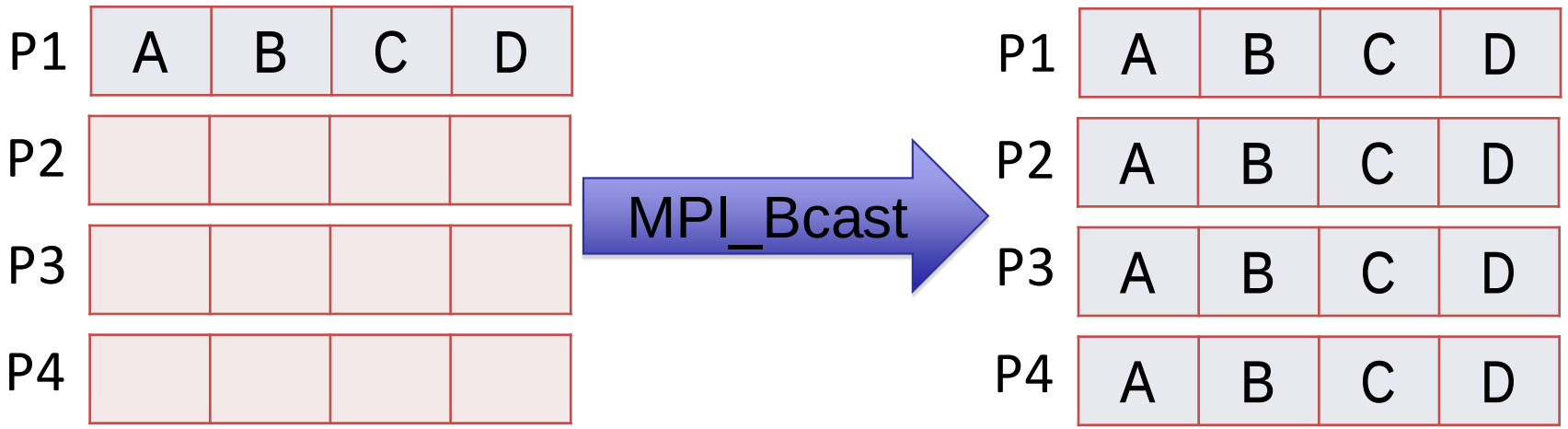
Collective communications

- 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 (e.g. MPI_Barrier)

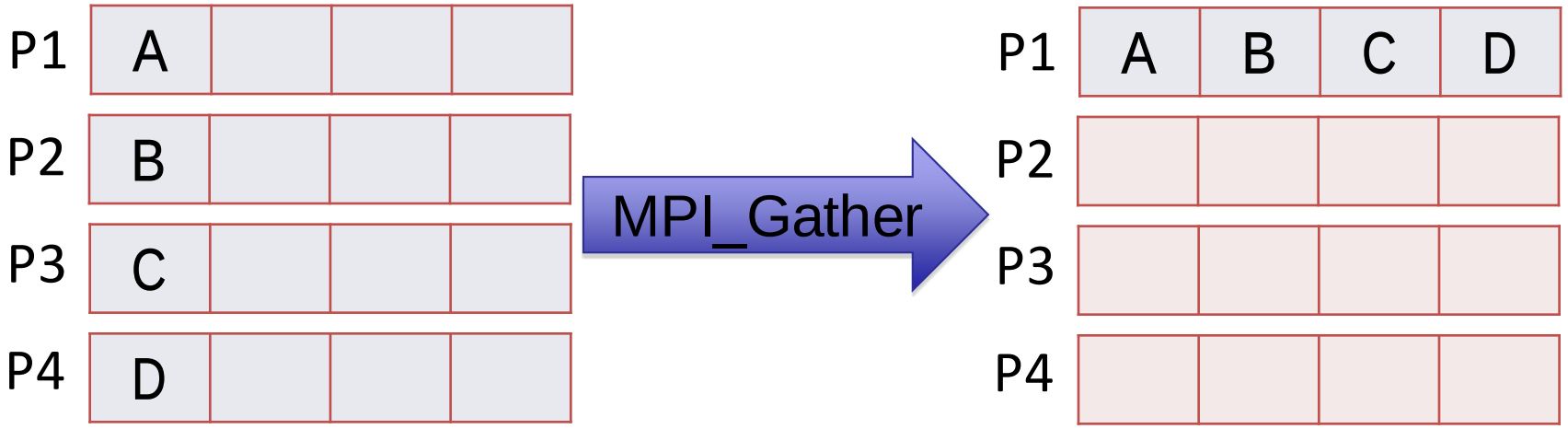
Broadcast

- int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
  - One process (root) sends data to all the other processes in the same communicator
  - Must be called by all the processes with the same arguments



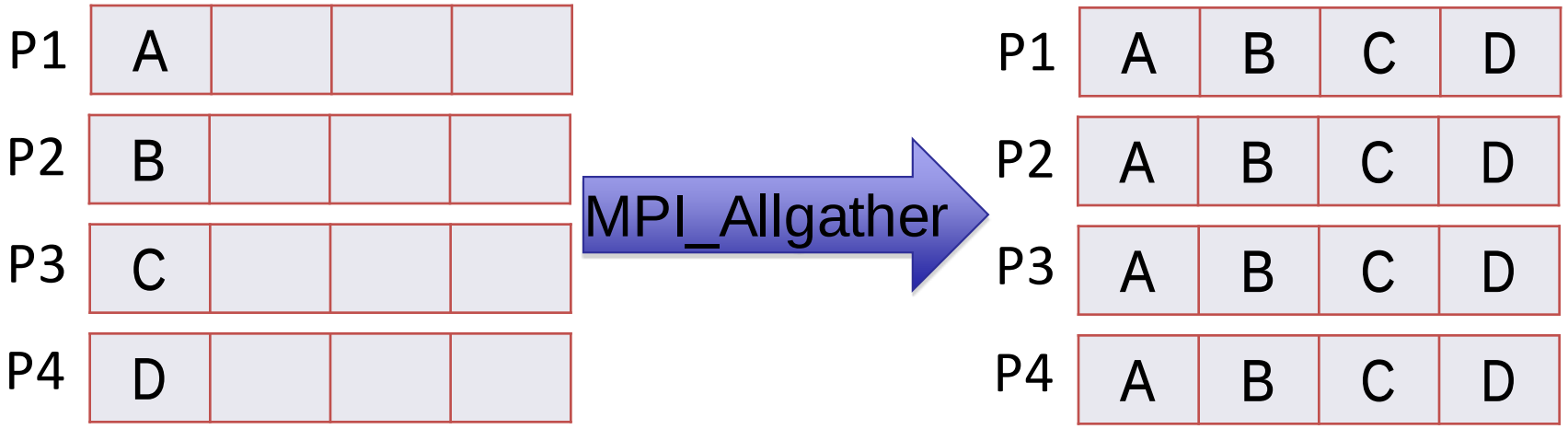
Gather

- int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
  - One process (root) collects data to all the other processes in the same communicator
  - Must be called by all the processes with the same arguments



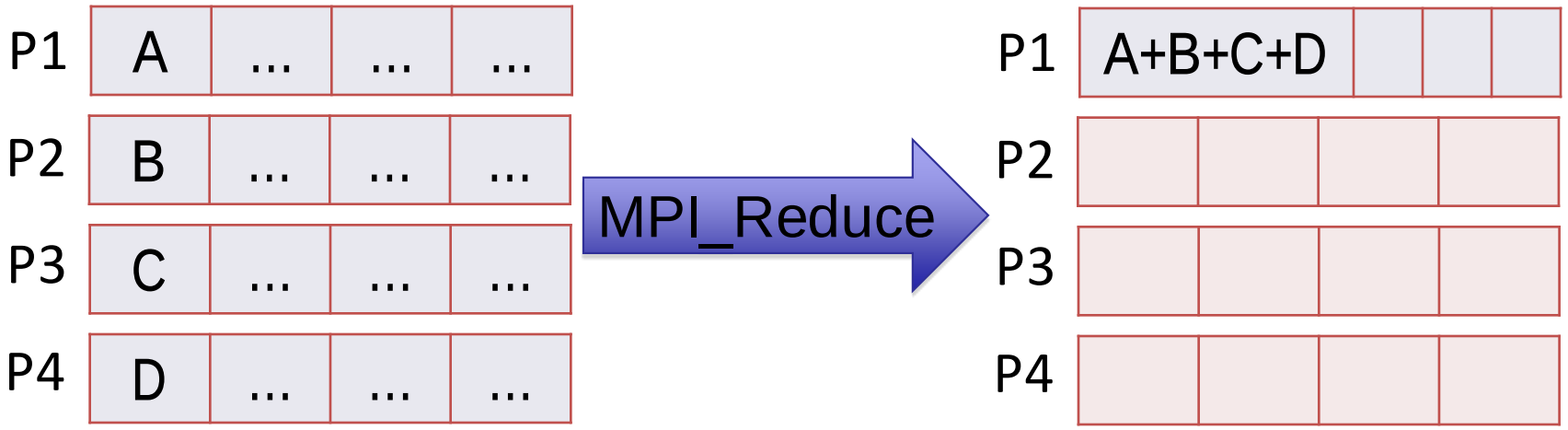
Gather to All

- int MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)
  - All the processes collects data to all the other processes in the same communicator
  - Must be called by all the processes with the same arguments



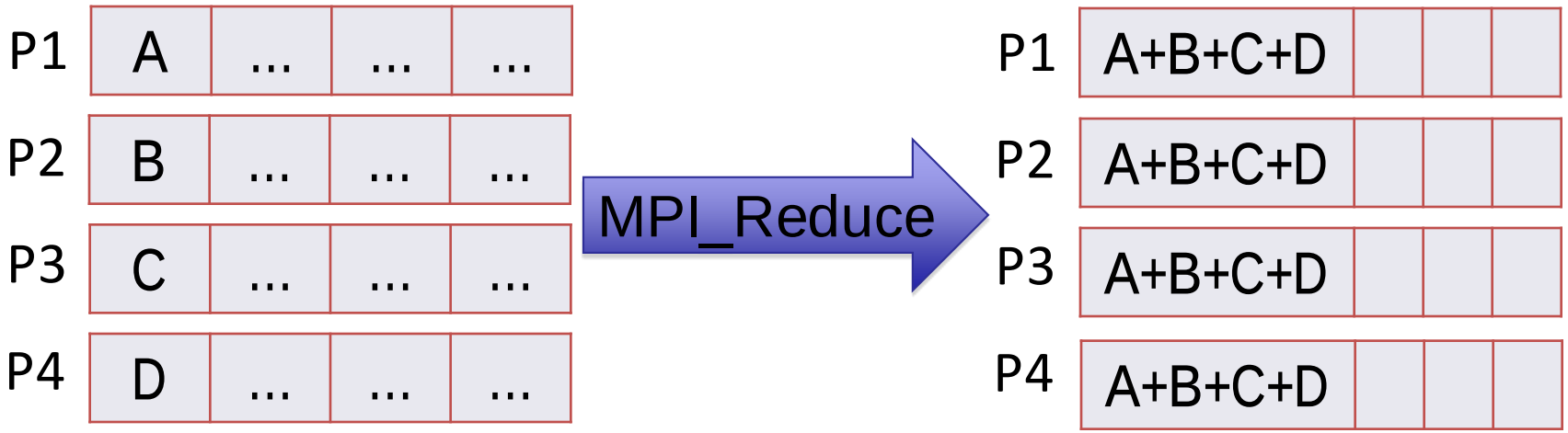
Reduction

- int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - One process (root) collects data to all the other processes in the same communicator, and performs an operation on the data
  - MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
  - MPI_Op_create(): User defined operator



Reduction to All

- int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  - All the processes collect data to all the other processes in the same communicator, and perform an operation on the data
  - MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
  - MPI_Op_create(): User defined operator



Synchronization

- int MPI_Barrier(MPI_Comm comm)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
   int rank, nprocs;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Barrier(MPI_COMM_WORLD);
   printf("Hello, world.  I am %d of %d\n", rank, nprocs);
   MPI_Finalize();
   return 0;
}
```

Some concepts

- The default communicator is the `MPI_COMM_WORLD`
- A process is identified by its rank in the group associated with a communicator.

Data Types

- The data message which is sent or received is described by a triple (address, count, datatype).
- The following data types are supported by MPI:
  - Predefined data types that are corresponding to data types from the programming language.
  - Arrays.
  - Sub blocks of a matrix
  - User defined data structure.
  - A set of predefined data types

Basic MPI types

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

Why defining the data types during the send of a message?

Because communications take place between heterogeneous machines. Which may have different data representation and length in the memory.

MPI blocking send

```
MPI_SEND(void *start, int
  count,MPI_DATATYPE datatype, int dest,
  int tag, MPI_COMM comm)
```

- The message buffer is described by (start, count, datatype).
- dest is the rank of the target process in the defined communicator.
- tag is the message identification number.

MPI blocking receive

```
MPI_RECV(void *start, int count,
MPI_DATATYPE datatype, int source, int tag,
MPI_COMM comm, MPI_STATUS *status)
```

- Source is the rank of the sender in the communicator.
- The receiver can specify a wildcard value for souce (MPI_ANY_SOURCE) and/or a wildcard value for tag (MPI_ANY_TAG), indicating that any source and/or tag are acceptable
- Status is used for exrtra information about the received message if a wildcard receive mode is used.
- If the count of the message received is less than or equal to that described by the MPI receive command, then the message is successfully received. Else it is considered as a buffer overflow error.

MPI_STATUS

- Status is a data structure
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(…, MPI_ANY_SOURCE, MPI_ANY_TAG, …,
  &status)
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recvd_count);
```

More info

- A receive operation may accept messages from an arbitrary sender, but a send operation must specify a unique receiver.

- Source equals destination is allowed, that is, a process can send a message to itself.

Why MPI is simple?

- Many parallel programs can be written using just these six functions, only two of which are non-trivial;
  - MPI_INIT
  - MPI_FINALIZE
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
  - MPI_SEND
  - MPI_RECV

Non-Blocking Send and Receive

```
MPI_ISEND(buf, count, datatype, dest, tag, comm,
   request)
MPI_IRECV(buf, count, datatype, dest, tag, comm,
   request)
```

- request is a request handle which can be used to query the status of the communication or wait for its completion.

Non-Blocking Send and Receive (Cont.)

- A non-blocking send call indicates that the system may start copying data out of the send buffer. The sender must not access any part of the send buffer after a non-blocking send operation is posted, until the complete-send returns.
- A non-blocking receive indicates that the system may start writing data into the receive buffer. The receiver must not access any part of the receive buffer after a non-blocking receive operation is posted, until the complete-receive returns.

Non-Blocking Send and Receive (Cont.)

```
MPI_WAIT (request, status)
MPI_TEST (request, flag, status)
```

- The MPI_WAIT will block your program until the non-blocking send/receive with the desired request is done.

- The MPI_TEST is simply queried to see if the communication has completed and the result of the query (TRUE or FALSE) is returned immediately in flag.

Deadlocks in blocking operations

- What happens with

Process 0	Process 1
Send(1)	Send(0)
Recv(1)	Recv(0)

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space(through a receive)
- This is called “unsafe” because it depends on the availability of system buffers.

Some solutions to the “unsafe” problem

- Order the operations more carefully

Process 0	Process 1
Send(1)	Recv(0)
Recv(1)	Send(0)

Use non-blocking operations:

Process 0		Process 1
ISend(1)	ISend(0)	
IRecv(1)	IRecv(0)	
Waitall		Waitall

Introduction to collective operations in MPI

- Collective operations are called by all processes in a communicator
- MPI_Bcast distributes data from one process(the root) to all others in a communicator.

  Syntax:

  ```
  MPI_Bcast(void *message, int count, MPI_Datatype
    datatype, int root, MPI_Comm comm)
  ```

- MPI_Reduce combines data from all processes in communicator or and returns it to one process

  Syntax:

  ```
  MPI_Reduce(void *message, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
  comm)
  ```

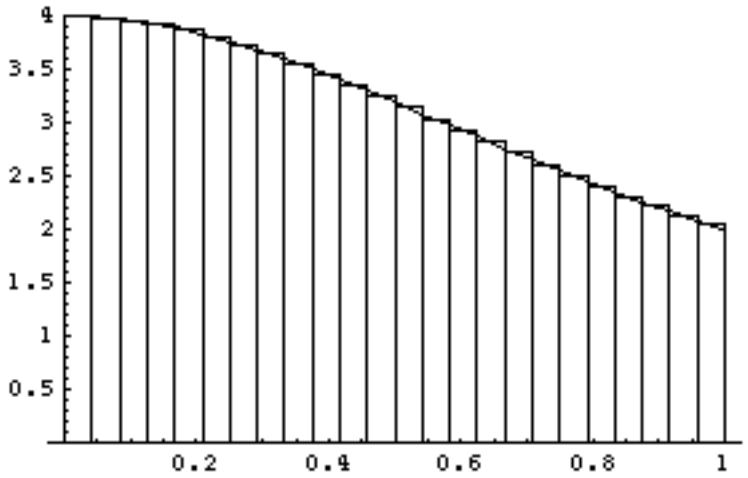
- In many numerical algorithm, send/receive can be replaced by Bcast/Reduce, improving both simplicity and efficiency.

Collective Operations

MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC

Example: Compute PI (0)

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



Example: Compute PI (1)

```
#include “mpi.h”
#include <math.h>

int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, I, rc;
   double PI25DT = 3.14159265358979323846264 3;
   double mypi, pi, h, sum, x, a;
   MPI_INIT(&argc, &argv);
   MPI_COMM_SIZE(MPI_COMM_WORLD, &numprocs);
   MPI_COMM_RANK(MPI_COMM_WORLD, &myid);
   while (!done)
   {
       if (myid == 0)
       {
       printf(“Enter the number of intervals: (0 quits) “);
              scanf(“%d”, &n);
       }
       MPI_BCAST(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
       if (n == 0)
   }
```

Example: Compute PI (2)

```
        h = 1.0 / (double)n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs)
        {
                x = h * ((double)i – 0.5);
                sum += 4.0 / (1.0 + x * x);
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

        if (myid == 0) printf(“pi is approximately %.16f, Error
is %.16f\n”, pi, fabs(pi – PI25DT));

        MPI_Finalize();
        return 0;
}
```

When to use MPI

- Portability and Performance
- Irregular data structure
- Building tools for others
- Need to manage memory on a per processor basis

Examples

- Simple

```
#include "main.h"

using namespace std;

int main(int argc, char* argv[]){
    int mytid, numprocs;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&mytid);

    char name[100];
    gethostname(name, sizeof(name));

    if(mytid>0){
     funkywork();
    }

    cout << "Hello, " << mytid <<  " and " << name << " say hi in a  C++ statement \n";

    MPI_Finalize();

 }
```

- # Serial v/s MPI parallel

```
#include<stdio.h>

#define NRA 5000                 /* number of rows in matrix A */
#define NCA 1000                 /* number of columns in matrix A */
#define NCB 100                  /* number of columns in matrix B */
main()
{
int   i, j, k;              /* misc */
double a[NRA][NCA],         /* matrix A to be multiplied */
     b[NCA][NCB],           /* matrix B to be multiplied */
     c[NRA][NCB];           /* result matrix C */

  /* Initialize A, B, and C matrices */
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;
  for(i=0;i<NRA;i++)
    for(j=0;j<NCB;j++)
      c[i][j] = 0.0;

  /* Perform matrix multiply */
  for(i=0;i<NRA;i++)
    for(j=0;j<NCB;j++)
      for(k=0;k<NCA;k++)
        c[i][j]+= a[i][k] * b[k][j];

  /* Okay, it's a trivial program */
  printf("Here is the result matrix\n");
  for (i=0; i<NRA; i++)
  {
    printf("\n");
    for (j=0; j<NCB; j++)
      printf("%6.2f   ", c[i][j]);
  }
  printf ("\n");
}
```

```
#include "mpi.h"
#include <stdio.h>
#define NRA 5000                 /* number of rows in matrix A */
#define NCA 1000                 /* number of columns in matrix A */
#define NCB 100                  /* number of columns in matrix B */
#define MASTER 0                 /* taskid of first task */
#define FROM_MASTER 1            /* setting a message type */
#define FROM_WORKER 2            /* setting a message type */

int main(argc,argv)
int argc;
char *argv[];
{
int     numtasks,               /* number of tasks in partition */
      taskid,                   /* a task identifier */
      numworkers,               /* number of worker tasks */
      source,                   /* task id of message source */
      dest,                     /* task id of message destination */
      mtype,                    /* message type */
      rows,                     /* rows of matrix A sent to each worker */
      averow, extra, offset,    /* used to determine rows sent to each
worker */
      i, j, k, rc;              /* misc */
double  a[NRA][NCA],            /* matrix A to be multiplied */
      b[NCA][NCB],              /* matrix B to be multiplied */
      c[NRA][NCB];              /* result matrix C */
MPI_Status status;

  rc = MPI_Init(&argc,&argv);
  rc|= MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  rc|= MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
  if (rc != 0)
    printf ("error initializing MPI and obtaining task ID
information\n");
  else
    printf ("task ID = %d\n", taskid);
  numworkers = numtasks-1;
/************************** master task
************************************/
}
```

```
if (taskid == MASTER)
{
  printf("Number of worker tasks = %d\n",numworkers);
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;

  /* send matrix data to the worker tasks */
  averow = NRA/numworkers;
  extra = NRA%numworkers;
  offset = 0;
  mtype = FROM_MASTER;
  for (dest=1; dest<=numworkers; dest++)
  {
    rows = (dest <= extra) ? averow+1 : averow;
    printf("   sending %d rows to task %d\n",rows,dest);
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
             MPI_COMM_WORLD);
    MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    offset = offset + rows;
  }

  /* wait for results from all worker tasks */
  mtype = FROM_WORKER;
  for (i=1; i<=numworkers; i++)
  {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
  }

  /* print results */
  printf("Here is the result matrix\n");
  for (i=0; i<NRA; i++)
  {
    printf("\n");
    for (j=0; j<NCB; j++)
      printf("%6.2f   ", c[i][j]);
  }
  printf ("\n");
}
```

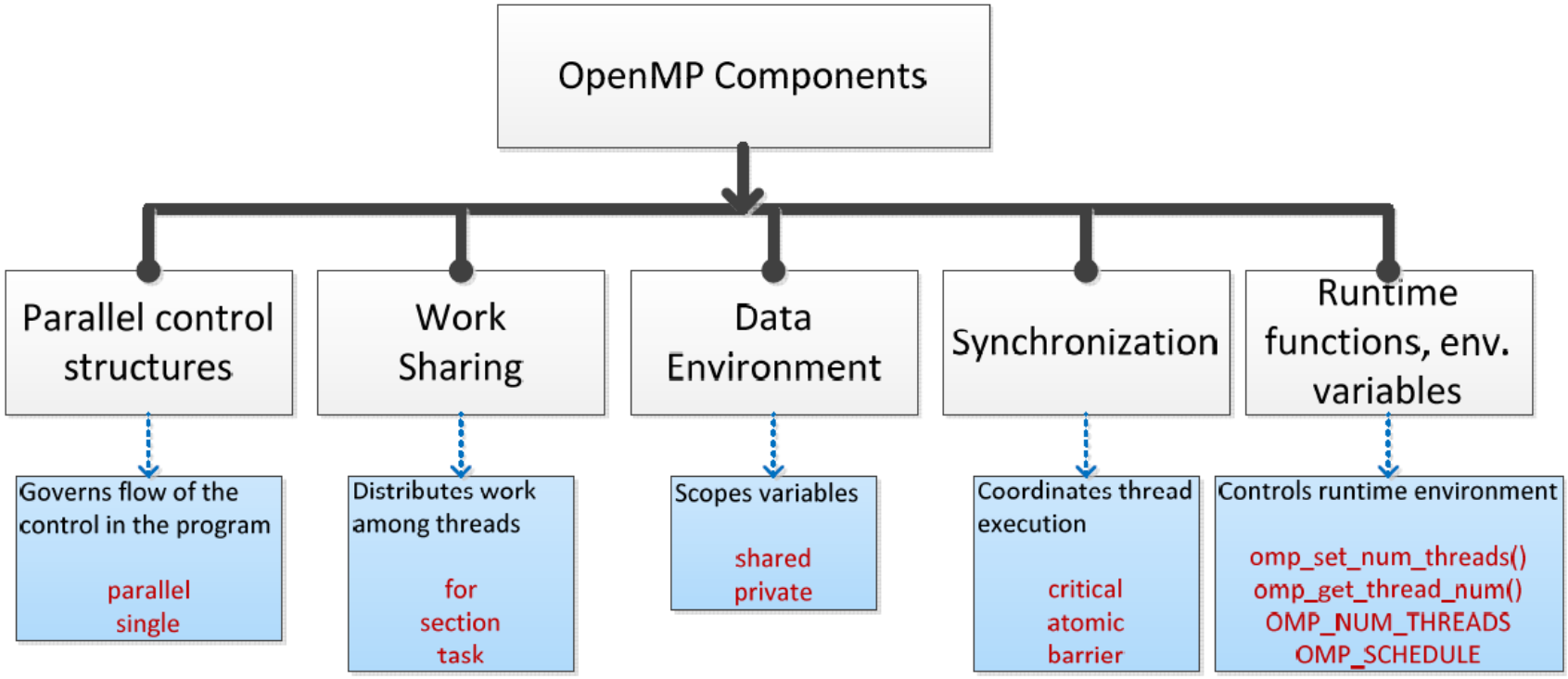
```
/**************************  worker task ************************************/
  if (taskid > MASTER)
  {
    mtype = FROM_MASTER;
    MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);

    for (k=0; k<NCB; k++)
      for (i=0; i<rows; i++)
      {
        c[i][k] = 0.0;
        for (j=0; j<NCA; j++)
          c[i][k] = c[i][k] + a[i][j] * b[j][k];
      }
    mtype = FROM_WORKER;
    MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
  }
  MPI_Finalize();
```

LETS RUN THIS ON DISCOVERY CLUSTER

OpenMP

- v2.5, v3.0 and v3.1 standards
- gcc 4.4 and higher implements v3.0, and 4.7 v3.1
- Syntax:**#pragma omp *directive-name [clause[[,] clause]...] new-line***
- Execution Model: OpenMP API uses the fork-join model of parallel execution
- Memory Model: OpenMP has a relaxed-consistency, shared-memory model
- Each thread has temporary view of memory and thread private memory
- OpenMP flush - enforces consistency between temporary view and memory

  http://gcc.gnu.org/onlinedocs/libgomp/index.html#toc_Runtime-Library-Routines


OpenMP Components
Parallel control structures
Work Sharing
Data Environment
Synchronization
Runtime functions, env. variables
Governs flow of the control in the program
parallel
single
Distributes work among threads
for
section
task
Scopes variables
shared
private
Coordinates thread execution
critical
atomic
barrier
Controls runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

- Application Programmer Interface (API) is combination of
  - Directives
    - Example: `#pragma omp task`
  - Runtime library routines
    - Example: `int omp_get_thread_num(void)`
  - Environment variables
    - Example: `setenv OMP_SCHEDULE "guided, 4"`

- Directives (or Pragmas) used to
  - Express/Define parallelism (flow control)
    - Example: `#pragma omp parallel for`
  - Specify data sharing among threads (communication)
    - Example: `#pragma omp parallel for private(x,y)`
  - Synchronization (coordination or interaction)
    - Example: `#pragma omp barrier`

Summary of Run-Time Library OpenMP Routines

1. omp_set_num_threads
2. omp_get_num_threads
3. omp_get_max_threads
4. omp_get_thread_num
5. omp_get_thread_limit
6. omp_get_num_procs
7. omp_in_parallel
8. omp_set_dynamic
9. omp_get_dynamic
10. omp_set_nested
11. omp_get_nested
12. omp_set_schedule
13. omp_get_schedule
14. omp_set_max_active_levels
15. omp_get_max_active_levels
16. omp_get_level
17. omp_get_ancestor_thread_num
18. omp_get_team_size
19. omp_get_active_level
20. omp_init_lock
21. omp_destroy_lock
22. omp_set_lock
23. omp_unset_lock
24. omp_test_lock
25. omp_init_nest_lock
26. omp_destroy_nest_lock
27. omp_set_nest_lock
28. omp_unset_nest_lock
29. omp_test_nest_lock
30. omp_get_wtime
31. omp_get_wtick

OMP_SCHEDULE

- Example: **`setenv OMP_SCHEDULE "guided, 4"`**

OMP_NUM_THREADS

- Sets the maximum number of threads to use during execution.
- 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
- Example: **`setenv OMP_DYNAMIC TRUE`**

OMP_NESTED

- Enables or disables nested parallelism. Valid values are TRUE or FALSE
- Example: **`setenv OMP_NESTED TRUE`**

OMP_STACKSIZE

- Controls the size [in KB] of the stack for created (non-Master) threads.

OMP_WAIT_POLICY

- Provides hint to an OpenMP implementation about desired behavior of waiting threads

OMP_MAX_ACTIVE_LEVELS

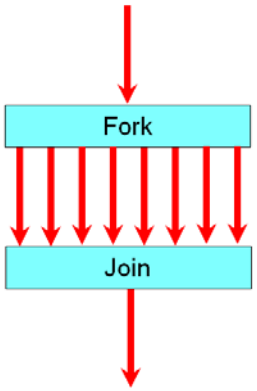
- Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. Example:
- `setenv OMP_MAX_ACTIVE_LEVELS 2`

OMP_THREAD_LIMIT

- Sets the number of OpenMP threads to use for the whole OpenMP program Example:
- `setenv OMP_THREAD_LIMIT 8`

Serial regions by default, annotate to create *parallel regions*

- Generic parallel regions
- Parallelized loops
- Sectioned parallel regions

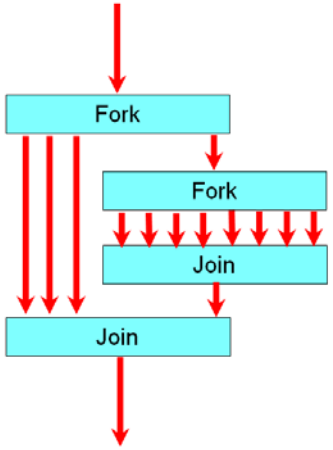


Thread-like Fork/Join model

- Arbitrary number of *logical* thread creation/ destruction events

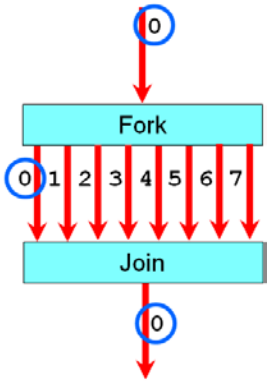
Fork/Join can be nested

- Nesting complication handled "automagically" at compile-time
- Independent of the number of threads actually running



Master Thread

- Thread with ID=0
- Only thread that exists in sequential regions
- Depending on implementation, may have special purpose inside parallel regions
- Some special directives affect only the master thread (like `master`)

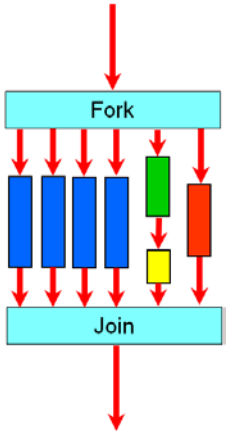


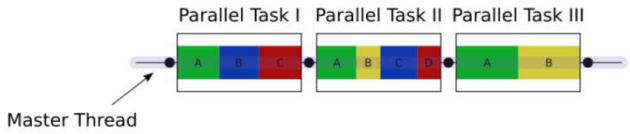
Data parallelism

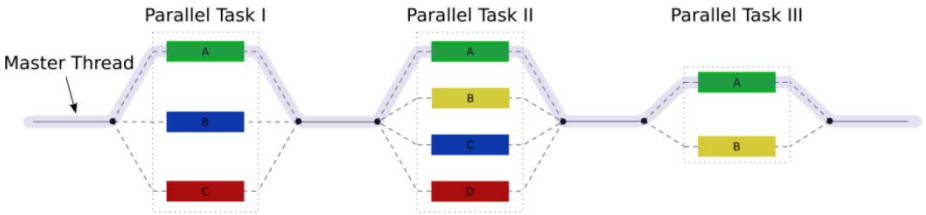
- Threads perform similar functions, guided by thread identifier

Control parallelism

- Threads perform differing functions
  - » One thread for I/O, one for computation, etc…







OpenMP Loop Scheduling

schedule clause determines how loop iterations are divided among the thread team

- static([chunk]) divides iterations statically between threads
  - » Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
  - » Default [chunk] is ceil(# iterations / # threads)
- dynamic([chunk]) allocates [chunk] iterations per thread, allocating an additional [chunk] iterations when a thread finishes
  - » Forms a logical work queue, consisting of all loop iterations
  - » Default [chunk] is 1
- guided([chunk]) allocates dynamically, but [chunk] is exponentially reduced with each allocation

```
#pragma omp parallel for \
 schedule(static)
for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
// Static Scheduling

int chunk = 16/T;
int base = tid * chunk;
int bound = (tid+1)*chunk;

for( i=base; i<bound; i++ )
{
  doIteration(i);
}

Barrier();
```

```
#pragma omp parallel for \
 schedule(dynamic)
for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
// Dynamic Scheduling

int current_i;

while( workLeftToDo() )
{
  current_i = getNextIter();
  doIteration(i);
}

Barrier();
```

OpenMP Data Sharing

Parallel programs often employ two types of data

- **Shared data, visible to all threads, similarly named**
- **Private data, visible to a single thread (often stack-allocated)**

PThreads:

- **Global-scoped variables are shared**
- **Stack-allocated variables are private**

OpenMP:

- **shared variables are shared**
- **private variables are private**

OpenMP Synchronization

- OpenMP Critical Sections
  - » Named or unnamed
  - » No *explicit* locks
- Barrier directives
- Explicit Lock functions
  - » When all else fails – may require flush directive
- Single-thread regions *within* parallel regions
  - » master, single directives

```
#pragma omp critical
{
  /* Critical code here */
}
#pragma omp barrier

omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );
#pragma omp single
{
  /* Only executed once */
}
```

OpenMP Program to Compute PI

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
int nthreads, tid;
int i, INTERVALS;
double n_1, x, pi = 0.0;

INTERVALS=128000;
/* Fork a team of threads giving them their own
      copies of variables */
#pragma omp parallel private(nthreads, tid)
 {
 /* Obtain thread number */
 tid = omp_get_thread_num();
 printf("Hello 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 disband */

 n_1 = 1.0 / (double)INTERVALS;

/* Parallel loop with reduction for calculating PI */
#pragma omp parallel for private(i,x) shared
(n_1,INTERVALS) reduction(+:pi)
 for (i = 0; i < INTERVALS; i++)
   {
   x = n_1 * ((double)i  - 0.5);
   pi += 4.0 / (1.0 + x * x);
   }
   pi *= n_1;
   printf ("Pi = %.12lf\n", pi);
}
```

LET’S RUN AN EXAMPLE OF MATRIX MULTIPLICATION IN SERIAL AND WITH OpenMP PARALLELIZATION ON DISCOVERY CLUSTER

Hybrid OpenMP + MPI

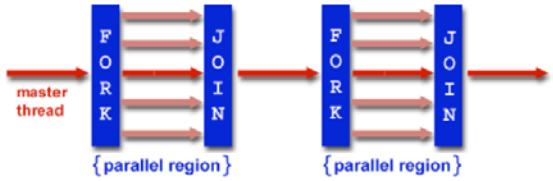
```
#include <mpi.h>      /* MPI Library                                  */
#include <omp.h>      /* OpenMP Library                               */
#include <stdio.h>    /* printf()                                     */
#include <stdlib.h>   /* EXIT_SUCCESS                                 */

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

   /* Parameters of MPI.                                              */
   int M_N;                  /* number of MPI ranks                  */
   int M_ID;                 /* MPI rank ID                          */
   int rtn_val;              /* return value                         */
   char name[128];           /* MPI_MAX_PROCESSOR_NAME == 128        */
   int namelen;

   /* Parameters of OpenMP.                                           */
   int O_P;                  /* number of OpenMP processors          */
   int O_T;                  /* number of OpenMP threads             */
   int O_ID;                 /* OpenMP thread ID                     */

   /* Initialize MPI.                                                 */
   /* Construct the default communicator MPI_COMM_WORLD.              */
   rtn_val = MPI_Init(&argc,&argv);

   /* Get a few MPI parameters.                                       */
   rtn_val = MPI_Comm_size(MPI_COMM_WORLD,&M_N);   /* get number of MPI ranks  */
   rtn_val = MPI_Comm_rank(MPI_COMM_WORLD,&M_ID);  /* get MPI rank ID          */
   MPI_Get_processor_name(name,&namelen);
   printf("name:%s  M_ID:%d  M_N:%d\n", name,M_ID,M_N);

   /* Get a few OpenMP parameters.                                    */
   O_P  = omp_get_num_procs();       /* get number of OpenMP processors  */
   O_T  = omp_get_num_threads();     /* get number of OpenMP threads     */
   O_ID = omp_get_thread_num();      /* get OpenMP thread ID             */
   printf("name:%s  M_ID:%d  O_ID:%d  O_P:%d  O_T:%d\n", name,M_ID,O_ID,O_P,O_T);

   /* PARALLEL REGION                                                 */
   /* Thread IDs range from 0 through omp_get_num_threads()-1.        */
   /* We execute identical code in all threads (data parallelization). */
   #pragma omp parallel private(O_ID)
   {
   O_ID = omp_get_thread_num();      /* get OpenMP thread ID            */
   MPI_Get_processor_name(name,&namelen);
   printf("parallel region:     name:%s M_ID=%d O_ID=%d\n", name,M_ID,O_ID);
   }

   /* Terminate MPI.                                                  */
   rtn_val = MPI_Finalize();

   /* Exit master thread.                                             */
   printf("name:%s M_ID:%d O_ID:%d   Exits\n", name,M_ID,O_ID);
   return EXIT_SUCCESS;
}
```

OpenMP Summary

Shared memory, thread-based parallelism

Explicit parallelism (relies on you specifying parallel regions)

Fork/join model



Industry-standard shared memory programming model

- First version released in 1997

OpenMP Architecture Review Board (ARB) determines updates to standard

- The final specification of Version 3.1 released in July of 2011 (minor update)

OpenMP provides small yet versatile programming model

- This model serves as the inspiration for the OpenACC effort to standardizing approaches that can factor in the presence of a GPU accelerator

Not at all intrusive, very straightforward to parallelize existing code

- Good efficiency gains achieved by using parallel regions in an existing code

Work-sharing constructs: `for`, `section`, `task` enable parallelization of computationally intensive portions of program

Parallelize small parts of application, one at a time (beginning with most time-critical parts)

Can implement complex algorithms

Code size grows only modestly

Expression of parallelism flows clearly, code is easy to read

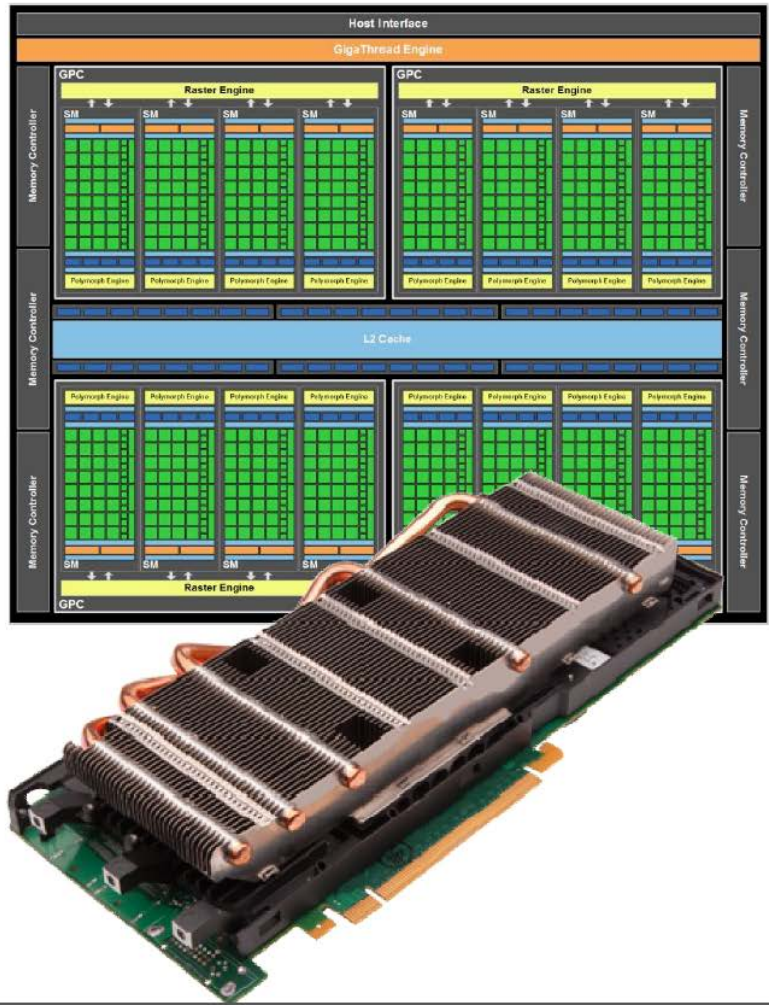
OpenMP threads are heavy

- Very good for handling parallel tasks
- Not particularly remarkable at handling fine grain data parallelism (vector architectures excel here)

GPGPU

NVIDIA GPUs

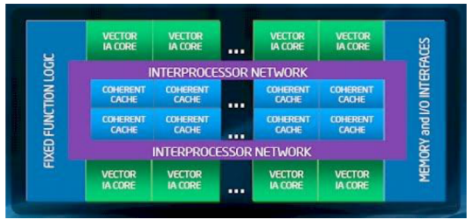
- Supports CUDA and OpenCL
- Fermi (Tesla version)
  - Up to 512 cores
  - DP **0.5 Tflop/s**
  - **3-6 GB of memory**
  - **Caches** included
    - **L1** per multiprocessor
    - **L2**: Shared
- Kepler in 2012
- Maxwell in 2014



NVIDA CUDA 6.5 SDK

Other accelerators

- **Intel**
  - Intel MIC (Many Integrated Core)
  - More than 50 X86 vector cores
  - OpenMP, OpenCL, Intel parallel building blocks…
  - First product (Knights Corner) in 2012



- **AMD**
  - Line of GPUs supporting OpenCL
  - Not so widely used in HPC at the moment

Intel® Xeon Phi™ Product Family Specifications

PRODUCT NUMBER	FORM FACTOR &, THERMAL SOLUTION[4]	BOARD TDP (WATTS)	NUMBER OF CORES	FREQUENCY (GHz)	PEAK DOUBLE PRECISION PERFORMANCE (GFLOP)	PEAK MEMORY BANDWIDTH (GB/s)	MEMORY CAPACITY (GB)	INTEL® TURBO BOOST TECHNOLOGY
3120P	PCIe, Passive	300	57	1.1	1003	240	6	N/A
3120A	PCIe, Active	300	57	1.1	1003	240	6	N/A
5110P	PCIe, Passive	225	60	1.053	1011	320	8	N/A
5120D	Dense form factor, None	245	60	1.053	1011	352	8	N/A
7110P	PCIe, Passive	300	61	1.238	1208	352	16	Peak turbo frequency: 1.33 GHz
7120X	PCIe, None	300	61	1.238	1208	352	16	

Intel Compilers and Intel Parallel Studio

Generation	Model	Video card equivalent	GPU Core	Threads max.	Core		Memory					Raw processing power (Floating-Point Operations per Second)		Peak TDP (watts)	Others
					SPUs [NB1]	Clock (MHz)	Bandwidth (GiB/s)	Type	Bus width (bit)	Amount (MiB)	Clock (MHz)	FP32 GFLOPs	FP64 GFLOPs		
1st [NB2]	580[7]/2U[10]	Radeon X1900 XTX	R580	512	48	600	83.2	GDDR3	256	1024	650	375[11]	N/A	≤165	
2nd [NB2]	9170[9][12]	Radeon HD 3870	RV670	?	64 (320)	800	51.2	GDDR3	256	2048	800	512	102.4 [NB3][13]	≤105	
3rd [NB2]	9250[14]	Radeon HD 4850	RV770	16,384[15]	160 (800)	625	63.5	GDDR3	256	1024	993	1000	200 [NB3]	≤150	
	9270[16]	Radeon HD 4870				750	108.8	GDDR5	256	2048	850	1200	240 [NB3]	<160	
4th [NB2]	9350	Radeon HD 5850	*Cypress (RV870)*	31,744[17]	288 (1440)	700	128	GDDR5	256	2048	1000	2016	403.2	150	codenamed *Kestrel*
	9370	Radeon HD 5870			320 (1600)	825	147.2		256	4096	1150	2640	528	225	codenamed *Osprey*

AMD stream processing lineup - AMD Stream SDK was replaced by AMD APP SDK, available for Microsoft Windows and Linux, 32-bit and 64-bit. APP stands for "Accelerated Parallel Processing"

How to use GPUs

1. **Use existing GPU software**
2. **Use numerical libraries for GPUs**
3. **Program GPU code with directives**
4. **Program native GPU code**

Use existing GPU software

- **HOOMD, NAMD, GROMACS, GPU-HMMER, TeraChem, Matlab (jacket etc)…**
- **Pros**
  - No implementation headaches for end users
- **Cons**
  - Existing applications do not cover all science areas
  - Often include limited number of algorithms/models
  - For many applications the GPU version is still immature

Use GPU libraries

- **CUBLAS, MAGMA (Lapack for GPU), …**
- **Pros**
  - Easy to implement in your code
  - Algorithms in libraries efficient
- **Cons**
  - Speedup limited by Amdahls law & transfer bottleneck

Directive based GPU code

- **Two main products**
  - PGI accelerator
  - HMPP (CAPS enterprise)
- **Normal C or Fortran code with directives to guide compiler in creating a GPU version**
- **Backends supporting CUDA, OpenCL and even normal CPUs**

```
//HMPP codelet
#pragma hmpp label1 codelet, args[B].io=out, args [C].io=inout,
target=CUDA:CAL/IL
void myFunc(int n, int A[n], int B[n], int C[n]) {
  for(int i=0 ; i<n ; i++) {
    B[i] = A[i] * A[i]; C[i] = C[i] * A[i];
}

//HMPP callsite
#pragma hmpp label1 callsite
  myFunc(n, A, B, C);
```

Native GPU code

- **CUDA, CUDA-Fortran (PGI), OpenCL**
- **Cons**
  - Requires most time
- **Pros**
  - Good control & performance
  - Can be combined with library & directive approaches

CUDA vs OpenCL

- **CUDA Nvidia specific**
- **OpenCL is a standard adopted by all major players**
  - Writing OpenCL code providing good performance on all platforms (Nvidia, AMD, etc.) is difficult
- **On Nvidia HW**
  - CUDA is faster (at the moment)
  - According to Nvidia
    - CUDA will evolve faster than OpenCl
    - Remains Nvidias main programming language

CUDA

- **C**ompute **U**nified **D**evice **A**rchitecture
- CUDA C is a C/C++ language extension for GPU programming
  - PGI has developed similar Fortran 2003 extension
- CUDA API is the most up-to-date GPGPU programming interface for NVIDIA GPUs
  - Two APIs: runtime and driver

CUDA C

Qualifiers
- global, device, shared, local, constant, ...

Built-in variables
- threadIdx, blockIdx, ...

Intrinsics
- __syncthreads, __fmul_rn, ...

Runtime API
- memory, device, execution management

Kernel launch

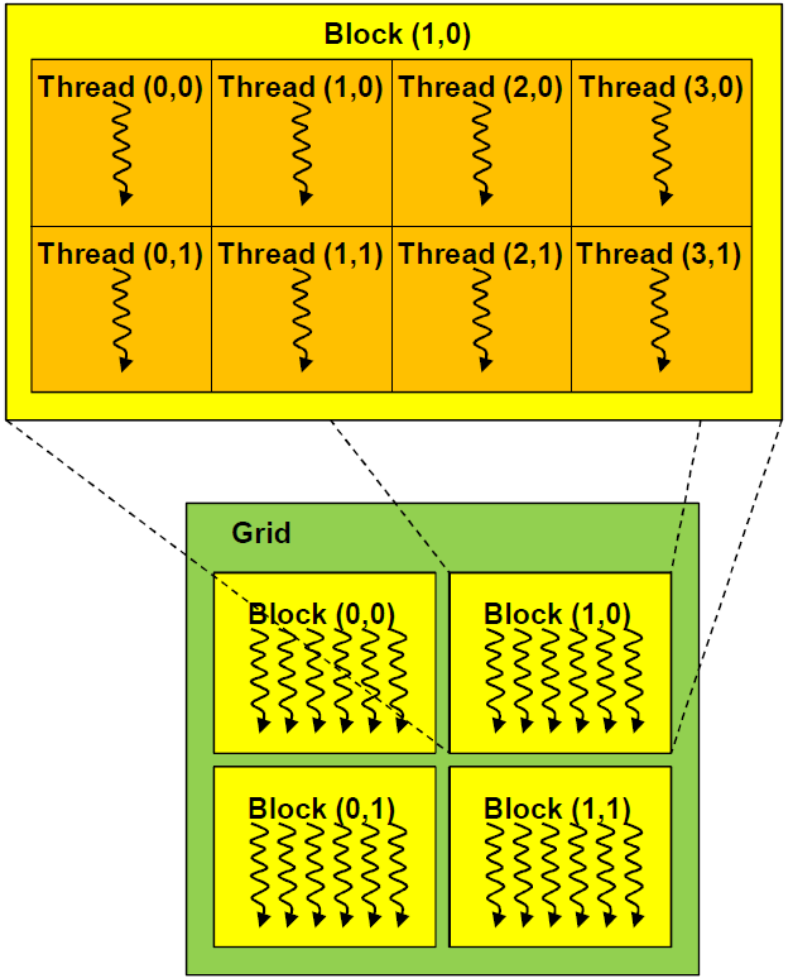
```
__device__ float array[128];
__global__ void kern(float *data) {
    __shared__ float buffer[32];
    ...
    buffer[threadIdx.x] = data[i];
    ...
    __syncthreads;
    ...
}

float *d_data;
cudaMalloc((void**)&d_data, bytes);

kern<<<1024, 128>>>(d_data);
```

CUDA PROGRAMMING MODEL

- GPU accelerator is called device, CPU is host
- GPU code (kernel) is launched and executed on the device by several threads
- Threads are grouped into thread blocks
- Program code is written from a single thread's point of view
  - Each thread can diverge and execute a unique code path (can cause performance issues)

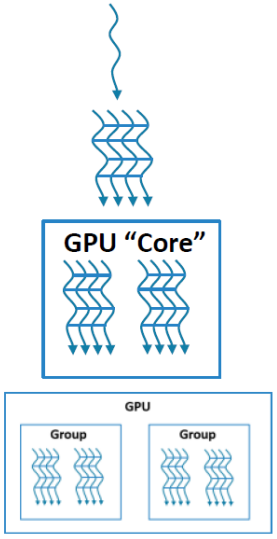
THREAD HIERARCHY

- Threads:
  - 3D IDs, unique in block
- Blocks:
  - 3D* IDs, unique in grid
- Dimensions are set at kernel launch
- Built-in variables for device code:
  - threadIdx, blockIdx
  - blockDim, gridDim



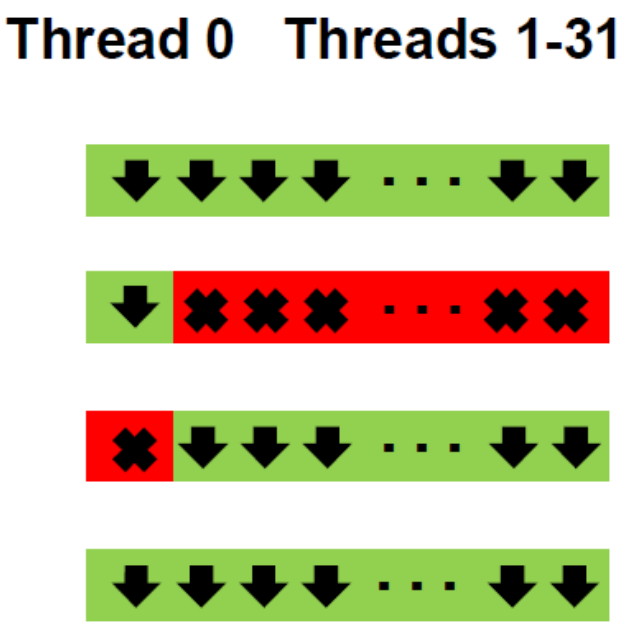
Hardware Implementation, SIMT Architecture

- Maximum number of threads in a block depends on the compute capability (1024 on Fermi)
- GPU multiprocessor creates, manages, schedules and executes threads in warps of 32*
- Warp executes one common instruction at a time
  - Threads are allowed to branch, but branches are serialized
- Context switch is fast, scheduler selects warps that are ready to execute → can hide latencies



Nvidia/CUDA	AMD/OpenCL	Nilay's CPU Analogy
CUDA Processor	Processing Element	Lane
CUDA Core	SIMD Unit	Pipeline
Streaming Multiprocessor	Compute Unit	Core
GPU Device	GPU Device	Device

Thread Branching



Program

```
int tid = threadIdx.x

if (tid == 0) {++var1;}

else {var1 = var1 + 2;}

var2 = 3 * var1;
```

Host and device have separate memories

Host manages the GPU memory

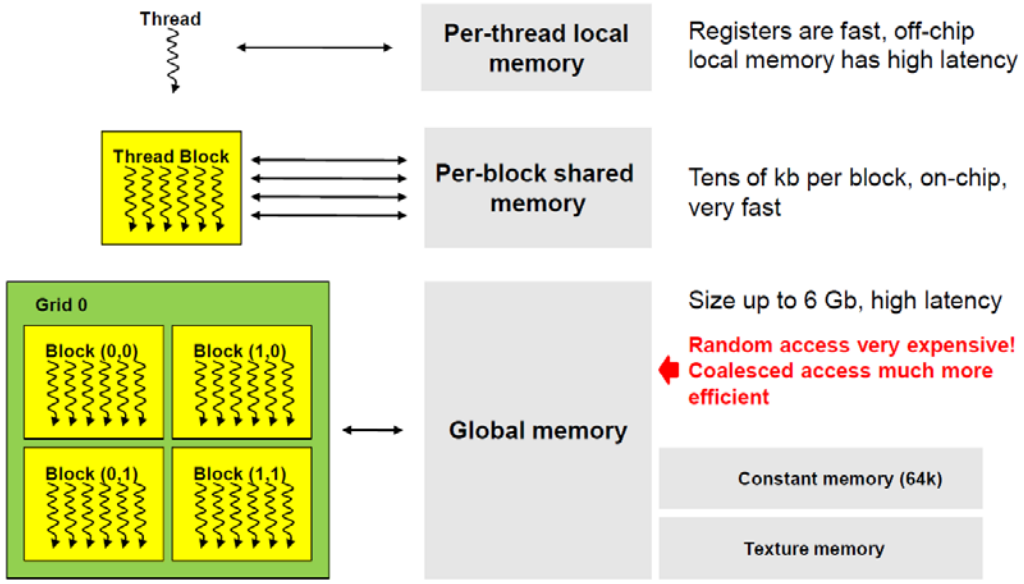
Usually one has to

1. Copy (explicitly) data from host to the device
2. Execute the GPU kernel
3. Copy (explicitly) the results back to the host

Data copies between host and device use the PCI bus with very limited bandwidth → minimize the transfers!

```
int main(void) {

   float *A = (float *) malloc(N*sizeof(float));
   float *d_A;
   cudaMalloc((void**)&d_A, N*sizeof(float));
   cudaMemcpy(d_A, A, N*sizeof(float), cudaMemcpyHostToDevice);
   ...
   float A0 = d_A[0];
   ...
   cudaMemcpy(A, d_A, N*sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(d_A);
   free(A);
   return 0;
}
```

Can not dereference device pointers in host code!



Device Code

- C functions with restrictions:
  - Can only dereference pointers to device memory
  - No static variables, no recursion
  - No variable number of arguments
- Functions must be declared with a qualifier
  - **__global__**: Kernel, called from CPU
    - Cannot be called from GPU
    - Must return void
  - **__device__**: Called from **__device__** and **__global__** funcs
    - Can not be called from CPU
  - **__host__**: Can only be called by CPU
    - Can be combined with **__device__** qualifier

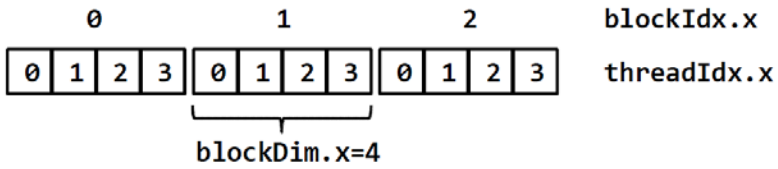
Kernel Example

3 blocks with 4 threads in each

```
__global__ void kern(int *A) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   A[idx] = idx;
}
```

Result: A = {0,1,2,3,4,5,6,7,8,9,10,11}

```
void main() {
   // Allocate memories, copy values
   dim3 grid, block;
   block.x = 4;
   grid.x = 3;
   kern<<<grid, block>>>(d_A);
   // Copy results back
}
```




```
__global__ void kern(int *A) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   A[idx] = blockIdx.x;
}
```

Result: A = {0,0,0,0,1,1,1,1,2,2,2,2}

```
__global__ void kern(int *A) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   A[idx] = threadIdx.x;
}
```

Result: A = {0,1,2,3,0,1,2,3,0,1,2,3}

Synchronization

- Kernel-level synchronization
- Blocks must be independent
  - Can run in any order, concurrently or sequentially
  - Can't synchronize between blocks
- Some level of coordination can be achieved using atomic intrinsics → performance issues
- Threads in a block can synchronize using __syncthreads intrinsic

- Compilation tools are a part of CUDA SDK
- Compiler driver is called nvcc
- nvcc separates the code for host and device
  - Host code is compiled with regular C/C++ compiler
- nvcc or C/C++ can be used for linking
- Note that nvcc uses C++ front end to parse the program code
- For Linux environment NVIDIA provides a command-line debugger, cuda-gdb
  - For memory access violation checks there is also cuda-memcheck
- Profiling information can be gathered and visualized using computeprof tool

Coalesced Memory Access

- Global memory access has very high latency
- Threads are executed in warps, memory operations are grouped in a similar fashion
  - Memory access is optimized for coalesced access where threads read from / write to successive memory locations
- Shared memory is better suited for more complicated data access

Pinned Memory

- Normal malloc call returns a pointer to virtual memory → swapping, page faults
- Pinning is beneficial in some cases
  - Higher transfer speeds between host and device
  - Copying can be interleaved with kernel execution
- Pinning large memory areas can decrease overall system performance

CUDA STREAMS

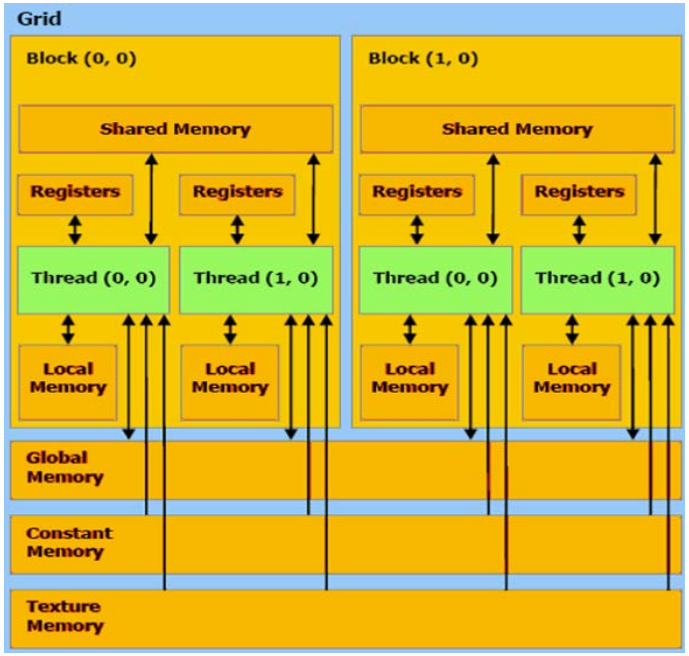
- Stream is a sequence of commands that execute in order
- Streams can be used to overlap memory copies and kernel execution
  - Copy-execute sequences are split into separate blocks that are associated to different streams

ERROR CHECKING

- All runtime function calls return an error code
  - Important to check!
- Kernel launches do not return error codes
  - Have to use separate error checking routines
- Asynchronous operations can fail long after the call has returned → finding exact place of error can be difficult
  - For development phase extra synchronization may be useful

SUMMARY

- CUDA programs consist of host and device code
- Device code is run parallel using threads organized into a grid of thread blocks
  - Kernel launch parameters
- Device and host have separate memories
  - Memory allocations and transfers



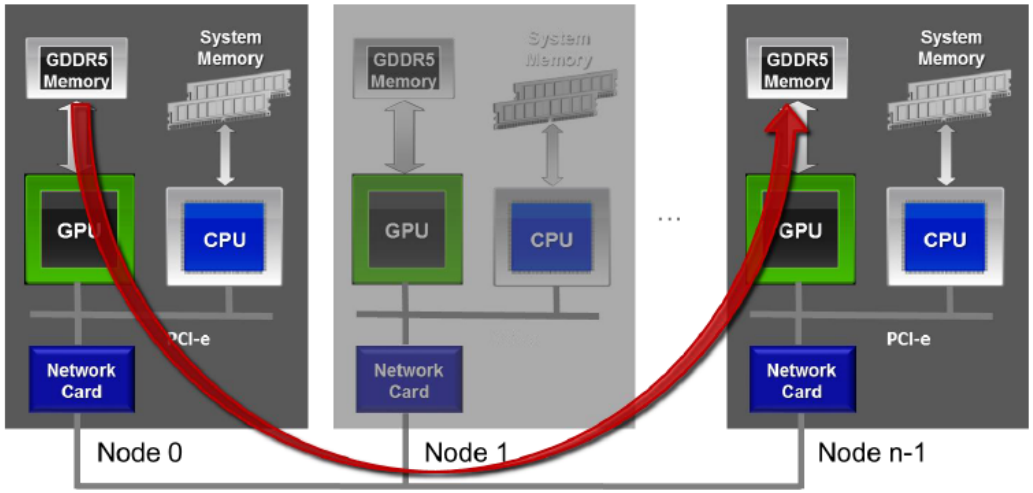
CUDA + MPI + OpenMP

OpenCL + MPI + OpenMP

Then we look at my notes that has runs and links to code run – on Discovery Cluster

http://nuweb12.neu.edu/rc/wp-content/uploads/2014/11/CUDA-MPI.pdf

MPI+CUDA



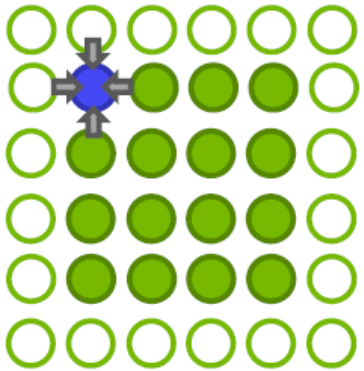

```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,n-1,tag,MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

EXAMPLE: JACOBI SOLVER – SINGLE GPU

While not converged

- Do Jacobi step:

```
for (int i=1; i < n-1; i++)
  for (int j=1; j < m-1; j++)
    u_new[i][j] = 0.0f - 0.25f*(u[i-1][j] + u[i+1][j]
                              + u[i][j-1] + u[i][j+1])
```



- Swap `u_new` and `u`
- Next iteration

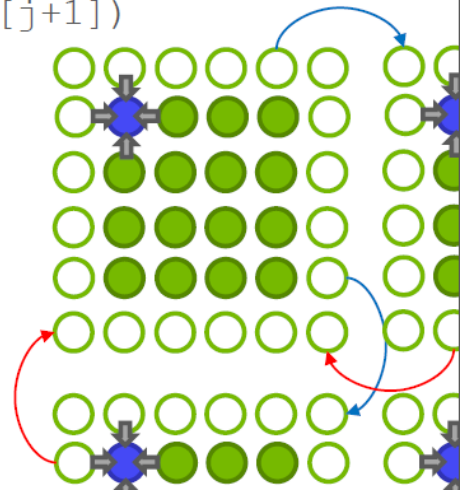
EXAMPLE: JACOBI SOLVER – MULTI GPU

While not converged

- Do Jacobi step:

```
for (int i=1; i < n-1; i++)
  for (int j=1; j < m-1; j++)
    u_new[i][j] = 0.0f - 0.25f*(u[i-1][j] + u[i+1][j]
                               + u[i][j-1] + u[i][j+1])
```

- Exchange halo with 2 4 neighbor
- Swap `u_new` and `u`
- Next iteration



EXAMPLE: JACOBI SOLVER

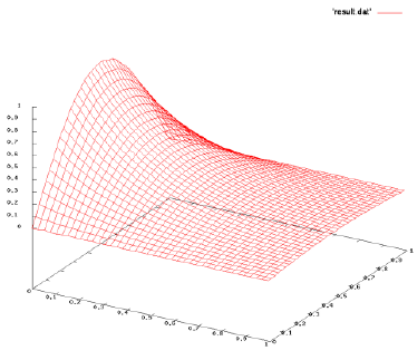
- Solves the 2D-Laplace equation on a rectangle

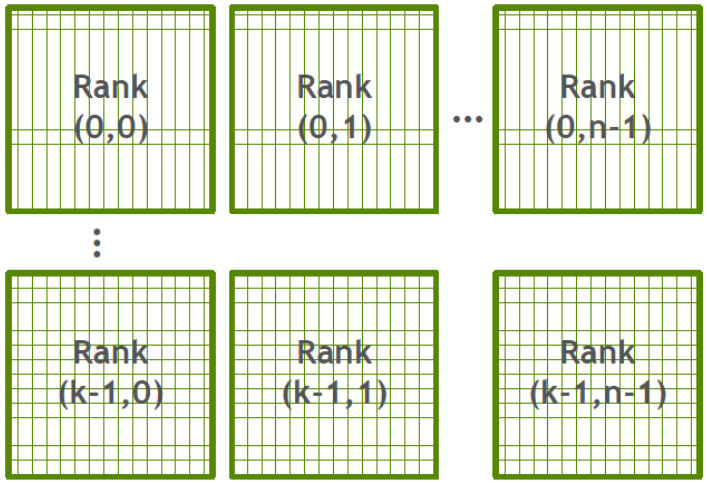
$$\Delta u(x,y) = 0 \; \forall \; (x,y) \in \Omega \backslash \delta\Omega$$

  - Dirichlet boundary conditions (constant values on boundaries)

$$u(x,y) = f(x,y) \in \delta\Omega$$

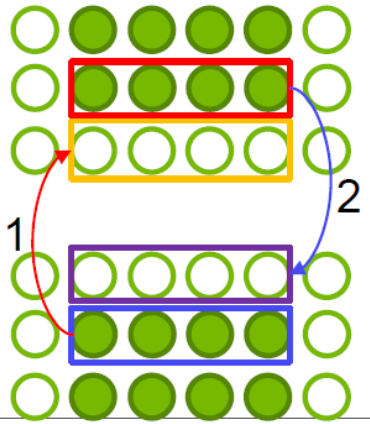
- 2D domain decomposition with n x k domains



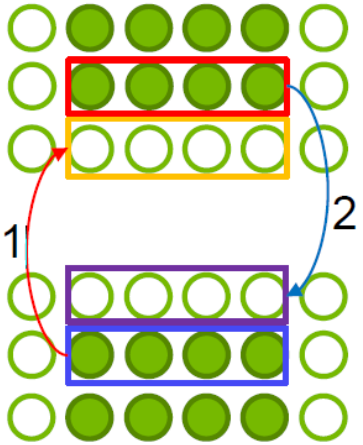


EXAMPLE: JACOBI – TOP/BOTTOM HALO UPDATE

```
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```




```
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



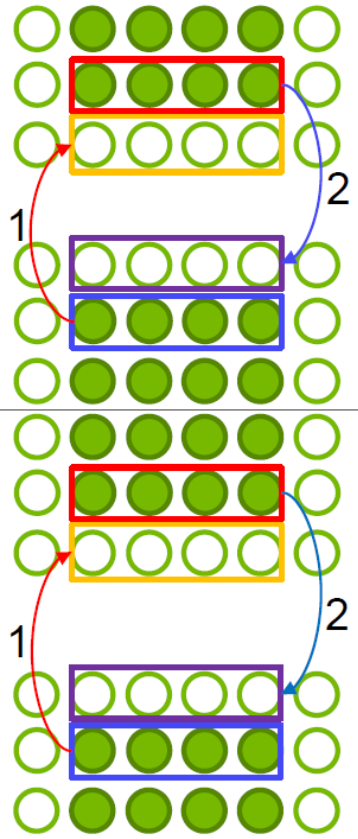
EXAMPLE: JACOBI – TOP/BOTTOM HALO UPDATE

OpenACC

```
#pragma acc host_data use_device ( u_new ) {
  MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
               u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
               u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

CUDA

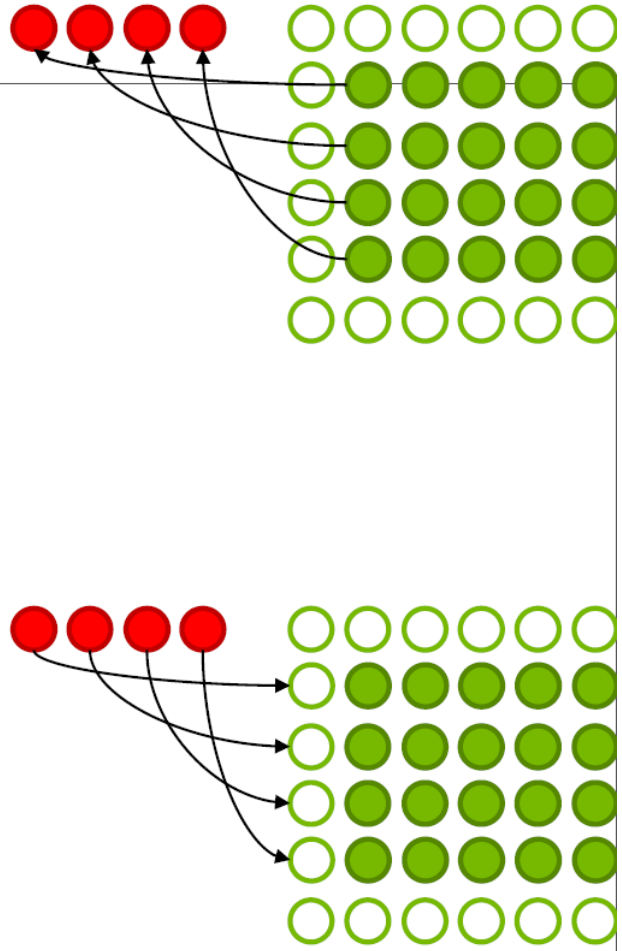
```
MPI_Sendrecv(u_new_d+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new_d+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(u_new_d+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new_d+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



EXAMPLE: JACOBI – LEFT/RIGHT HALO UPDATE

OpenACC

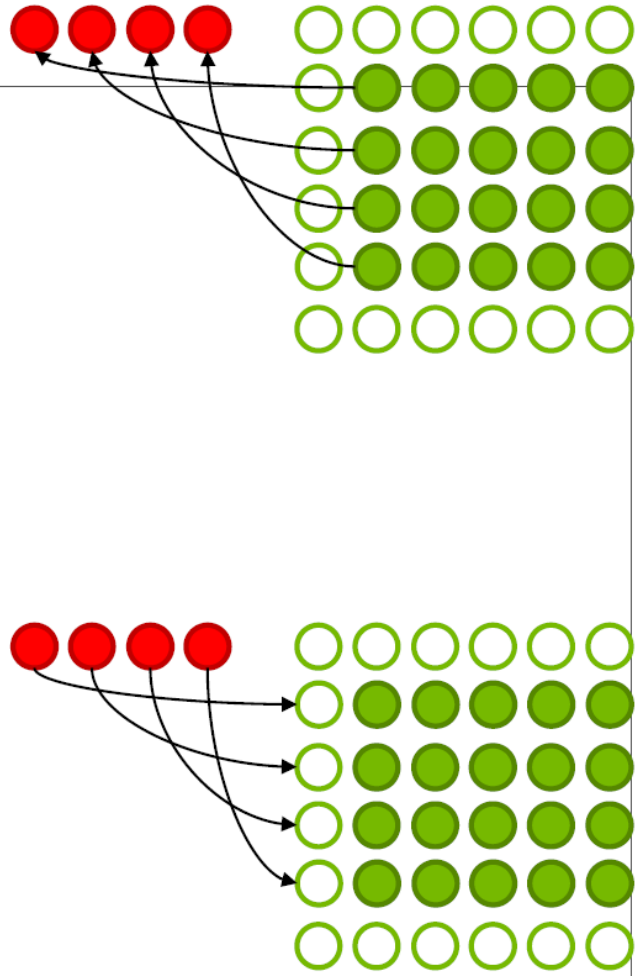
```
//right neighbor omitted
#pragma acc parallel loop present ( u_new, to_left )
for ( int i=0; i<n-2; ++i )
     to_left[i] = u_new[(i+1)*m+1];

#pragma acc host_data use_device ( from_left, to_left ) {
  MPI_Sendrecv( to_left, n-2, MPI_DOUBLE, l_nb, 0,
                from_left, n-2, MPI_DOUBLE, l_nb, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}

#pragma acc parallel loop present ( u_new, from_left )
for ( int i=0; i<n-2; ++i )
     u_new[(i+1)*m] = from_left[i];
```



EXAMPLE: JACOBI – LEFT/RIGHT HALO UPDATE

CUDA

```
//right neighbor omitted
pack<<<gs,bs,0,s>>>(to_left_d, u_new_d, n, m);
cudaStreamSynchronize(s);




MPI_Sendrecv( to_left_d, n-2, MPI_DOUBLE, l_nb, 0,
              from_left_d, n-2, MPI_DOUBLE, l_nb, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );




unpack<<<gs,bs,0,s>>>(u_new_d, from_left_d, n, m);
```



EXAMPLE: JACOBI - TOP/BOTTOM HALO UPDATE - WITHOUT CUDA-AWARE MPI

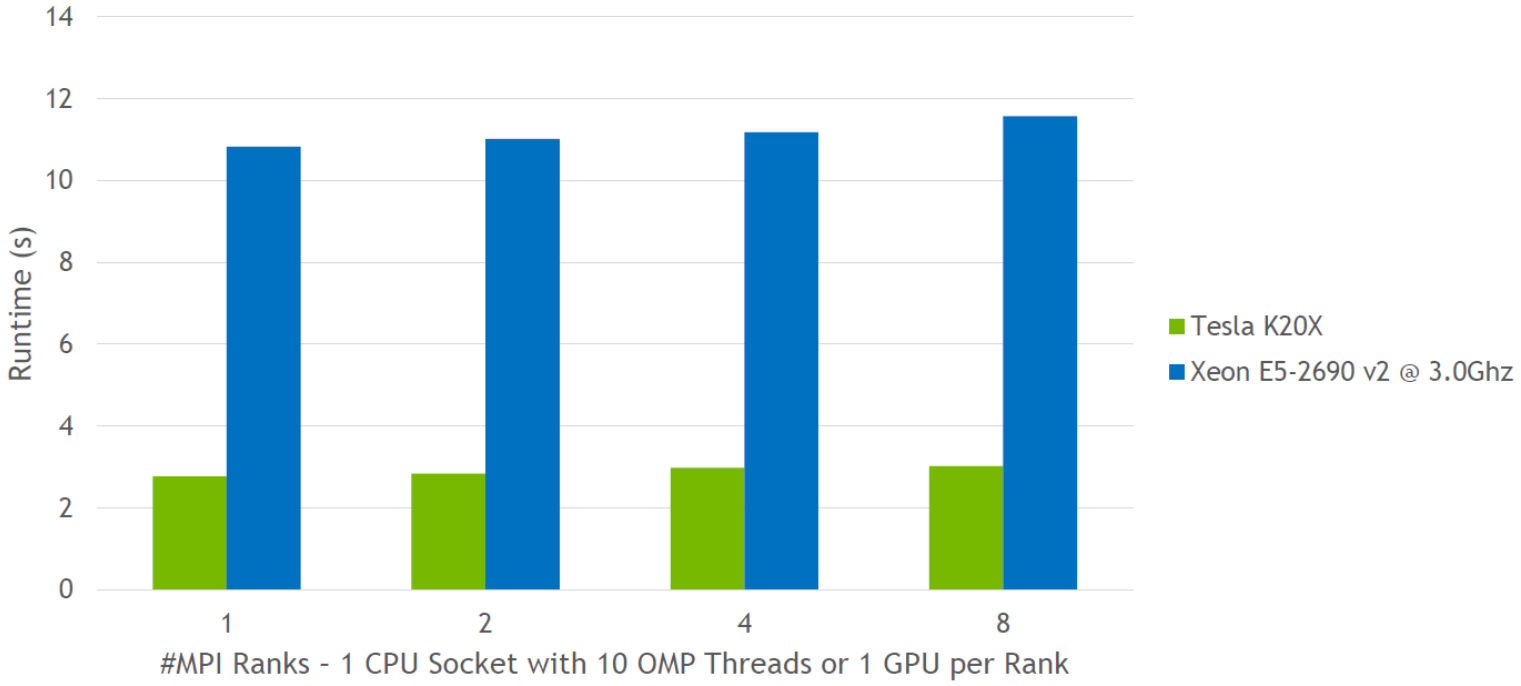
OpenACC

```
#pragma acc update host( u_new[1:m-2], u_new[(n-2)*m+1:m-2] )
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
#pragma acc update device( u_new[0:m-2], u_new[(n-2)*m:m-2] )
```

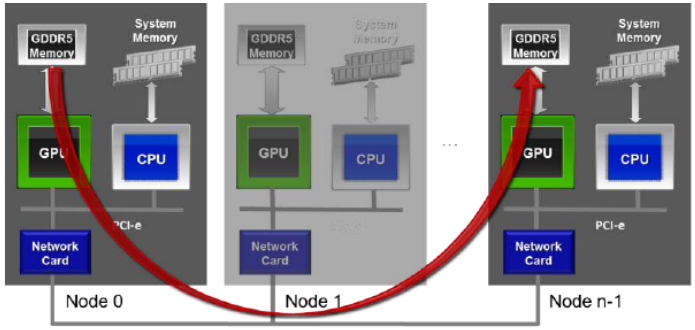
CUDA

```
//send to bottom and receive from top - top bottom omitted

cudaMemcpy(u_new+1, u_new_d+1, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
cudaMemcpy(u_new_d, u_new, (m-2)*sizeof(double), cudaMemcpyDeviceToHost);
```

JACOBI RESULTS (1000 STEPS)
WEAK SCALING 4K X 4K PER PROCESS



MPI+CUDA



With UVA and CUDA-aware MPI

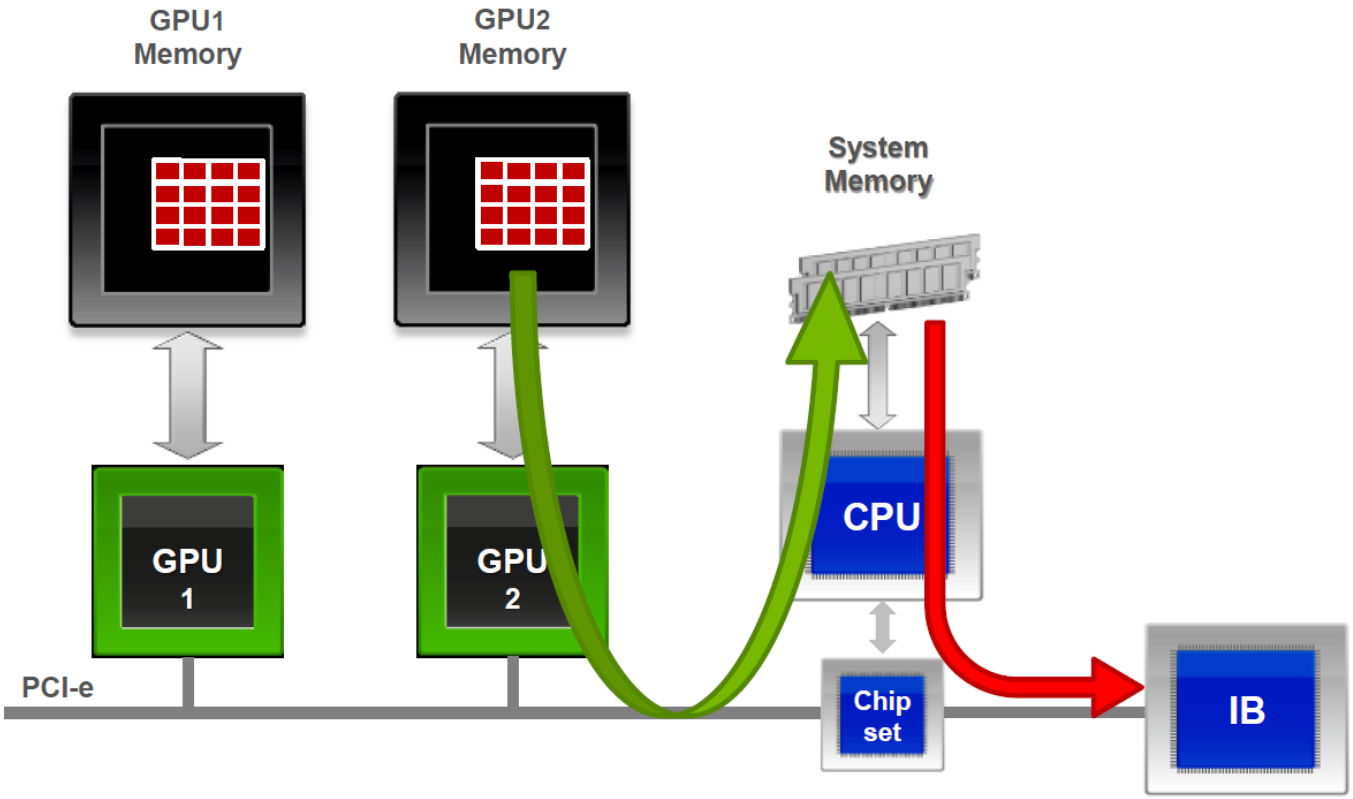
```
//MPI rank 0
MPI_Send(s_buf_d,size,…);

//MPI rank n-1
MPI_Recv(r_buf_d,size,…);
```

No UVA and regular MPI

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,…);
MPI_Send(s_buf_h,size,…);

//MPI rank n-1
MPI_Recv(r_buf_h,size,…);
cudaMemcpy(r_buf_d,r_buf_h,size,…);
```

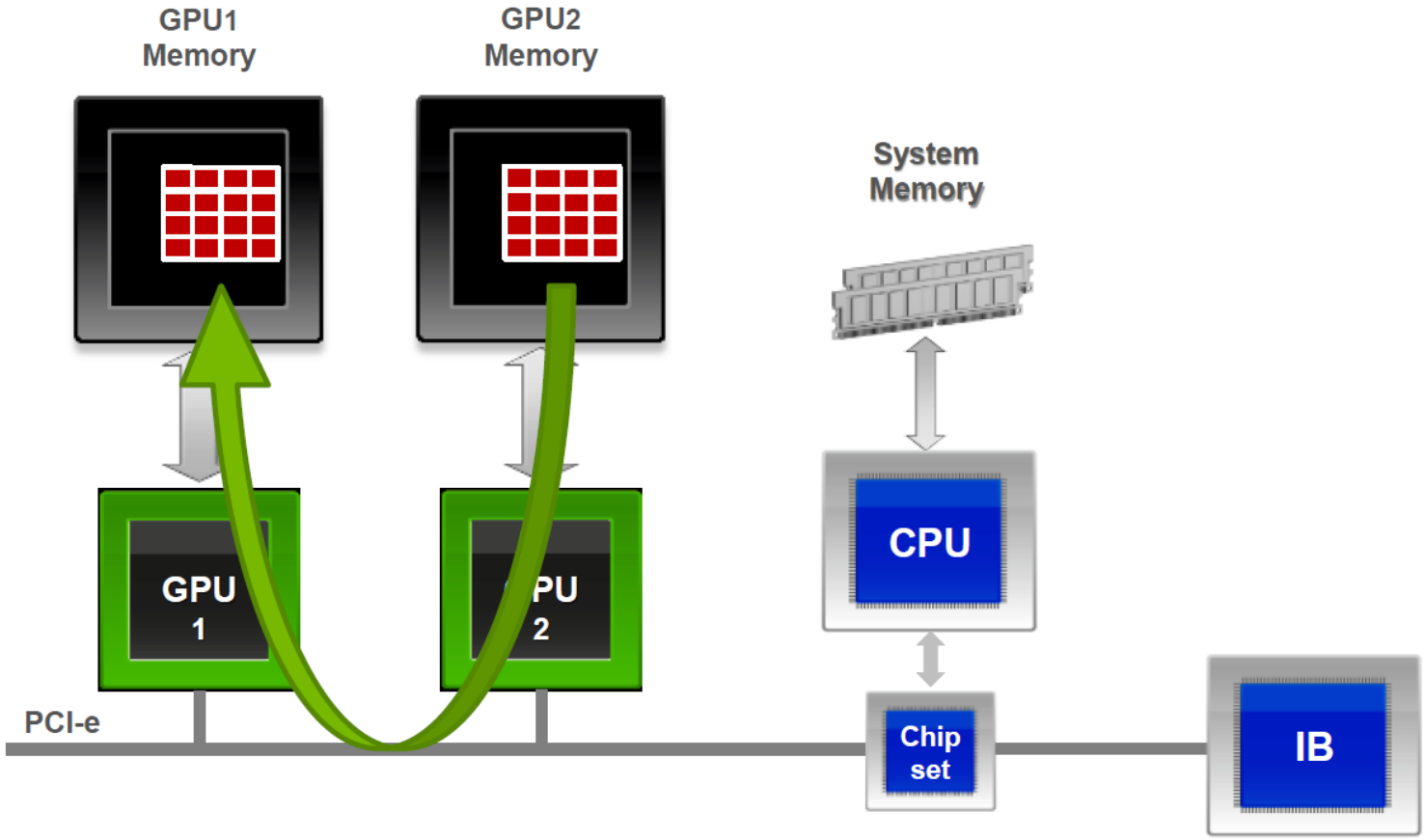
NVIDIA GPUDIRECT™

ACCELERATED COMMUNICATION WITH NETWORK & STORAGE DEVICES



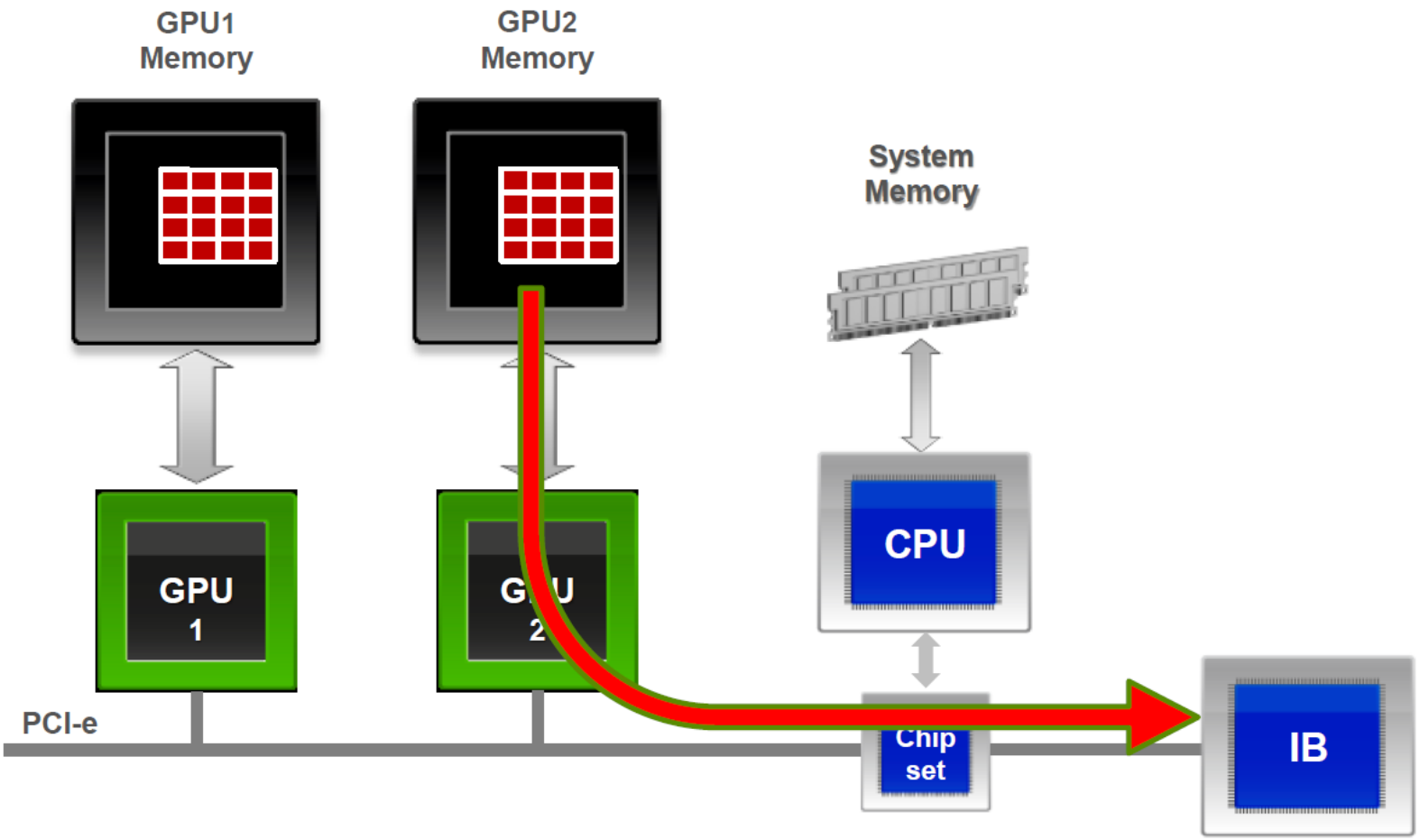
NVIDIA GPUDIRECT™

PEER TO PEER TRANSFERS

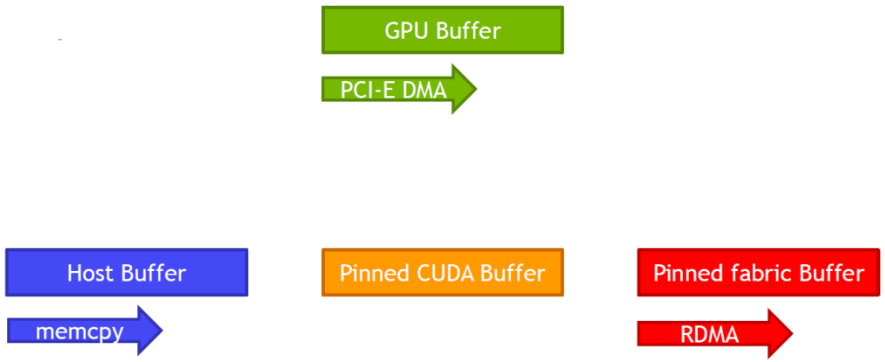


NVIDIA GPUDIRECT™

SUPPORT FOR RDMA

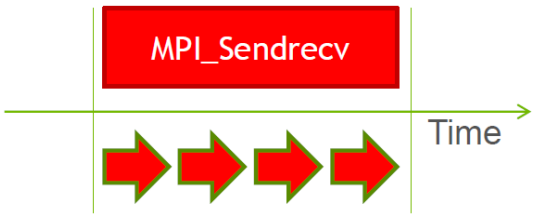


CUDA-AWARE MPI

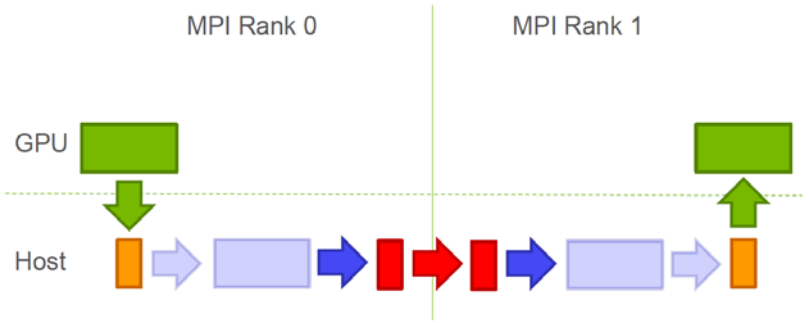


MPI GPU TO REMOTE GPU
GPUDIRECT SUPPORT FOR RDMA

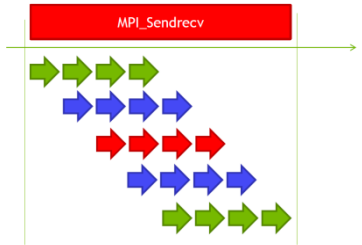

```
MPI_Send(s_buf_d,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```



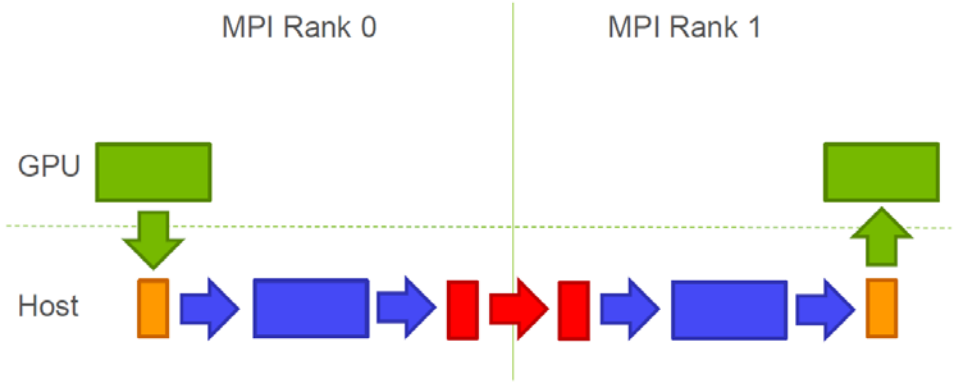
MPI GPU TO REMOTE GPU
WITHOUT GPUDIRECT




```
MPI_Send(s_buf_h,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf_h,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

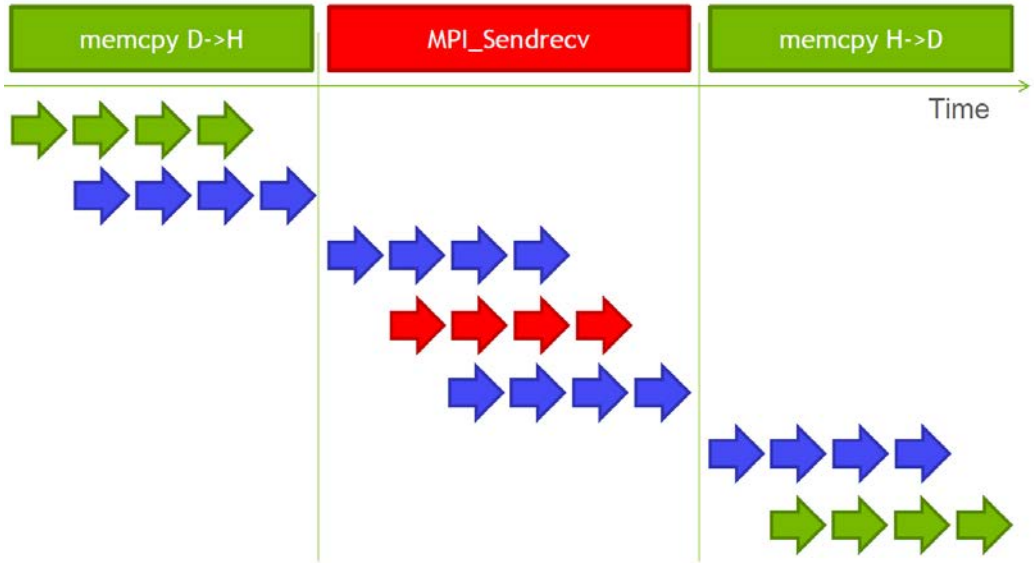


REGULAR MPI GPU TO REMOTE GPU

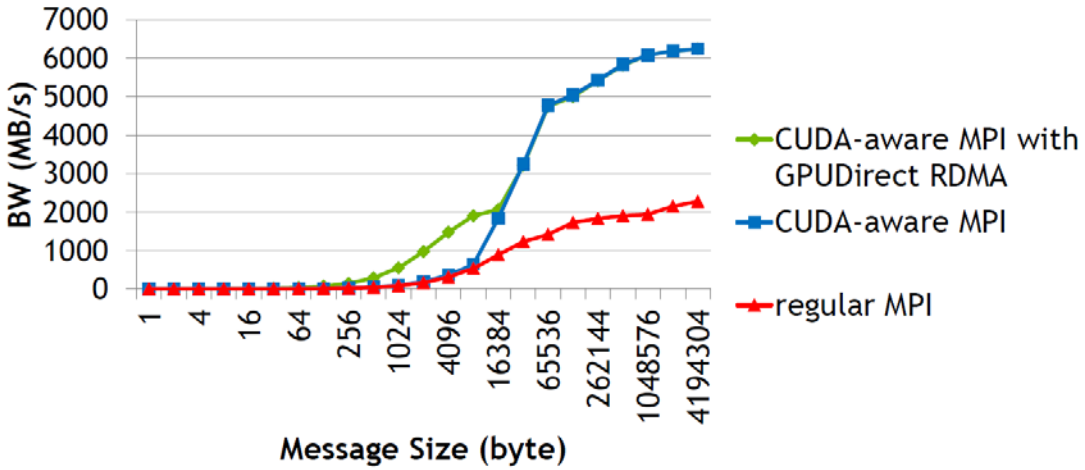



```
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);

MPI_Recv(r_buf_h,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

REGULAR MPI GPU TO REMOTE GPU



PERFORMANCE RESULTS TWO NODES - EXAMPLE



Latency (1 byte)	19.04 us	16.91 us	5.52 us

GPU ACCELERATION OF LEGACY MPI APPLICATION

Typical legacy application

—MPI parallel

—Single or few threads per MPI rank (e.g. OpenMP)

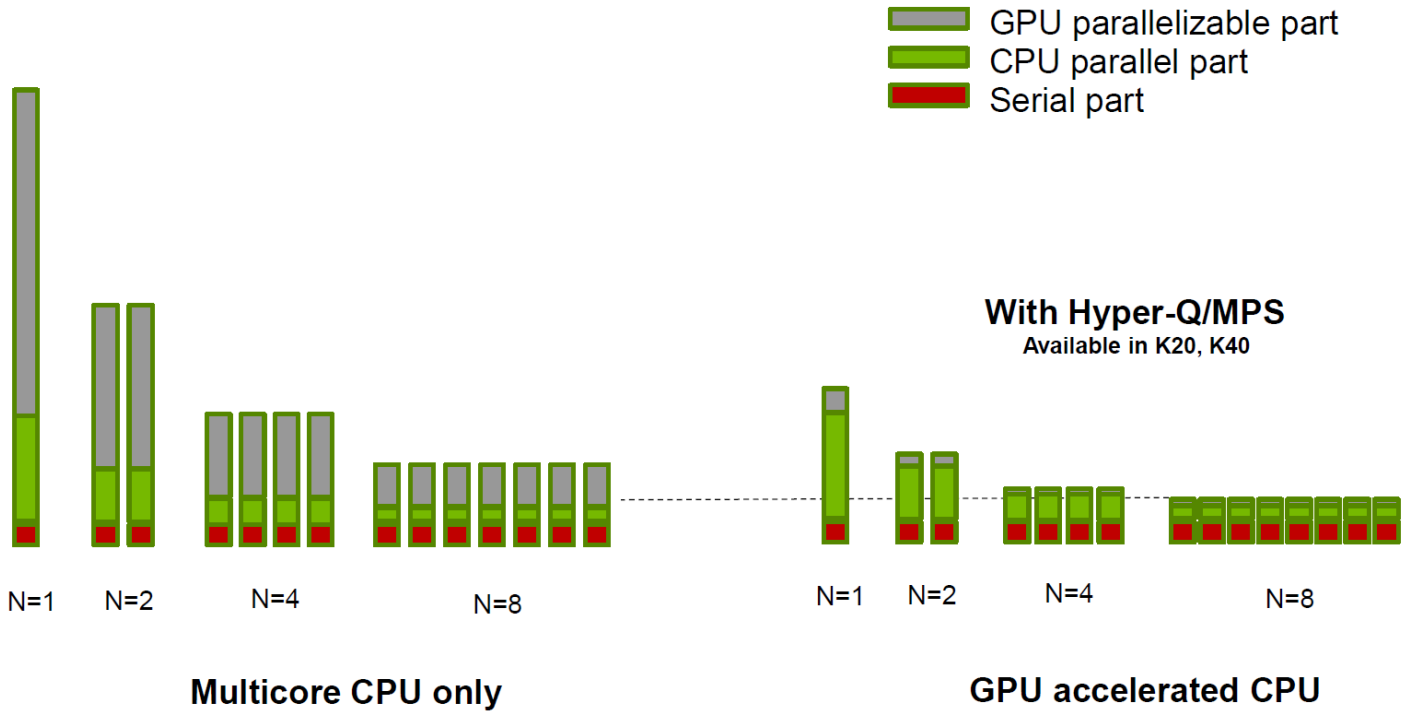
Running with multiple MPI ranks per node

GPU acceleration in phases

—Proof of concept prototype, ..

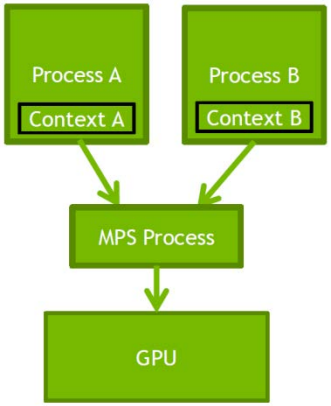
—Great speedup at kernel level

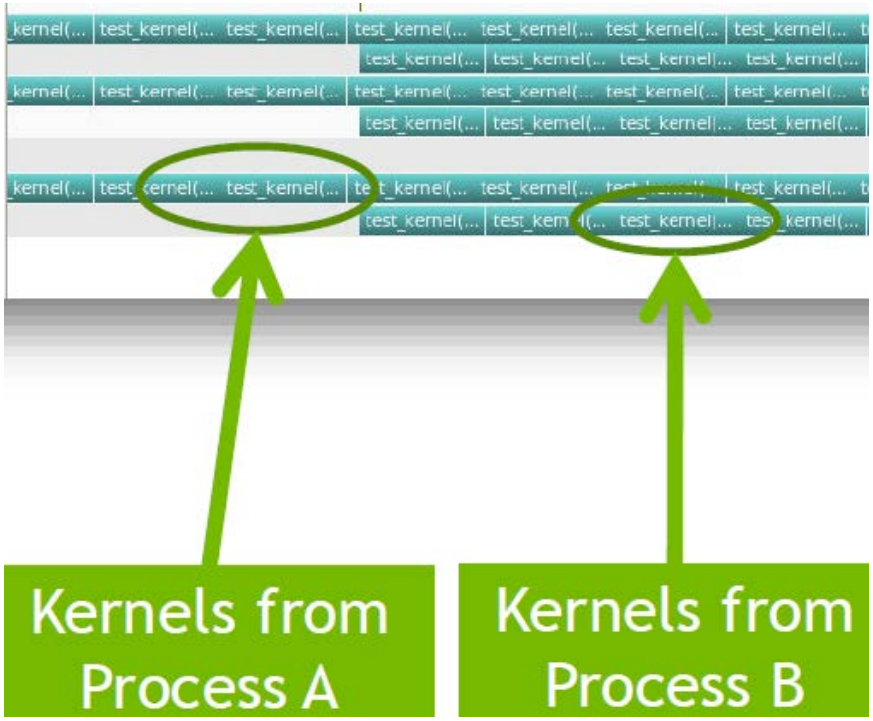
Application performance misses expectations



PROCESSES SHARING GPU WITH MPS / HYPER-Q:

- MAXIMUM OVERLAP
- Enables overlap between copy and compute of different processes
- Sharing the GPU between multi-MPI ranks increases GPU utilization





BEST PRACTICE: USE NONE-BLOCKING MPI

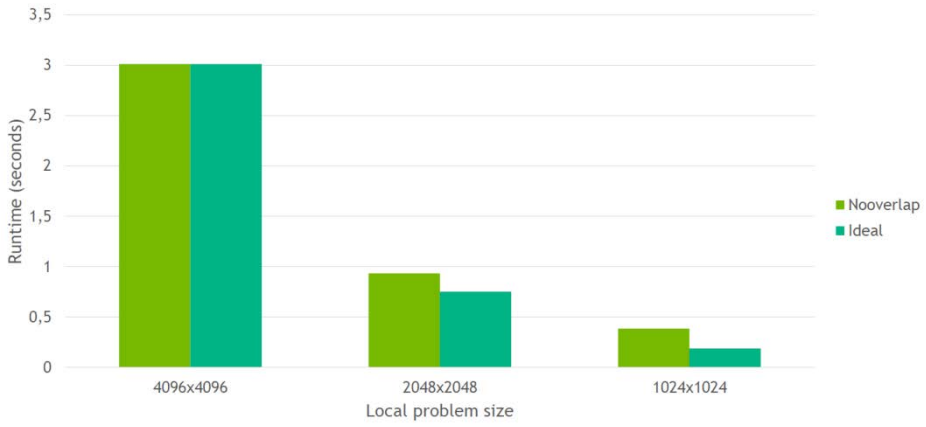
BLOCKING

```
#pragma acc host_data use_device ( u_new ) {
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

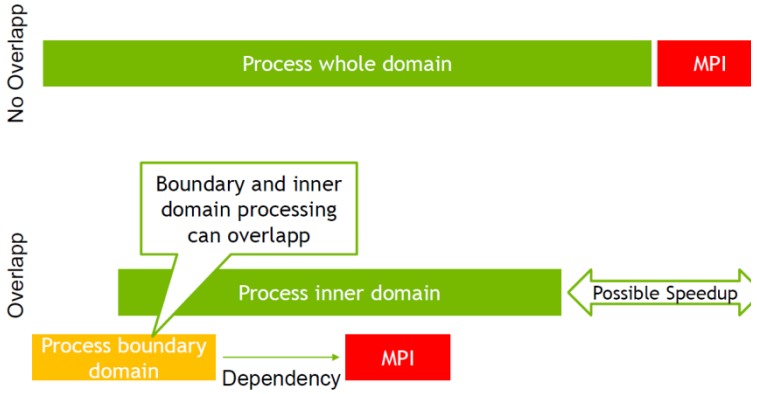
NONE-BLOCKING

```
MPI_Request t_b_req[4];
#pragma acc host_data use_device ( u_new ) {
  MPI_Irecv(u_new+offset_top_bondary,m-2
  MPI_Irecv(u_new+offset_bottom_bondar
  MPI_Isend(u_new+offset_last_row,m-2,M
  MPI_Isend(u_new+offset_first_row,m-2,I
}
MPI_Waitall(4, t_b_req, MPI_STATUSES_IGNORE);
```

Gives MPI more opportunities to build efficient piplines



OVERLAPPING COMMUNICATION AND COMPUTATION

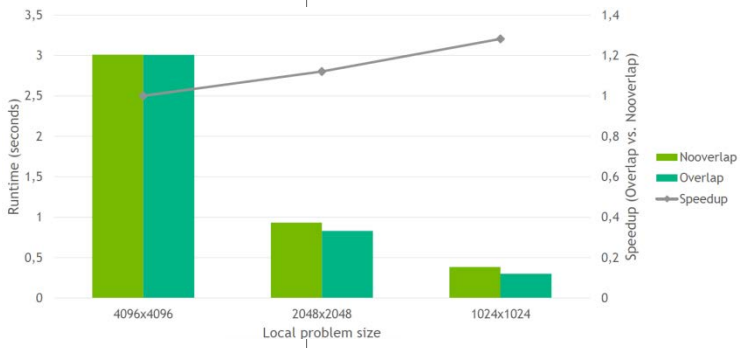


OpenACC

```
#pragma acc parallel loop present ( u_new, u, to_left, to_right ) async(1)
for ( ... )
      //Process boundary and pack to_left and to_right
#pragma acc parallel loop present ( u_new, u ) async(2)
for ( ... )
      //Process inner domain
#pragma acc wait(1)              //wait for boundary
MPI_Request req[8];
#pragma acc host_data use_device ( from_left, to_left, form_right, to_right, u_new ) {
  //Exchange halo with left, right, top and bottom neighbor
}
MPI_Waitall(8, req, MPI_STATUSES_IGNORE);
#pragma acc parallel loop present ( u_new, from_left, from_right )
for ( ... )
      //unpack from_left and from_right
#pragma acc wait                 //wait for iteration to finish
```

CUDA

```
process_boundary_and_pack<<<gs_b,bs_b,0,s1>>>(u_new_d,u_d,to_left_d,to_right_d,n,m);


process_inner_domain<<<gs_id,bs_id,0,s2>>>(u_new_d, u_d,to_left_d,to_right_d,n,m);


cudaStreamSynchronize(s1);       //wait for boundary
MPI_Request req[8];

  //Exchange halo with left, right, top and bottom neighbor

MPI_Waitall(8, req, MPI_STATUSES_IGNORE);
unpack<<<gs_s,bs_s>>>(u_new_d, from_left_d, from_right_d, n, m);


cudaDeviceSynchronize();         //wait for iteration to finish
```



MPI AND UNIFIED MEMORY

- Unified Memory support for CUDA-aware MPI needs changes to the MPI implementations
  - Check with your MPI implementation of choice for their plans
  - It might work in some situations but it is not supported
- Unified Memory and regular MPI
  - Require unmanaged staging buffers
    - Regular MPI has no knowledge of managed memory
    - CUDA 6 managed memory does not play well with RDMA protocols

HANDLING MULTI GPU NODES

- Multi GPU nodes and GPU-affinity:
  - Use local rank:

```
int local_rank = //determine local rank
int num_devices = 0;
cudaGetDeviceCount(&num_devices);
cudaSetDevice(local_rank % num_devices);
```

  - Use exclusive process mode + `cudaSetDevice(0)`
- How to determine local rank:
  - Rely on process placement (with one rank per GPU)

```
int rank = 0;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
int num_devices = 0;
cudaGetDeviceCount(&num_devices); // num_devices == ranks per node
int local_rank = rank % num_devices;
```

  - Use environment variables provided by MPI launcher
    - e.g for OpenMPI

```
int local_rank = atoi(getenv("OMPI_COMM_WORLD_LOCAL_RANK"));
```

TOOLS FOR MPI+CUDA APPLICATIONS

- Memory Checking `cuda-memcheck`
- Debugging `cuda-gdb`
- Profiling `nvprof` and NVIDIA Visual Profiler

MEMORY CHECKING WITH CUDA-MEMCHECK

- Cuda-memcheck is a functional correctness checking suite similar to the valgrind memcheck tool
- Can be used in a MPI environment

```
mpiexec -np 2 cuda-memcheck ./myapp <args>
```

- Problem: output of different processes is interleaved
  - Use save, log-file command line options and launcher script

```
#!/bin/bash
LOG=$1.$OMPI_COMM_WORLD_RANK
#LOG=$1.$MV2_COMM_WORLD_RANK
cuda-memcheck --log-file $LOG.log --save $LOG.memcheck $*

mpiexec -np 2 cuda-memcheck-script.sh ./myapp <args>
```

DEBUGGING MPI+CUDA APPLICATIONS

USING CUDA-GDB WITH MPI APPLICATIONS

- You can use cuda-gdb just like gdb with the same tricks
- For smaller applications, just launch xterms and cuda-gdb

```
> mpiexec -x -np 2 xterm -e cuda-gdb ./myapp <args>
```

DEBUGGING MPI+CUDA APPLICATIONS

CUDA-GDB ATTACH

- CUDA 5.0 and forward have the ability to attach to a running process

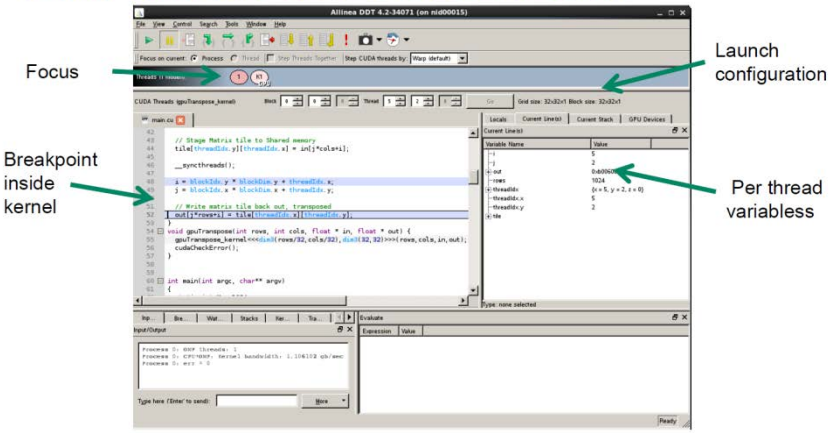
```
if ( rank == 0 )  {
    int i=0;
    printf("rank %d: pid %d on %s ready for attach\n.", rank, getpid(),name);
    while (0 == i) {
        sleep(5);
    }
}
> mpiexec -np 2 ./jacobi_mpi+cuda
rank 0: pid 30034 on ge107 ready for attach
 > ssh ge107
 cuda-gdb --pid 30034
```

DEBUGGING MPI+CUDA APPLICATIONS

THIRD PARTY TOOLS

- Allinea DDT debugger
- Totalview

DDT: THREAD LEVEL DEBUGGING



PROFILING MPI+CUDA APPLICATIONS
USING NVPROF+NVVP

3 Usage modes:

- Embed pid in output filename

```
mpirun -np 2 nvprof --output-profile profile.out.%p
```

- Only save the textual output

```
mpirun -np 2 nvprof --log-file profile.out.%p
```

- Collect profile data on all processes that run on a node

```
nvprof --profile-all-processes -o profile.out.%p
```

PROFILING MPI+CUDA APPLICATIONS
THIRD PARTY TOOLS

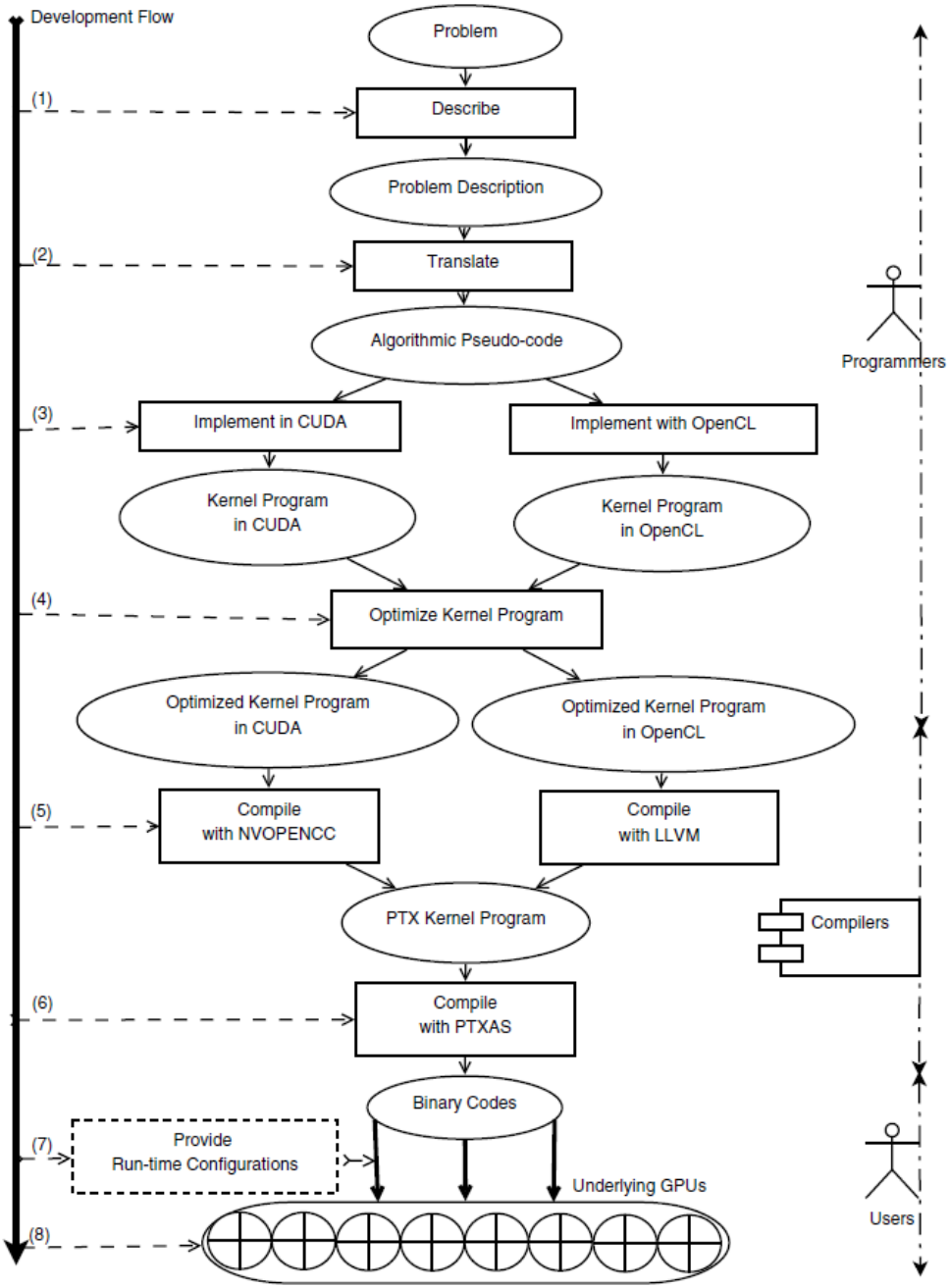
- Multiple parallel profiling tools are CUDA aware
  - Score-P
  - Vampir
  - Tau
- These tools are good for discovering MPI issues as well as basic CUDA performance inhibitors

OVERLAPPING COMMUNICATION AND COMPUTATION – TIPS AND TRICKS

- CUDA-aware MPI might use the default stream
  - Allocate stream with the non-blocking flag (cudaStreamNonBlocking)
- In case of multiple kernels for boundary handling the kernel processing the inner domain might sneak in
  - Use single stream or events for inter stream dependencies via cudaStreamWaitEvent (#pragma acc wait async) - disables overlapping of boundary and inner domain kernels
  - Use high priority streams for boundary handling kernels - allows overlapping of boundary and inner domain kernels
- As of CUDA 6.0 GPUDirect P2P in multi process can overlap disable it for older releases

CONCLUSIONS

- Using MPI as abstraction layer for Multi GPU programming allows multi GPU programs to scale beyond a single node
  - CUDA-aware MPI delivers ease of use, reduced network latency and increased bandwidth
- All NVIDIA tools are usable and third party tools are available
- Multipe CUDA-aware MPI implementations available
  - OpenMPI, MVAPICH2, Cray, IBM Platform MPI


Development Flow
Problem
(1)
Describe
Problem Description
(2)
Translate
Algorithmic Pseudo-code
Programmers
(3)
Implement in CUDA
Implement with OpenCL
Kernel Program in CUDA
Kernel Program in OpenCL
(4)
Optimize Kernel Program
Optimized Kernel Program in CUDA
Optimized Kernel Program in OpenCL
(5)
Compile with NVOPENCC
Compile with LLVM
PTX Kernel Program
Compilers
(6)
Compile with PTXAS
Binary Codes
(7)
Provide Run-time Configurations
Underlying GPUs
Users
(8)

Debugging MPI with MPE

MPI Parallel Environment (MPE)

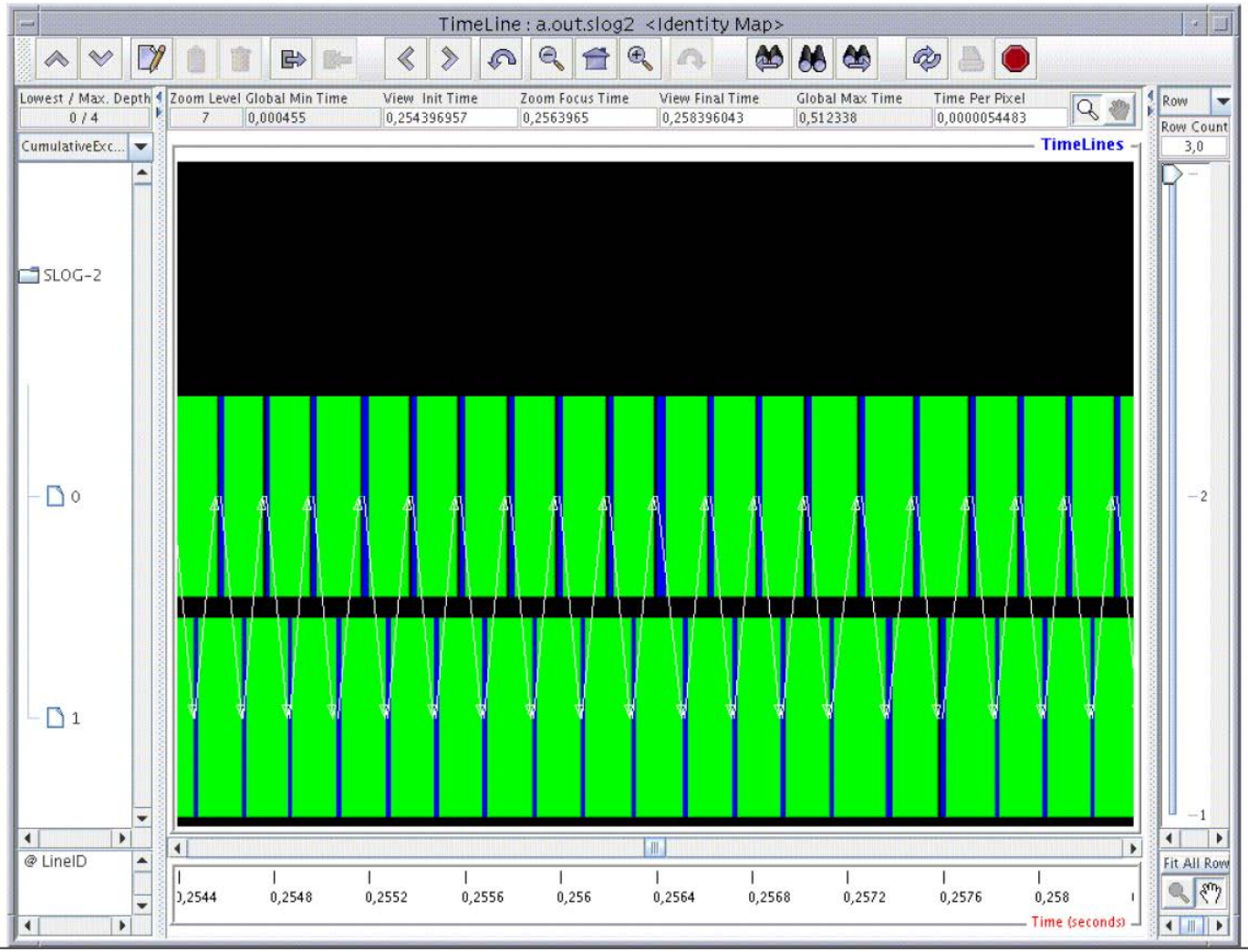
- Software package for MPI programmers
- Provides users with a number of useful tool
  - visualisation, log converters, tracers
- Documentation
  - http://www-unix.mcs.anl.gov/perfvis/download/index.htm
- Compile the MPI program with the mpecc wrapper
  - **-mpilog**: Automatic MPI and MPE user-defined states logging
  - **-mpitrace**: Trace MPI program with printf
  - **-mpianim**: Animate MPI program in real-time.
  - . . .
- Log file formats
  - ALOG (ASCII), CLOG (BINARY) maintained for compatibility reasons
  - SLOG = Scalable log

MPI Program Tracing

- **mpecc -mpilog mpi_latency.c**
- **mpirun –np 2 a.out**
  - Produces a **a.out.clog** trace file
  - Convert to SLOG format using **clogToslog2** program
- **mpirun -np 2 ./a.out**

  –MPDENV -MPE_LOG_FORMAT=SLOG
  - Produces a **a.out.slog2** file
- Open and visualize the **slogs** file with Jumpshot

Jumpshot Latency Program Snapshot



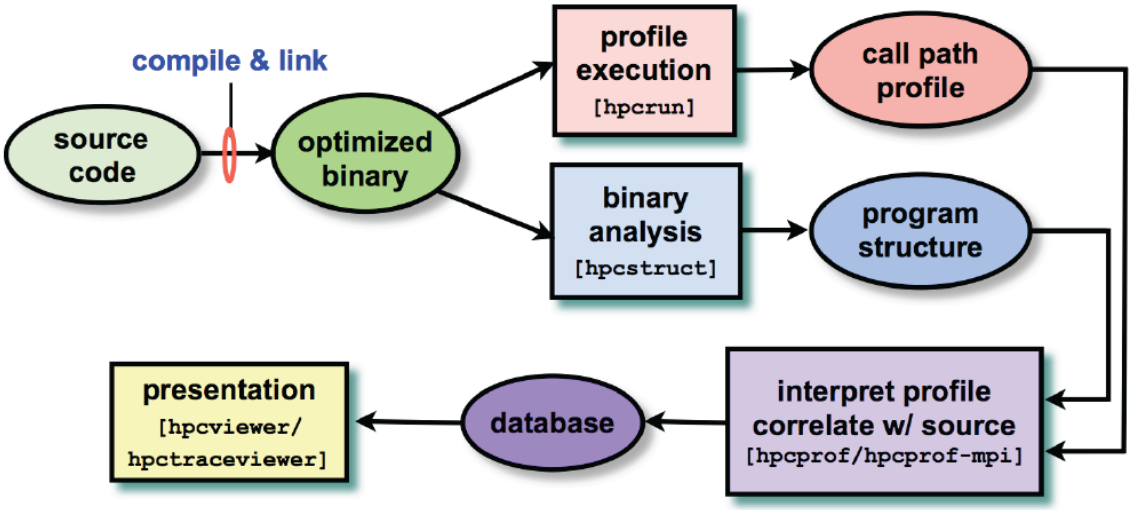
Let’s look at the parallel matrix multiplication example with MPE and Jumpshot on Discovery Cluster

- Performance Counters
- Profiling
  - PAPI
  - TAU
  - HPCToolkit

- **P**erformance **A**pplication **P**rogramming **I**nterface
- To design, standardize and implement a portable and efficient API
- C and Fortran Interface

```
[nroy@discovery2 utils]$ ./papi_avail
Available events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.3.0.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz (45)
CPU Revision             : 7.000000
CPUID Info               : Family: 6 Model: 45 Stepping: 7
CPU Max Megahertz        : 2599
CPU Min Megahertz        : 2599
Hdw Threads per core     : 2
Cores per Socket         : 8
Sockets                  : 2
NUMA Nodes               : 2
CPUs per Node            : 16
Total CPUs               : 32
Running in a VM          : no
Number Hardware Counters : 11
Max Multiplex Counters   : 32
--------------------------------------------------------------------------------

    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
PAPI_BTAC_M  0x8000001b  No    No   Branch target address cache misses
instructions
PAPI_FNV_INS 0x80000065  No    No   Floating point inverse instructions
PAPI_FP_OPS  0x80000066  Yes   Yes  Floating point operations
PAPI_SP_OPS  0x80000067  Yes   Yes  Floating point operations; optimized to count scaled single precision vector operations
PAPI_DP_OPS  0x80000068  Yes   Yes  Floating point operations; optimized to count scaled double precision vector operations
PAPI_VEC_SP  0x80000069  Yes   Yes  Single precision vector/SIMD instructions
PAPI_VEC_DP  0x8000006a  Yes   Yes  Double precision vector/SIMD instructions
PAPI_REF_CYC 0x8000006b  Yes   No   Reference clock cycles
--------------------------------------------------------------------------------
Of 108 possible events, 50 are available, of which 17 are derived.

avail.c                         PASSED
[nroy@discovery2 utils]$ pwd
/shared/apps/hpctoolkit/papi-5.3.0/src/utils
[nroy@discovery2 utils]$
```

- http://hpctoolkit.org
- Measurement and analysis of program performance
- Using statistical sampling of timers and hardware performance counters
- Platforms supported: Linux X86_64, Linux-x86, Linux-Power, Cray XT/XE/XK, IBM Blue Gene/Q, Blue Gene/P



hpcrun

collecting calling-context-sensitive performance measurements

hpcstruct

Analyzing application binaries and information between binaries and source codes

hpcprof

correlating the result with source code

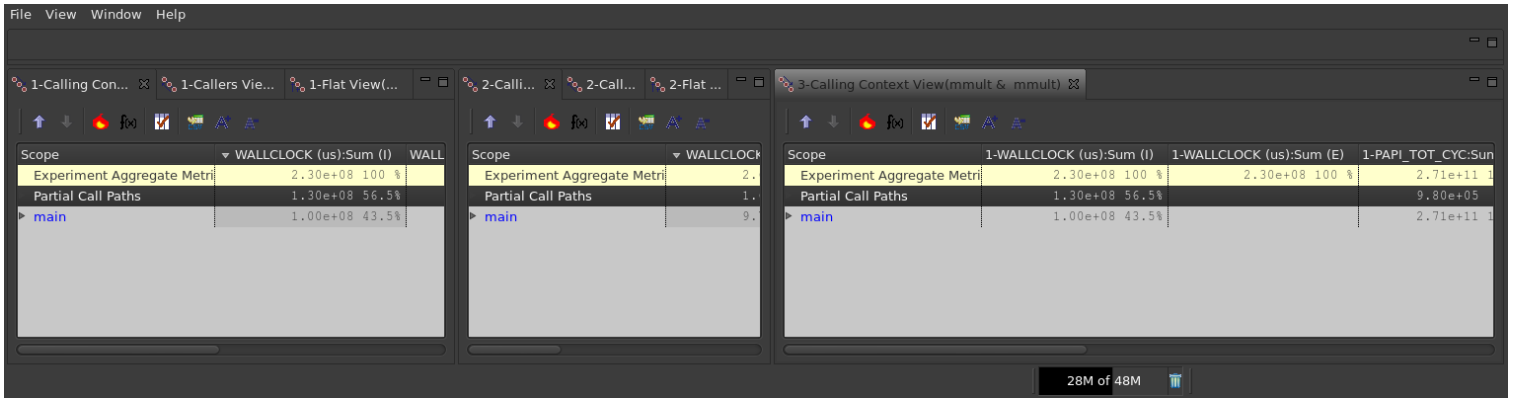
hpcviewer

graphical user interface that presents a hierarchical, time-centric view of a program execution

Weak/strong scaling

Example: HPCToolkit use in determining scalability of programs (and assigning blame on a function to function basis). Since now we are only interested in execution time and number of cycles taken, we use the following counters.

Counter name	Description	Period
PAPI_TOT_CYC	Total cycles	10000
WALLCLOCK	Wall clock time used by the process in microseconds	100000



- Let go to Discovery Cluster and run a example.

mpicc -g -O3 -o cpi_debug cpi.c -gdwarf-2

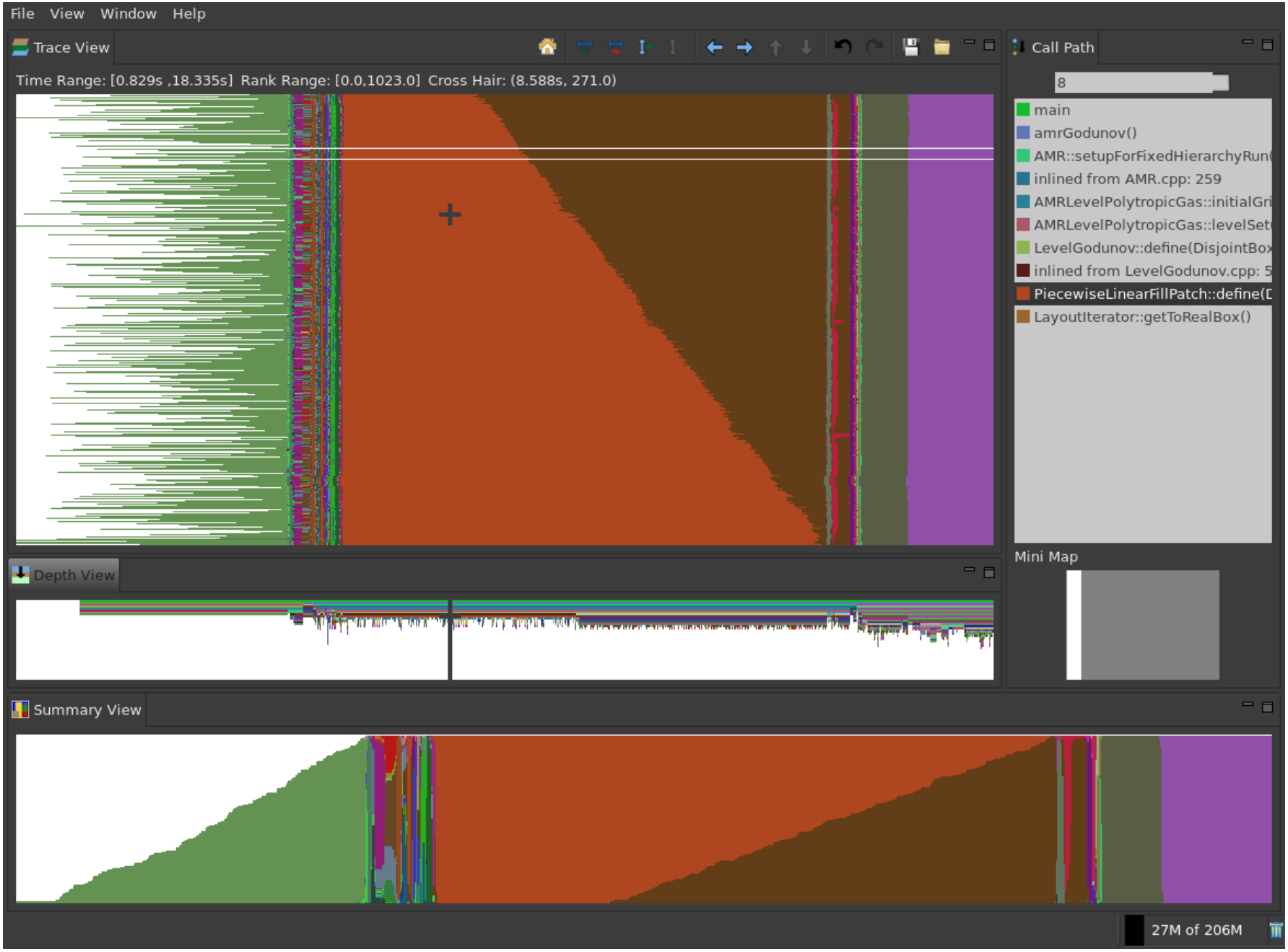
hpcstruct ./cpi_debug

bsub < bsubmit_debug.bash

hpcprof-mpi -S cpi_debug.hpcstruct -I ./'*' hpctoolkit-cpi_debug-measurements

hpctraceviewer hpctoolkit-cpi_debug-database

hpcviewer hpctoolkit-cpi_debug-database


File View Window Help
Trace View
Time Range: [0.829s ,18.335s] Rank Range: [0.0,1023.0] Cross Hair: (8.588s, 271.0)
Call Path
8
main
amrGodunov()
AMR::setupForFixedHierarchyRun(
inlined from AMR.cpp: 259
AMRLevelPolytropicGas::initialGri
AMRLevelPolytropicGas::levelSet(
LevelGodunov::define(DisjointBox
inlined from LevelGodunov.cpp: 5
PiecewiseLinearFillPatch::define(D
LayoutIterator::getToRealBox()
Mini Map
Depth View
Summary View
27M of 206M

THANK YOU
QUESTIONS?