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)
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 $20\times$, no matter how many processors are used.
Many Computational Science Modeling and Simulation Algorithms and Numerical Methods are Massively Parallel

Good Better Best
Platform Positioning – Application Characteristics

“Parallel behavior”
- Application scalability
- MPI vs OMP
- “Memory per core”

Auto/Aero
Data Analysis / Data Mining
Media & Entertainment
Electronics
Environment
HCLS
Petroleum

IO Bandwidth

Memory Intensive

Compute Intensive

Climate / Ocean
Reservoir Simulation
Weather
Selected CFD
Gene Sequencing & Assembly
Seismic Migration & Imaging
Games, DCC, Image Processing
General Seismic
Structure Based Drug Design
NVH, Structural & Thermal Analysis, Selected CFD

EDA
Bioinformatics
Crash
Data Analysis / Data Mining
PLM
24 Orders Magnitude of Spatial and Temporal Range
Platform Positioning - Scalability

Scale Up
(Uniform Memory Architecture)

Large SMP

Enterprise / Divisional

>4-way SMP

<4-way SMP

Blade Servers

Clustered SMP

Rack-optimized

Departmental / Workgroup

One CPU

Scale-out (Distributed Memory Architecture)

Hybrid x86/Cell BE

Capability

Capability

(Custom)

Thousands of CPUs
HPC Top500 Performance Evolution

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

HPC Application Spectrum

<table>
<thead>
<tr>
<th>Core Limited</th>
<th>Bandwidth Limited</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linpack</td>
<td>DAXPY</td>
</tr>
<tr>
<td>SPECfp2000</td>
<td>SparseMV</td>
</tr>
</tbody>
</table>
CPU design, scalability, memory, i/o, overall architectures

Hardware: CPU, Memory, I/O, Network
Software: O/S, Compilers, Libraries
Algorithm: Data structures, Data Locality, procedures

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

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

```c
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

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

```c
produce(ptr);
produce(NULL);
```

```c
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

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

• This is called “unsafe” because it depends on the availability of system buffers
MPI – Deadlock Solutions

- Order the operations more carefully:
  
<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

- Use non-blocking operations:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>
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

```
P1
P2
P3
P4
A
B
C
D
```

```
MPI_Allgather
```

```
P1
P2
P3
P4
A B C D
A B C D
A B C D
A B C D
```
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

<table>
<thead>
<tr>
<th>P1</th>
<th>A</th>
<th>...</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>B</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>P3</td>
<td>C</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>P4</td>
<td>D</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

![Diagram showing MPI_Reduce operation](image)
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

<table>
<thead>
<tr>
<th>P1</th>
<th>A</th>
<th>...</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>B</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>P3</td>
<td>C</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>P4</td>
<td>D</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

`MPI_Reduce`

```
P1
A+B+C+D
```

```
P2
A+B+C+D
```

```
P3
A+B+C+D
```

```
P4
A+B+C+D
```
Synchronization

- `int MPI_Barrier(MPI_Comm comm)`

```c
#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

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
</tbody>
</table>
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 source (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 extra 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:

  ```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
    - Send(1)
    - Recv(1)
  - Process 1
    - Send(0)
    - 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:
  ```c
  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:
  ```c
  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)

```c
#include "mpi.h"
#include <math.h>

int main(int argc, char *argv[]) {
    int done = 0, n, myid, numprocs, I, rc;
    double PI25DT = 3.141592653589793238462643;
    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)
            break;
    }
    MPI_Finalize();
}
```
Example: Compute PI (2)

```c
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

```cpp
#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

```c
#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 */

int 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");
    }
}
```
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

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

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)

Fork/Join can be nested
- Nesting complication handled “automagically” at compile-time
- Independent of the number of threads actually running

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...

Parallel Task I Parallel Task II Parallel Task III
Master Thread

Parallel Task I Parallel Task II Parallel Task III
Master Thread
OpenMP Loop Scheduling

The `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

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

// Static Scheduling
int chunk = 16/T;
int base = tid * chunk;
int bound = (tid+1)*chunk;
for( i=base; i<bound; i++ )
{
    dolteration(i);
}
Barrier();
```

```c
// Dynamic Scheduling
int current_i;
while( workLeftToDo() )
{
    current_i = getNextIter();
    dolteration(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 */
}
```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int nthreads, tid;
    INTERVALS;
    double n_1, x, pi = 0.0;

    n_1 = 1.0 / (double)INTERVALS;

    /* 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 */

    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);

    /* 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
#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[]) {

    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;

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

    /* Parameters of MPI. */
    intM_N;
    intM_ID;
    int rtn_val;
    char name[128];
    int namelen;

    /* Parameters of OpenMP. */
    int O_P;
    int O_T;
    int O_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
", 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_ID:%d
", name,M_ID,O_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
", 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
- OpenMP threads are heavy
  - Very good for handling parallel tasks
  - Not particularly remarkable at handling fine grain data parallelism (vector architectures excel here)

Code size grows only modestly

Expression of parallelism flows clearly, code is easy to read
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

NVIDIA CUDA 6.5 SDK
### 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".

### Intel Xeon Phi™ Product Family Specifications

<table>
<thead>
<tr>
<th>Product Number</th>
<th>Form Factor &amp; Thermal Solution</th>
<th>Board TDP (Watts)</th>
<th>Number of Cores</th>
<th>Frequency (GHz)</th>
<th>Peak Double Precision Performance (GFLOP)</th>
<th>Peak Memory Bandwidth (GB/s)</th>
<th>Memory Capacity (GB)</th>
<th>Intel Turbo Boost Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>3120P</td>
<td>PCIe, Passive</td>
<td>300</td>
<td>57</td>
<td>1.1</td>
<td>1003</td>
<td>240</td>
<td>6</td>
<td>N/A</td>
</tr>
<tr>
<td>3120A</td>
<td>PCIe, Active</td>
<td>300</td>
<td>57</td>
<td>1.1</td>
<td>1003</td>
<td>240</td>
<td>6</td>
<td>N/A</td>
</tr>
<tr>
<td>5110P</td>
<td>PCIe, Passive</td>
<td>225</td>
<td>60</td>
<td>1.053</td>
<td>1011</td>
<td>320</td>
<td>8</td>
<td>N/A</td>
</tr>
<tr>
<td>5120D</td>
<td>Dense form factor, None</td>
<td>215</td>
<td>60</td>
<td>1.053</td>
<td>1011</td>
<td>352</td>
<td>8</td>
<td>N/A</td>
</tr>
<tr>
<td>7110P</td>
<td>PCIe, Passive</td>
<td>300</td>
<td>61</td>
<td>1.238</td>
<td>1208</td>
<td>352</td>
<td>15</td>
<td>Peak turbo frequency: 1.33 GHz</td>
</tr>
<tr>
<td>7120X</td>
<td>PCIe, None</td>
<td>300</td>
<td>61</td>
<td>1.238</td>
<td>1208</td>
<td>352</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

**Intel Compilers and Intel Parallel Studio**

**AMD** line of GPUs supporting OpenCL.

Not so widely used in HPC at the moment.
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 Amdahl's 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

```c
//HMPP codelet
#pragma hmp ncols 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++){
    }
}
```

Native GPU code

- CUDA, CUDA-Fortran (PGI), OpenCL

  Pros
  - Requires most time

  Cons
  - 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

- **Compute Unified Device Architecture**
- 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
Qualifiers
  global, device, shared,
  local, constant, ...

Built-in variables
  threadIdx, blockIdx, ...

Intrinsics
  __syncthreads, __fmalu_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

Nilay’s CPU Analogy

- Nvidia/CUDA
  - CUDA Processor
  - CUDA Core
  - Streaming Multiprocessor
- AMD/OpenCL
  - Processing Element
  - SIMD Unit
  - Compute Unit
  - GPU Device

Lane
Pipeline
Core
Device
Thread Branching

Thread 0  Threads 1-31

Program

```c
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!

```c
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!

![Diagram showing memory hierarchy and bandwidth considerations](image-url)
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
__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 = {0,0,0,1,1,1,2,2,2,2}

__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 compute profiling 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

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 $u_{\text{new}}$ 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++)
    \[ u_{\text{new}}[i][j] = 0.0f - 0.25f \times (u[i-1][j] + u[i+1][j] \]
    \[ + u[i][j-1] + u[i][j+1]) \]

- Exchange halo with 2 4 neighbor
- Swap \texttt{u\_new} and \texttt{u}
- Next iteration
EXAMPLE: JACOBI SOLVER

- Solves the 2D-Laplace equation on a rectangle
  \[ \Delta u(x, y) = 0 \quad \forall \ (x, y) \in \Omega \setminus \delta \Omega \]
  
  - Dirichlet boundary conditions (constant values on boundaries)
  \[ u(x, y) = f(x, y) \in \delta \Omega \]

- 2D domain decomposition with \( n \times k \) 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_boundary, 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_boundary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
EXAMPLE: JACOBI - TOP/BOTTOM HALO UPDATE

```c
#pragma acc host_data use_device ( u_new ) {
    MPI_Ssendrecv (u_new+offseet first row) m-2, MPI_DOUBLE, t_nb, 0,
    u_new+offset bottom boundary, m-2, MPI_DOUBLE, b_nb, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Ssendrecv (u_new+offset last row) m-2, MPI_DOUBLE, b_nb, 1,
    u_new+offset top boundary, m-2, MPI_DOUBLE, t_nb, 1,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

OpenACC

```
MPI_Ssendrecv (u_new d+offset first row) m-2, MPI_DOUBLE, t_nb, 0,
    u_new d+offset bottom boundary, m-2, MPI_DOUBLE, b_nb, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Ssendrecv (u_new d+offset last row) m-2, MPI_DOUBLE, b_nb, 1,
    u_new d+offset top boundary, m-2, MPI_DOUBLE, t_nb, 1,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

CUDA
EXAMPLE: JACOBI - LEFT/RIGHT HALO UPDATE

```c
//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, 1_nb, 0,
                  from_left, n-2, MPI_DOUBLE, 1_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

```c
//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

```
#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_boundary, 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_boundary, 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] )
//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_boundary, 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

Runtime (s)

#MPI Ranks - 1 CPU Socket with 10 OMP Threads or 1 GPU per Rank

- Tesla K20X
- Xeon E5-2690 v2 @ 3.0Ghz
With UVA and CUDA-aware MPI

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

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

No UVA and regular MPI

```c
//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

Diagram showing peer-to-peer transfers between two GPUs with memory connections and system memory.
NVIDIA GPUDIRECT™ SUPPORT FOR RDMA
CUDA-AWARE MPI

MPI GPU TO REMOTE GPU
GPUDIRECT SUPPORT FOR RDMA

```c
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);
```
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

memcpy D->H | MPI_Sendrecv | memcpy H->D
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

```c
#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_boundary, 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_boundary, m-2, MPI_DOUBLE, t_nb, 1,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

MPI_Request t_b_req[4];
#pragma acc host_data use_device ( u_new ) {
    MPI_Irecv(u_new+offset_top_boundry, m-2, MPI_DOUBLE, t_b_req);
    MPI_Irecv(u_new+offset_bottom_boundray, m-2, MPI_DOUBLE, t_b_req+1);
    MPI_Isend(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1, MPI_COMM_WORLD, t_b_req+2);
    MPI_Isend(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 1, MPI_COMM_WORLD, t_b_req+3);
}

MPI_Waitall(4, t_b_req, MPI_STATUSES_IGNORE);
```

Gives MPI more opportunities to build efficient piplines

![Runtime comparison graph](chart.png)
OVERLAPPING COMMUNICATION AND COMPUTATION

CUDA

#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() //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

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 == 1) {
      sleep(5);
    }
  }

  > mpiexec -np 2 ./jacobi_mpi+cuda

  rank 0: pid 30034 on gel07 ready for attach
  > ssh gel07
  cuda-gdb --pid 30034
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
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
- 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

  **Performance** **Application** **Programming** **Interface**

- To design, standardize and implement a portable and efficient API
- C and Fortran Interface
Available events and hardware information.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Avail</th>
<th>Deriv</th>
<th>Description (Note)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>0x80000000</td>
<td>Yes</td>
<td>No</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>0x80000001</td>
<td>Yes</td>
<td>No</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>0x80000002</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>0x80000003</td>
<td>Yes</td>
<td>No</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L3_DCM</td>
<td>0x80000004</td>
<td>No</td>
<td>No</td>
<td>Level 3 data cache misses</td>
</tr>
<tr>
<td>PAPI_L3_ICM</td>
<td>0x80000005</td>
<td>No</td>
<td>No</td>
<td>Level 3 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L1_TCM</td>
<td>0x80000006</td>
<td>Yes</td>
<td>Yes</td>
<td>Level 1 cache misses</td>
</tr>
<tr>
<td>PAPI_L2_TCM</td>
<td>0x80000007</td>
<td>Yes</td>
<td>No</td>
<td>Level 2 cache misses</td>
</tr>
<tr>
<td>PAPI_L3_TCM</td>
<td>0x80000008</td>
<td>Yes</td>
<td>No</td>
<td>Level 3 cache misses</td>
</tr>
<tr>
<td>PAPI_BTAC_M</td>
<td>0x8000001b</td>
<td>No</td>
<td>No</td>
<td>Branch target address cache misses instructions</td>
</tr>
<tr>
<td>PAPI_FNV_INS</td>
<td>0x80000065</td>
<td>No</td>
<td>No</td>
<td>Floating point inverse instructions</td>
</tr>
<tr>
<td>PAPI_FP_OPS</td>
<td>0x80000066</td>
<td>Yes</td>
<td>Yes</td>
<td>Floating point operations</td>
</tr>
<tr>
<td>PAPI_SP_OPS</td>
<td>0x80000067</td>
<td>Yes</td>
<td>Yes</td>
<td>Floating point operations; optimized to count scaled single precision vector operations</td>
</tr>
<tr>
<td>PAPI_DP_OPS</td>
<td>0x80000068</td>
<td>Yes</td>
<td>Yes</td>
<td>Floating point operations; optimized to count scaled double precision vector operations</td>
</tr>
<tr>
<td>PAPI_VEC_SP</td>
<td>0x80000069</td>
<td>Yes</td>
<td>Yes</td>
<td>Single precision vector/SIMD instructions</td>
</tr>
<tr>
<td>PAPI_VEC_DP</td>
<td>0x8000006a</td>
<td>Yes</td>
<td>Yes</td>
<td>Double precision vector/SIMD instructions</td>
</tr>
<tr>
<td>PAPI_REF_CYC</td>
<td>0x8000006b</td>
<td>No</td>
<td>Yes</td>
<td>Reference clock cycles</td>
</tr>
</tbody>
</table>

Of 108 possible events, 50 are available, of which 17 are derived.

| avail.c | PASSED |
- 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
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.

<table>
<thead>
<tr>
<th>Counter name</th>
<th>Description</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_TOT_CYC</td>
<td>Total cycles</td>
<td>10000</td>
</tr>
<tr>
<td>WALLCLOCK</td>
<td>Wall clock time used by the process in microseconds</td>
<td>100000</td>
</tr>
</tbody>
</table>
• 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 -l ./"*" hpctoolkit-cpi_debug-measurements
hpctraceviewer hpctoolkit-cpi_debug-database
hpcviewer hpctoolkit-cpi_debug-database
THANK YOU QUESTIONS?