Parallel Debugging on Discovery Cluster from pthreads and openmp to GPGPU, MPI and RDMA..

The session is structured as follows:

• pthreads and OpenMP programs – debugging issues and differences in the two methods
• Debugging pthreads and OpenMP programs using TotalView on the Discovery Cluster – several examples
• MPI and Hybrid MPI with OpenMP and GPGPU – debugging issues
• Debugging MPI and Hybrid MPI / OpenMP and GPGPU codes with TotalView, Nsight, cuda-gdb and cuda-memcheck on Discovery Cluster – several examples
• Debugging MPI - RDMA aware code on Discovery Cluster – several examples
• Profiling and tracing techniques to implement scalability and thread safety and as an aid to debugging
• Several examples of profiling and tracing various parallel codes on Discovery Cluster
• Load balancing and synchronization – example debugging runs on Discovery cluster and issues related with race conditions, deadlocks and non-determinism in parallel code

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

Threads
Accessing shared data
Critical sections
Shared memory multiprocessor system

*Single address space* exists – each memory location given unique address within single range of addresses.

Any memory location can be accessible by any of the processors.

Multicore processors are of this type.

Also multiprocessor servers have both multicore processors and multiple such processors.
Programming a Shared Memory System

Generally, more convenient and efficient than message passing.

Can take advantage of shared memory for holding data rather than explicit message passing to share data.

However access to shared data by different processors needs to be carefully controlled usually explicitly by programmer.

Shared memory systems have been around for a long time but with the advent of multi-core systems, it has become very important to able to program for them
Methods for Programming Shared Memory Multiprocessors

1. Using heavyweight processes.

2. Using threads explicitly - e.g. Pthreads, Java threads

3. Using a sequential programming language such as C supplemented with compiler directives and libraries for specifying parallelism. e.g. OpenMP. Underlying mechanism on OpenMP is thread-based.

4. Using a “parallel programming” language, e.g. Ada, UPC - not popular.

We will look mostly at thread API’s and OpenMP
Using Heavyweight Processes

Operating systems often based upon notion of a process.

Processor time shares between processes, switching from one process to another. Might occur at regular intervals or when an active process becomes delayed.

Offers opportunity to de-schedule processes blocked from proceeding for some reason, e.g. waiting for an I/O operation to complete.

Concept could be used for parallel programming. Not much used because of overhead but fork/join concepts used elsewhere.
(UNIX) FORK-JOIN construct

Fork here creates a complete copy of the main program and starts it at the same place as the Fork.

Both programs continue together.
UNIX System Calls

No join routine – use `exit()` to exit from process and `wait()` to wait for slave to complete:

```c
... pid = fork();
if (pid == 0) {
    // code to be executed by child
} else {
    // code to be executed by parent
}
if (pid == 0) exit(0); else wait (0);
...
```
Differences between a process and threads

“heavyweight” process - completely separate program with its own variables, stack, and memory allocation.

Threads - shares the same memory space and global variables between routines.
Pthreads

IEEE Portable Operating System Interface, POSIX standard.

Executing a Pthread Thread

Main program

```c
pthread_create(&thread1, NULL, proc1, &arg);

thread1
{
    proc1(&arg)
    return(*status);
}

pthread_join(thread1, *status);
```
Detached Threads

It may be that thread are not bothered when a thread it creates terminates and then a join not needed.

Threads not joined are called detached threads.

When detached threads terminate, they are destroyed and their resource released.
Pthreads Detached Threads

Main program

pthread_create();

Parameter (attribute) specifies a detached thread

Thread

pthread_create();

Thread

pthread_create();

Thread

Termination

Termination

Termination
Issues in writing shared memory programs
Interleaved Statements

Instructions of processes/threads interleaved in time.

Example

<table>
<thead>
<tr>
<th>Process/Thread 1</th>
<th>Process/Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1.1</td>
<td>Instruction 2.1</td>
</tr>
<tr>
<td>Instruction 1.2</td>
<td>Instruction 2.2</td>
</tr>
<tr>
<td>Instruction 1.3</td>
<td>Instruction 2.3</td>
</tr>
</tbody>
</table>

Many possible orderings, e.g.:

Instruction 1.1  
Instruction 1.2  
Instruction 2.1  
Instruction 1.3  
Instruction 2.2  
Instruction 2.3  

Each process/thread must achieve the desired results irrespective of the interleaving order assuming instructions cannot be divided into smaller steps.
Thread-Safe Routines

*Thread safe* if routine can be called from multiple threads simultaneously and always produce correct results.

Standard I/O thread safe (prints messages without interleaving the characters).

System routines that return time *may not be thread safe*.

Routines that access shared data may require special care to be made thread safe.
Re-ordering code

Static re-ordering - Compilers may re-order code during compilation and prior to execution of code.

Dynamic re-ordering - Processors may re-order code during execution.

In both cases, objective is to best utilize available computer resources and minimize execution time.
Compiler/Processor Optimizations

Compiler and processor reorder instructions to improve performance.

Example

Suppose one had the code:

\[
\begin{align*}
    a &= b + 5; \\
    x &= y * 4; \\
    p &= x + 9;
\end{align*}
\]

and processor can perform, as is usual, multiple arithmetic operations at the same time. Can reorder to:

\[
\begin{align*}
    x &= y * 4; \\
    a &= b + 5; \\
    p &= x + 9;
\end{align*}
\]

and still be logically correct. This gives multiply operation longer time to complete before result \(x\) is needed in last statement. Very common for processors to execute machines instructions “out of program order” for increased speed.
## Accessing Shared Data

Accessing shared data needs careful control.

Consider two processes each of which is to add one to a shared data item, \(x\).

Location \(x\) is read, \(x + 1\) computed, and the result written back to the location:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = x + 1;)</td>
<td>read (x)</td>
<td>read (x)</td>
</tr>
<tr>
<td></td>
<td>compute (x + 1)</td>
<td>compute (x + 1)</td>
</tr>
<tr>
<td></td>
<td>write to (x)</td>
<td>write to (x)</td>
</tr>
</tbody>
</table>
### Instruction

<table>
<thead>
<tr>
<th>x = x + 1;</th>
<th>Process/thread 1</th>
<th>Process/thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read x</td>
<td>read x</td>
<td></td>
</tr>
<tr>
<td>compute x + 1</td>
<td>compute x + 1</td>
<td></td>
</tr>
<tr>
<td>write to x</td>
<td>write to x</td>
<td></td>
</tr>
</tbody>
</table>

Get $x = x + 2$ finally.

### Instruction

<table>
<thead>
<tr>
<th>x = x + 1;</th>
<th>Process/thread 1</th>
<th>Process/thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read x</td>
<td>read x</td>
<td></td>
</tr>
<tr>
<td>compute x + 1</td>
<td>compute x + 1</td>
<td></td>
</tr>
<tr>
<td>write to x</td>
<td>write to x</td>
<td></td>
</tr>
</tbody>
</table>

Get $x = x + 1$ finally.
Critical Section

A mechanism for ensuring that only one process accesses a particular resource at a time.

critical section – a section of code for accessing resource

Arrange that only one such critical section is executed at a time.

This mechanism is known as mutual exclusion.

Concept also appears in an operating systems.
Locks

Simplest mechanism for ensuring mutual exclusion of critical sections.

A lock - a 1-bit variable that is a 1 to indicate that a process has entered the critical section and a 0 to indicate that no process is in the critical section.

Operates much like that of a door lock:

A process coming to the “door” of a critical section and finding it open may enter the critical section, locking the door behind it to prevent other processes from entering. Once the process has finished the critical section, it unlocks the door and leaves.
Control of critical sections through busy waiting

Must be indivisible

Process 1

while (lock == 1) do_nothing;
lock = 1;

Critical section

lock = 0;

Process 2

while (lock == 1) do_nothing;

Better to deschedule process

lock = 1;

Critical section

lock = 0;
Pthreads

Lock routines

Locks implemented in Pthreads with *mutually exclusive lock variables*, or “mutex” variables:

```c
...  
pthread_mutex_lock(&mutex1);
    critical section
    pthread_mutex_unlock(&mutex1);
...  
```

If a thread reaches mutex lock and finds it locked, it will wait for lock to open. If more than one thread waiting for lock to open, when it does open, system selects one thread to be allowed to proceed. Only thread that locks a mutex can unlock it.
Condition Variables

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

With locks, the global variable would need to be examined at frequent intervals (“polled”) within a critical section.

Very time-consuming and unproductive exercise.

Can be overcome by introducing so-called *condition variables.*
Pthread Condition Variables

Pthreads arrangement for signal and wait:

```c
action()
{
    ...
    pthread_mutex_lock(&mutex1);
    while (c <> 0)
        pthread_cond_wait(cond1,mutex1);
    pthread_mutex_unlock(&mutex1);
    take_action();
}

counter()
{
    ...
    pthread_mutex_lock(&mutex1);
    c--; 
    if (c == 0) pthread_cond_signal(cond1);
    pthread_mutex_unlock(&mutex1);
}
```

Notes:
Signals *not* remembered - threads must already be waiting for a signal to receive it. `pthread_cond_wait()` unlocks `mutex1` so that it can be used other thread and relocks it after woken up. Value of `c` checked in both threads.
Critical Sections Serializing Code

High performance programs should have as few as possible critical sections as their use can serialize the code.

Suppose, all processes happen to come to their critical section together.

They will execute their critical sections one after the other.

In that situation, the execution time becomes almost that of a single processor.
Illustration

When $t_{\text{comp}} < p t_{\text{crit}}$, less than $p$ processor will be active.
Deadlock

Can occur with two processes when one requires a resource held by the other, and this process requires a resource held by the first process.

Two-process deadlock
Deadlock

Deadlock can also occur in a circular fashion with several processes having a resource wanted by another.
Pthreads

pthtrylock routine

Offers one routine that can test whether a lock is actually closed without blocking the thread:

```
pthread_mutex_trylock()
```

Will lock an unlocked mutex and return 0 or will return with **EBUSY** if the mutex is already locked – might find a use in overcoming deadlock.
Semaphores

A positive integer (including zero) operated upon by two operations:

**P operation on semaphore s**

Waits until $s$ is greater than zero and then decrements $s$ by one and allows the process to continue.

**V operation on semaphore s**

Increments $s$ by one and releases one of the waiting processes (if any).
\textbf{P} and \textbf{V} operations are performed indivisibly.

Mechanism for activating waiting processes implicit in \textbf{P} and \textbf{V} operations. Though exact algorithm not specified, algorithm expected to be fair.

Processes delayed by \textbf{P}(s) are kept in abeyance until released by a \textbf{V}(s) on the same semaphore.

Devised by Dijkstra in 1968.

Letter \textbf{P} from Dutch word \textit{passeren}, meaning “to pass”
Letter \textbf{V} from Dutch word \textit{vrijgeven}, meaning “to release”
Mutual exclusion of critical sections can be achieved with one semaphore having the value 0 or 1 (a binary semaphore), which acts as a lock variable, but P and V operations include a process scheduling mechanism:

**Process 1**

Noncritical section

...  
P(s)  
Critical section

V(s)

...  
Noncritical section

**Process 2**

Noncritical section

...  
P(s)  
Critical section

V(s)

...  
Noncritical section

**Process 3**

Noncritical section

...  
P(s)  
Critical section

V(s)

...  
Noncritical section
General semaphore (or counting semaphore)

Can take on positive values other than zero and one.

Provide, for example, a means of recording number of “resource units” available or used. Can solve producer/consumer problems - more on that in operating system courses.

Semaphore routines exist for UNIX processes.

Does not exist in Pthreads as such, though they can be written.

Do exist in real-time extension to Pthreads.
Monitor

Suite of procedures that provides only way to access shared resource. **Only one process can use a monitor procedure at any instant.**

Could be implemented using a semaphore or lock to protect entry, i.e.:

```c
monitor_proc1() {
    lock(x);

    monitor body

    unlock(x);
    return;
}
```

A version of a monitor exists in Java threads, see later.
Program example

To sum the elements of an array, \texttt{a[1000]}:

\begin{verbatim}
int sum, a[1000];
sum = 0;
for (i = 0; i < 1000; i++)
    sum = sum + a[i];
\end{verbatim}
Pthreads program example

$n$ threads created, each taking numbers from list to add to their local partial sums. When all numbers taken, threads can add their partial sums to a shared location `sum`.

Shared location `global_index` used by each thread to select next element of `a[]`. After `global_index` read, it is incremented in preparation for next element to be read.
```c
#include <stdio.h>
#include <pthread.h>

#define ARRAY_SIZE 1000
#define NO_THREADS 10

int n[ARRAY_SIZE];
int global_index = 0;
int sum = 0; /* final result, also used by slaves */

void *slave(void *ignored)
{
    int local_index, partial_sum = 0;
    do {
        pthread_mutex_lock(&mutex1);
        local_index = global_index;
        global_index++;
        pthread_mutex_unlock(&mutex1);

        if (local_index < ARRAY_SIZE) partial_sum += *(n + local_index);
    } while (local_index < ARRAY_SIZE);

    pthread_mutex_lock(&mutex1); /* add partial sum to global sum */
    sum += partial_sum;
    pthread_mutex_unlock(&mutex1);

    return NULL; /* Thread exits */
}

main()
{
    int i;
    pthread_t thread[10];
    pthread_mutex_init(&mutex1, NULL);

    for (i = 0; i < NO_THREADS; i++) /* create threads */
        o[i] = i + 1;

    for (i = 0; i < NO_THREADS; i++) /* create threads */
        if (pthread_create(&thread[i], NULL, slave, NULL) != 0) perror("Thread creation fails");

    for (i = 0; i < NO_THREADS; i++)
        if (pthread_join(thread[i], NULL) != 0) perror("Thread join fails");
    printf("The sum of 1 to %d is %d\n", ARRAY_SIZE, sum);
}
```

TotalView supports most HPC parallel programming models:

- MPI
- Threads
- OpenMP
- PVM
- SHMEM
- Fork/exec
- Hybrid

**TotalView P/T Groups:**

- Process/Thread (P/T) groups are a TotalView fabrication. Their purpose is to organize processes and threads into associations that a user can operate on.

- Dynamic membership: TotalView automatically creates these P/T groups and places processes and threads in them as they are created.

- **Motivation:** TotalView commands typically act upon a specific P/T group. **It is important for parallel program users to know which P/T group is being acted upon!**

**User-defined P/T Groups:**

- In most cases, the default TotalView P/T groups are sufficient - however...
  - TotalView provides a way for users to create their own P/T groups.
  - Non-trivial and not covered here.

**TotalView’s P/T groups are described very well in the "TotalView User Guide".**

**Types of P/T Groups:**

- **Control Group:**
  - Contains all processes and threads created by the program across all processors

- **Share Group:**
  - Contains all of the processes and their threads, that are running the same executable
  - A program may have multiple Share Groups. For example all processes executing a.out would be in one Share Group, and all processes executing b.out would be in another Share Group

- **Workers Group:**
  - Contains all threads that are executing user code
  - May span multiple process Share Groups
  - Does not contain kernel-level manager threads

- **Lockstep Group:**
  - Includes all threads in a Share Group that are at the same PC (program counter) address
  - A subset of the Workers Group
  - Only valid for stopped threads - meaningless otherwise
Selecting P/T Groups:

- When you select a P/T group, you are telling TotalView which set of processes and threads to act upon.
- You can select any of the available predefined P/T groups. The default is Control Group.
- Always relative to the Thread-of-Interest (TOI) and the Process-of-Interest (POI), which are the thread and process being viewed in the current Process Window.
- P/T groups can be selected from the Process Window’s P/T Selection menu as shown below.

<table>
<thead>
<tr>
<th>P/T Selection</th>
<th>What is affected by any execution Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group (Control)</td>
<td>Default. All processes and their threads</td>
</tr>
<tr>
<td>Group (Share)</td>
<td>All processes and their threads that are in the same share group as the POI (process-of-interest)</td>
</tr>
<tr>
<td>Group (Workers)</td>
<td>All threads that are executing user code</td>
</tr>
<tr>
<td>Group (Lockstep)</td>
<td>All user threads that are stopped at the same PC</td>
</tr>
<tr>
<td>Rank 0</td>
<td>Only the POI and its threads. In the above example, the POI happens to have an MPI rank of 0</td>
</tr>
<tr>
<td>Process (Workers)</td>
<td>User threads in the POI</td>
</tr>
<tr>
<td>Process (Lockstep)</td>
<td>User threads stopped at the same PC in the POI</td>
</tr>
<tr>
<td>Thread 2.1</td>
<td>Only the TOI (thread-of-interest). In the above example, the TOI happens to be 2.1</td>
</tr>
</tbody>
</table>

- P/T groups can also be selected from other locations, such as the Evaluate Dialog Box:

Important:

- For most users (especially new users), just accepting the TotalView default Control P/T group does the trick.
- There is quite a bit more to TotalView’s P/T groups than what is described above. See the TotalView documentation for details.
General Threads Model:

- Most operating systems support programs that have multiple threads of execution. Although implementations differ, they usually possess the following common characteristics:
  - Shared address space - threads can read/write the same variables and execute the same code.
  - Private execution context - every thread has its own set of registers
  - Private execution stack - every thread has address space reserved for its stack
  - Thread - process association - threads exist within and use the resources of a process. They cannot exist outside of a process.

- The diagram below depicts the general threads model. TotalView follows this general model.

Supported Platforms:

- TotalView supports debugging threaded applications on all of its supported platforms.

Important Differences:

- Threads are implemented differently by different operating systems. Also, different versions of the same operating system may differ in the way threads are handled.

- Because of this, some thread behavior within TotalView is both architecture and software version dependent:
  - Not all features are implemented, or implemented identically on all platforms
  - Patches and/or upgrades to the OS and other software may be required
  - Hardware requirements vary between platforms (minimum disk, memory, etc.)
  - Restrictions and known problems vary between platforms
Finding Thread Information

Root Window:

- In order to view thread information in the Root Window, you must first click on the "expand toggle" that appears next to the process.
- A list of threads associated with the process will then appear showing one line of information for each thread (below).

![](image)

- Note that the list of threads may contain both user threads and system threads. User threads are created and managed by your program, whereas system threads are not. The latter should be ignored (don’t try to debug them).

Process Window:

- Most of what TotalView knows about a thread is able to be found in the Process Window’s pane.
  1. Status Bar: Show status information for the selected thread and its associated process.
  2. Stack Trace Pane: Displays the call stack of routines that the selected thread is executing.
  3. Stack Frame Pane: Shows selected thread’s stack variables, registers, etc.
  4. Source Pane: Shows the source code for the selected thread.
  5. Threads Pane: Shows threads associated with the selected process.
Selecting a Thread

- By Diving:
  - After selecting a thread in either the Root Window or the Process Window Threads Pane, you can dive on it by three different methods:
    - Double left clicking
    - Right clicking and then selecting Dive from the pop-up menu
    - Selecting Dive from the Root Window's View Menu.
  - That thread's information will then be displayed in the current Process Window.
  - To force a new Process Window for a thread, use Dive in New Window from the View Menu or pop-up menu. Multiple Process Windows, one for each thread, can be created this way.

- By Thread Navigation Buttons:
  - Use the thread navigation control buttons (below) located in the bottom right corner of the Process Window.
  - "Cycle-through" the threads until the desired thread's information fills the Process Window.
Differentiating Threads:

- Debugging multi-threaded programs can be confusing, especially if you've opened multiple Process Windows for the different threads. TotalView provides two easy ways for you to differentiate threads from each other.

- Every thread has a unique "Thread ID" number assigned by TotalView. The TID appears in several locations, such as the Root Window, Process Window Threads Pane and Process Window Status Bar.

- Different threads are given different pane "trims", as shown below.

- The examples below demonstrate how threads are differentiated from each other as just described.

Threads pane shows all threads for the process.

Root Window toggle to show/hide threads for a process.

Different pane trim and thread identifiers.
Execution Control for Threaded Programs

Three Scopes of Influence:

- Depending upon the type of parallel application, TotalView can provide up to three different levels of control for thread execution commands. The table below describes these.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
</table>
| Group  | Typically used for multi-process, multi-threaded codes  
Execution commands apply to all threads in all processes  
**PATH:** Process Window > Group Menu |
| Process| Typically used for a multi-threaded process  
Applies to all threads in a single process  
**PATH:** Process Window > Process Menu |
| Thread | Applies to a single thread within a single process  
**PATH:** Process Window > Thread Menu  
Note that the thread specific execution control commands are not available on all platforms. They will appear to be dimmed in the menu if they are not available on the platform you are using. |

Synchronous vs. Asynchronous:

- **Synchronous**: if one thread in a process runs/stop, all threads must do likewise.
- **Asynchronous**: threads within a process can run/stop independently of each other.
- Platforms may differ in the way individual threads can be stopped and made to run.

For asynchronous thread control, unexpected program behavior (like hanging) can occur if some threads step or run while others are stopped - particularly in library routines. **CTRL-C** may be able to be used to cancel the command that caused the hang.

Thread-specific Breakpoints:

- Normally, all threads in a process stop when any one of them encounters a breakpoint.
- Thread-specific breakpoints are implemented through evaluation points and the use of TotalView expressions that include intrinsic variables and built-in statements.
- For example, the following expression will cause the process to stop only when thread 3 encounters it as part of an evaluation point:
Viewing and Modifying Thread Data

**Laminated Variables:**

- Often times in a parallel program, the same variable will have multiple instances across threads and/or processes. In such cases, it is frequently desirable to view all occurrences simultaneously.

- TotalView provides a way for you to do this by "laminating" the variable. Laminating a variable means to display all occurrences simultaneously in a Variable Window.

- Laminated variables can include scalars, arrays, structures and pointers.

- TotalView also enables you to edit laminated variables - either collectively (same value applies to all instances) or individually.

**Method 1:** Right click on the variable and select "Across Threads" from the pop-up menu. A new Variable Window will appear showing the laminated variable (example below).

**Method 2:** Dike on the variable so that it appears in a new Variable Window. Then:

```
PATH Variable Window > View Menu > Show Across > Thread
```

- Example of a laminated variable. Note that when laminating a variable, not all threads may be at a point in the program yet where the variable has a value. In such cases, the "Has no matching call frame" message will appear.

```
<table>
<thead>
<tr>
<th>Thread</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>(4912511008000)</td>
</tr>
<tr>
<td>1.2</td>
<td>(49125188009000)</td>
</tr>
<tr>
<td>1.3</td>
<td>(49125198810000)</td>
</tr>
<tr>
<td>1.4</td>
<td>(49125210818000)</td>
</tr>
<tr>
<td>1.5</td>
<td>(49125240828000)</td>
</tr>
</tbody>
</table>
```

- After laminating a variable, you can return to the non-laminated view by:

```
PATH Variable Window > View Menu > Show Across > None
```

**In the Kernel:**

- The Process Window below shows what can happen when a thread calls a system kernel routine. The debugger may not have full access to thread state information when it executes within the kernel. There's not much you can do at this point, debugging wise.
Programming with Shared Memory

Introduction to OpenMP
OpenMP

An accepted standard developed in the late 1990s by a group of industry specialists.

Consists of a small set of:

- Compiler directives,
- Library routines, and
- Environment variables

using the base language Fortran and C/C++.

Several compilers available to compile OpenMP programs include recent Linux C compilers.
Overview

OpenMP language extensions

- parallel control structures
- work sharing
- data environment
- synchronization
- runtime functions, env. variables

- governs flow of control in the program
- parallel directive
- distributed work among threads
  - do/parallel do and section directives
- scopes variables
  - shared and private clauses
- coordinates thread execution
  - critical and atomic directives
  - barrier directive
- runtime environment
  - omp_set_num_threads()
  - omp_get_thread_num()
  - OMP_NUM_THREADS
  - OMP_SCHEDULE
OpenMP

• Uses a thread-based shared memory programming model

• OpenMP programs will create multiple threads with “fork-join” model

• All threads have access to global memory

• Data can be shared among all threads or private to one thread

• Synchronization occurs but often implicit
Initially, single thread executed by a master thread.

*parallel* directive creates team of threads with a specified block of code executed by multiple threads in parallel.

Exact number of threads determined by one of several ways.

Other directives used within a *parallel* construct to specify parallel for loops and different blocks of code for threads.
OpenMP Compiler Directives

For C/C++, uses `#pragma` statements (“Pragmatic” directives). Format:

```plaintext
#pragma omp directive_name ...
```

where `omp` is an OpenMP keyword.

May be additional parameters (clauses) after directive name for different options.

Some directives require code to specified in a structured block that follows directive and then directive and structured block form a “construct”.

Parallel Directive

#pragma omp parallel
structured_block

directive creates multiple threads, each one executing the
specified structured_block, (a single statement or a
compound statement created with { ...} with a single
entry point and a single exit point.)

Implicit barrier at end of construct.
Hello world example

```c
#pragma omp parallel
{
    printf("Hello World from thread = %d\n", omp_get_thread_num(), omp_get_num_threads());
}
```

Output from an 8-processor/core machine:

Hello World from thread 0 of 8
Hello World from thread 4 of 8
Hello World from thread 3 of 8
Hello World from thread 2 of 8
Hello World from thread 7 of 8
Hello World from thread 1 of 8
Hello World from thread 6 of 8
Hello World from thread 5 of 8
Private and shared variables

Variables could be declared within each parallel region but OpenMP provides \texttt{private} clause.

```c
int tid;
...
#pragma omp parallel private(tid)
{
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
}
```

Each thread has a local variable \texttt{tid}

Also a \texttt{shared} clause available.
#pragma omp parallel private(x, num_threads)
{
    x = omp_get_thread_num();
    num_threads = omp_get_num_threads();
    a[x] = 10*num_threads;
}

Two library routines

omp_get_num_threads() returns number of threads that are currently being used in parallel directive
omp_get_thread_num() returns thread number (an integer from 0 to omp_get_num_threads() - 1 where thread 0 is the master thread).
Array a[] is a global array, and x and num_threads are declared as private to the threads.
Number of threads in a team

Established by either:

1. **num_threads** clause after the **parallel** directive, or

2. **omp_set_num_threads()** library routine being previously called, or

3. Environment variable **OMP_NUM_THREADS** is defined in order given or is system dependent if none of above.

Number of threads available can also be altered dynamically to achieve best use of system resources.
Work-Sharing

Three constructs in this classification:

sections
for
single

In all cases, implicit barrier at end of construct unless a nowait clause included, which overrides the barrier.

Note: These constructs do not start a new team of threads. That done by an enclosing parallel construct.
The construct

```c
#pragma omp sections
{
    #pragma omp section
    structured_block
    ...
    ...
    ...
    #pragma omp section
    structured_block
}
```

cause structured blocks to be shared among threads in team. The first `section` directive optional.
```c
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid = omp_get_thread_num();
    #pragma omp sections nowait
    {
        #pragma omp section
        {
            printf("Thread %d doing section 1\n",tid);
            for (i=0; i<N; i++) {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
            }
        }
        #pragma omp section
        {
            printf("Thread %d doing section 2\n",tid);
            for (i=0; i<N; i++) {
                d[i] = a[i] * b[i];
                printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
            }
        }
    } /* end of sections */
} /* end of parallel section */
```
For Loop

```c
#pragma omp for
for ( i = 0; ... ) {
    ... // for loop body
}
```

causes `for` loop to be divided into parts and parts shared among threads in the team – equivalent to a forall.

Must be a “For” loop of a simple C form such as `(i = 0; i < n; i++)`
Example

```c
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid) 
{
    tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d
", nthreads);
    }
    printf("Thread %d starting...
",tid);

    #pragma omp for
    for (i=0; i<N; i++) {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }
}
/* end of parallel section */
```
Single

The directive

```c
#pragma omp single
structured block
```

cause the structured block to be executed by one thread only.
Combined Parallel Work-sharing Constructs

If a parallel directive is followed by a single for directive, it can be combined into:

```
#pragma omp parallel for
<for loop> {
    ...
}
```

with similar effects.
If a parallel directive is followed by a single “sections” directive, it can be combined into

```c
#pragma omp parallel sections
{
    #pragma omp section
    structured_block

    #pragma omp section
    structured_block
    .
    .
    .
}
```

with similar effect. (In both cases, nowait clause is not allowed.)
Master Directive

The **master** directive:

```c
#pragma omp master
structured_block
```

causes the master thread to execute the structured block.

Different to those in the work sharing group in that there is no implied barrier at the end of the construct (nor the beginning).

Other threads encountering this directive will ignore it and the associated structured block, and will move on.
Loop Scheduling and Partitioning

OpenMP offers scheduling clauses to add to `for` construct:

1. Static

   ```
   #pragma omp parallel for schedule (static,chunk_size)
   ```

   Partitions loop iterations into equal sized chunks specified by `chunk_size`. Chunks assigned to threads in round robin fashion.

2. Dynamic

   ```
   #pragma omp parallel for schedule (dynamic,chunk_size)
   ```

   Uses internal work queue. Chunk-sized block of loop assigned to threads as they become available.
3. Guided

```c
#pragma omp parallel for schedule (guided,chunk_size)
```

Similar to dynamic but chunk size starts large and gets smaller to reduce time threads have to go to work queue.

\[
\text{chunk size} = \frac{\text{number of iterations remaining}}{2 \times \text{number of threads}}
\]

4. Runtime

```c
#pragma omp parallel for schedule (runtime)
```

Uses OMP_SCHEDULE environment variable to specify which of static, dynamic or guided should be used.
Reduction clause

Used combined the result of the iterations into a single value c.f. with MPI _Reduce().

Can be used with parallel, for, and sections,

Example

```c
sum = 0
#pragma omp parallel for reduction(+:sum)
    for (k = 0; k < 100; k++) {
        sum = sum + funct(k);
    }
```

Private copy of sum created for each thread by compiler.
Private copy will be added to sum at end.
Eliminates here the need for critical sections.
Private variables

**private** clause – creates private copies of variables for each thread

**firstprivate** clause - as private clause but initializes each copy to the values given immediately prior to parallel construct.

**lastprivate** clause – as private but “the value of each lastprivate variable from the sequentially last iteration of the associated loop, or the lexically last section directive, is assigned to the variable’s original object.”
Synchronization Constructs
Critical

critical directive will only allow one thread execute the associated structured block. When one or more threads reach critical directive:

```
#pragma omp critical name
structured_block
```

they will wait until no other thread is executing the same critical section (one with the same name), and then one thread will proceed to execute the structured block.

name is optional. All critical sections with no name map to one undefined name.
Barrier

When a thread reaches the barrier

```
#pragma omp barrier
```

it waits until all threads have reached the barrier and then they all proceed together.

Restrictions on the placement of barrier directive in a program. In particular, all threads must be able to reach the barrier.
Atomic

The atomic directive

```
#pragma omp atomic
expression_statement
```

implements a critical section efficiently when the critical section simply updates a variable (adds one, subtracts one, or does some other simple arithmetic operation as defined by `expression_statement`).
Flush

A synchronization point which causes thread to have a “consistent” view of certain or all shared variables in memory.

All current read and write operations on variables allowed to complete and values written back to memory but any memory operations in code after flush are not started.

Format:

#pragma omp flush (variable_list)

Only applied to thread executing flush, not to all threads in team.

Flush occurs automatically at entry and exit of parallel and critical directives, and at the exit of for, sections, and single (if a no-wait clause is not present).
Ordered clause

Used in conjunction with **for** and **parallel for** directives to cause an iteration to be executed in the order that it would have occurred if written as a sequential loop.
Debugging OpenMP Programs

Setting the Number of Threads:

- Setting the number of threads to use during a debug session is handled exactly as specified by the OpenMP standard. In order of precedence (lowest to highest):
  1. Default: usually equal to the number of cpus on the machine
  2. OMP_NUM_THREADS environment variable at run time
  3. OMP_SET_NUM_THREADS routine within the source code

Code Transformation:

- Probably the most obvious difference between OpenMP codes and other threaded codes is the compiler’s creation of outlined routines.

- Outlined routines are created when the compiler replicates the body of a PARALLEL region into a new, compiler created routine. This process is called outlining because it is the inverse of inlining a subroutine into its call site.

- In place of the parallel region, the compiler inserts a call to a run-time library routine. As the master thread creates worker threads, it dispatches them to the outlined routine, and then actually calls the outlined routine itself.

- Outlined Routine Names: These vary by compiler/platform. An example from the Intel Linux C compiler is shown below:

![Compiler generated outlined routine names](image)
Master Thread vs. Worker Threads:

- Thread Identifiers:
  - In TotalView, the OpenMP master thread always has a thread id of 1, and the worker threads greater than 1.
  - They do NOT match the actual OpenMP thread number. For example, in OpenMP, the master thread's id is zero.

- Depending upon the platform/compiler, the master thread may look different than the worker threads. The most important difference is how shared variables are displayed in the Stack Frame.

Case 1 - Different: Only the master thread displays a program's shared variables. Worker threads are limited to displaying their private variables. This is the case when using the IBM compilers on BG/Q systems at LC. The master/worker Stack Frames below demonstrate this:

**Case 2 - Same:** Both master and worker threads are enabled to display a program's shared variables. They also display their private variables identically. This is the case when using Intel compilers on Linux systems at LC. The master/worker Stack Frames below demonstrate this.
Example OpenMP Session:
1. Master thread Stack Trace Pane showing original routine (highlighted) and the outlined routine above it
2. Process/thread status bars differentiating threads
3. Master thread Stack Frame Pane showing shared variables
4. Worker thread Stack Trace Pane showing outlined routine.
5. Worker thread Stack Frame Pane, in this case showing both private and shared variables
6. Root Window showing all threads
7. Threads Pane showing all threads plus selected thread
Execution Control:

- Similar to threads as discussed previously.

- Stepping: you can not step into or out of a PARALLEL region. Instead, set a breakpoint within the parallel region and allow the process to run to it. From there you can single step within the parallel region.

Asynchronous execution: single stepping or running one OpenMP thread while others are stopped can lead to unexpected program behavior (like hanging). CTRL-C may be able to be used to cancel the command that caused the hang.

Viewing and Modifying Data:

- With the exception of SHARED and THREADPRIVATE variables/common blocks, (covered later) viewing and displaying data behaves the same as for other threaded codes.

- As with threaded codes, TotalView supports laminated variable displays for OpenMP also.

Manager Threads:

- Some platforms create additional threads for management purposes. Manager threads are given a negative thread id by TotalView.

- Manager threads should be ignored - do not try to debug them.

Example showing manager threads in addition to OpenMP threads. The Process Window Threads Pane is shown.
MPI THREAD SAFETY


Thread-safety in an MPI implementation: Requirements and analysis

William Gropp, Rajeev Thakur

Abstract
The MPI-2 Standard has carefully specified the interaction between MPI and user-created threads. The goal of this specification is to allow users to write multithreaded MPI programs while also allowing MPI implementations to deliver high performance. However, a simple reading of the thread-safety specification does not reveal what its implications are for an Implementation and what Implementers must be aware (and careful of). In this paper, we describe and analyze what the MPI Speciﬁes says about thread-safety and what it implies for an Implementation. We classify the MPI functions based on their thread-safety requirements and discuss several issues to consider when implementing thread-safety in MPI. We use the example of generating new context IDs (required for creating new communicators) to demonstrate how a simple solution for the single-threaded case does not naturally extend to the multithreaded case and how a naive thread-safe algorithm can be expensive. We then present an algorithm for generating context IDs that works efficiently in both single-threaded and multithreaded cases.

Keywords
Message-passing interface (MPI), Thread-safety, MPI Implementation, Multithreaded programming

1. Introduction
With SMP machines being commonly available and multicore chips becoming the norm, users are looking for ways to make better use of the multiple processors available on a single machine. One programming model being considered is a mixture of message-passing and multithreading, in which user programs consist of one or more MPI processes on each SMP node or multicore chip, with each MPI process itself comprising multiple threads [18]. MPI implementations must be able to support such programs efficiently.

The MPI-2 Standard has clearly deﬁned the interaction between MPI and user-created threads in an MPI program [8]. This specification was written with the goal of allowing users to write multithreaded MPI programs easily, without unduly burdening MPI implementations to support more than what a user might need. However, a simple reading of the Standard does not reveal all the implications the thread-safety specification has for an MPI implementation. Indeed, implementing thread safety in MPI correctly and without sacrificing too much performance requires careful thought and analysis.
int main (int argc, char *argv[]) {

    /* Parameters of MPI. */
    int n, /* number of MPI ranks */
    int my_id, /* MPI rank ID */
    int ret_val, /* return value */
    char name[128], /* MPI_MAX_PROCESSOR_NAME = 128 */
    int nname;

    /* 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, &n); /* get number of MPI ranks */
    rtn_val = MPI_Comm_rank(MPI_COMM_WORLD, &my_id); /* get MPI rank ID */
    MPI_Get_processor_name(name, &nname);
    printf("name: %s M_ID: %d M_N: %d\n", name, my_id, n);

    /* Get a few OpenMP parameters. */
    o_p = omp_get_num_procs(); /* get number of OpenMP processors */
    o_t = omp_get_max_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_T: %d\n", name, my_id, o_id, o_t);

    /* PARALLEL REGION */

    /* Thread IDs range from 0 through omp_get_max_threads()-1. */
    /* No 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_Comm_rank(MPI_COMM_WORLD, &nname);
        printf("parallel region: name: %s M_ID: %d O_ID: %d\n", name, my_id, o_id);

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

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

}
Example:
1. Start TotalView with the parallel task manager process. Note that the order of arguments and executables is important, and differs between platforms.

<table>
<thead>
<tr>
<th>MVAPICH Linux</th>
<th>totalview srun -n 16 -p debug myprog</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM AIX</td>
<td>totalview pse -m myprog -p 4 -genol D</td>
</tr>
<tr>
<td>SGI</td>
<td>totalview mpiexec -n myprog -np 16</td>
</tr>
<tr>
<td>Sun</td>
<td>totalview mpiexec -n myprog -np 16</td>
</tr>
<tr>
<td>MPICH</td>
<td>mpiexec -np 16 -tv myprog</td>
</tr>
</tbody>
</table>

2. The Root Window and Process Window will appear as usual, however, it will be the manager process that will be loaded, not your program. Start the manager process by typing `m` in the Process Window or by:

```
PATH: Process Window > Process Menu > Go
```

3. A dialog window will then appear notifying you that it is a parallel job and asking whether or not you wish to stop the job now. Click on “Yes” (see below). Note: if you click on “No” the job will begin to immediately execute before you have a chance to set breakpoints, etc.

4. TotalView will then acquire the MPI tasks which are running under the manager process. When this is done, the Process Window will default to displaying the state information and source for MPI task 0. You are now ready to begin debugging your program.
Selecting an MPI Process

- **By Dragging:**
  - After selecting a process in the *Root Window*, you can drag it by three different methods:
    - Double clicking
    - Right clicking and then selecting Drag from the pop-up menu
    - Selecting Drag from the *Root Window’s View Menu*
  - That process’s information will then be displayed in the current *Process Window*.
  - To force a new *Process Window* for a process, use *Drag in New Window* from the *View Menu* or right click pop-up menu. Multiple *Process Windows*, one for each MPI task, can be created this way.

- **By Process Navigation Buttons:**
  - Use the process navigation control buttons (below) located in the bottom right corner of the *Process Window*.
  - "Cycle through" the processes until the desired task’s information fills the *Process Window*.

- **Example:**
  - The example below demonstrates an MPI debug session. Some items of interest:
    2. Multiple *process windows* - one for MPI task 0 and one for MPI task 3
    3. *Root Window* showing manager task and multiple MPI processes
    4. Tasks are in different states
    5. Navigation buttons enabled for processes
Controlling MPI Process Execution

- MPI task execution can be controlled at the individual process level, or collectively as a "group".

- TotalView provides two different levels of control for MPI process execution commands. The table below describes these.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>Execution commands apply to all MPI processes</td>
</tr>
<tr>
<td>Process</td>
<td>Applies to a single MPI process</td>
</tr>
</tbody>
</table>

- Note that command scope is constrained to the selected TotalView P/T group (Control, Share, Workers, Lockstep) as discussed in the Process/Thread Groups section.

- Starting and Stopping Processes:
  - As seen previously, TotalView will ask you whether or not you wish to stop your parallel job before it starts to execute. Saying "Yes" to this allows you to set breakpoints and do other things before your tasks actually start running;
  - Starting your program and controlling its execution is then up to you, using either the Group Menu or the Process Menu from the Process Window.

  ![Question](image)

  - If you use accelerator keys to control execution, be sure to type the right key! It is a fairly common accident to use a process level command instead of group level command (and vice-versa). For example, typing `g` instead of `n`.

- Holding and Releasing Processes:
  - When a process is held, it is unresponsive to commands that would cause it to run, such as Go, Step, Next...
  - Processes are automatically placed in a hold state when they encounter a barrier point. They can also be placed on hold manually by either method below, depending upon whether you want to hold all processes or just one:

  ![Path](image)

  - Held processes will display an x state code in the Root Window (below).

  ![Root Window](image)

  - Processes are released automatically whenever all processes have reached the same barrier point. They can also be released manually:

  ![Path](image)

  - Note that releasing a process does not make it "Go". It only allows it to respond again to run type commands.
Breakpoints and Barrier Points:

- TotalView provides two options that control the behavior of breakpoints and barrier points:
  - **Sharing**: Should the action point be "planted" in all processes of the group? Planting means that if you set the action point in one MPI task, TotalView will automatically replicate it in all MPI tasks. The default behavior for both breakpoints and barrier points is to automatically plant the action point in all processes.
  - **Scoping**: Should the action point affect the group, the process or the thread(s)? The default behavior for both breakpoints and barrier points is to stop the process.

Individual breakpoint and barrier point behavior can be customized via the Action Point Properties Dialog Box. To open this window, first select a source line with a breakpoint or barrier point. Then do either:

  - Right-click (right-mouse) on the source code line and then select **Properties** from the resulting pop-up menu.

Action Point Properties Dialog Boxes for both breakpoints and barrier points are shown below.

- You can also customize the default behavior for all breakpoints and barrier points:
  1. Use either:
     - **PATH**: Root Window > File Menu > Preferences
     - **PATH**: Process Window > File Menu > Preferences
  2. Select the **Action Points Page** and then choose your desired options.

- See the previous discussion on **Action Points** for more information on using Breakpoints and Barrier Points.

**Warning About Single Process Commands:**

- If you use a process-level single stepping command in a multi-process MPI program, it is possible that TotalView will appear to hang. This happens when you step over a statement that cannot complete because the process it depends upon is stopped (as in communications).
- Using CTRL-C may be able to be used to cancel the step command that caused the hang.
Debugging Hybrid Programs

Starting a Hybrid Code Debug Session:

- If your hybrid code is a combination of MPI with either OpenMP or Pthreads, then you will most likely start your debug session as you would for MPI. See Starting an MPI Debug Session for examples.

- OpenMP programs will typically follow the usual convention for setting the number of threads as defined by the OpenMP standard: In order of precedence (lowest to highest):
  1. Default: usually equal to the number of cpus on the machine
  2. OMP_NUM_THREADS environment variable at run time
  3. OMP_SET_NUM_THREADS routine within the source code

Tying it All Together:

- MPI tasks behave individually as processes and collectively as a group
- Threads exist within an MPI process
- Execution control can be specified at the thread, process or group level within the selected P/T group
- Action points can be shared across a group or remain local to a process
- Every thread and process can have its own Process Window here.
- Selection and navigation between threads and processes works as usual
Example:

An example debug session with a hybrid MPI / Pthreads program is shown below. Some details of interest:

1. Each MPI task / thread can have its own Process Window - two are shown here.
2. Processes and threads are differentiated by pane trim and status bars.
3. Root Window showing multiple MPI processes, each with multiple threads. Manager process (run in this case) also appears.
5. MPI task identifiers and thread identifiers are the same as usual.
6. Both process and thread navigation buttons are active.
Remote Memory Access Programming in MPI-3

Torsten Hoefler, ETH Zurich
James Dinan, Argonne National Laboratory
Rajeev Thakur, Argonne National Laboratory
Brian Barrett, Sandia National Laboratories
Pavan Balaji, Argonne National Laboratory
William Gropp, University of Illinois at Urbana-Champaign
Keith Underwood, Intel Inc.

The Message Passing Interface (MPI) 3.0 standard, introduced in September 2012, includes a significant update to the one-sided communication interface, also known as remote memory access (RMA). In particular, the interface has been extended to better support popular one-sided and global-address-space parallel programming models, to provide better access to hardware performance features, and to enable new data-access modes. We present the new RMA interface and extract formal models for data consistency and access semantics. Such models are important for users, enabling them to reason about data consistency, and for tools and compilers, enabling them to automatically analyze, optimize, and debug RMA operations.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming; D.3.3 [Programming Languages]: Language Constructs and Features—Concurrent Programming Structures

General Terms: Design, Performance

Additional Key Words and Phrases: MPI, One-sided communication, RMA

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/000000.000000

1. MOTIVATION

Parallel programming models can be split into three categories: (1) shared memory with implicit communication and explicit synchronization, (2) message passing with explicit communication and implicit synchronization (as a side effect of communication), and (3) remote memory access and partitioned global address space (PGAS) where synchronization and communication are managed independently.

At the hardware side, high-performance networking technologies have converged toward remote direct memory access (RDMA) because it offers the highest performance (operating system bypass [Shivam et al. 2001]) and is relatively easy to implement. Thus, current high-performance networks, such as Cray's Gemini and Aries, IBM’s PERCS and BG/Q networks, InfiniBand, and Ethernet (using RoCE), all offer RDMA functionality.
Sequence diagram for client-server file transfer.
Ifconfig uses the ioctl access method to get the full address information, which limits hardware addresses to 8 bytes. Because Infiniband address has 20 bytes, only the first 8 bytes are displayed correctly. Ifconfig is obsolete! For replacement check ip.
ib0  Link encap:InfiniBand  HWaddr A0:00:01:00:FE:80:00:00:00:00:00:00:00:00:00:00:00 inet addr:10.100.68.104  Bcast:10.100.68.255  Mask:255.255.255.0
inet6 addr: fe80::202:c903:3d:f501/64 Scope:Link
UP BROADCAST RUNNING MULTICAST  MTU:2044  Metric:1
RX packets:373675843752 errors:0 dropped:0 overruns:0 frame:0
TX packets:308836448930 errors:0 dropped:5 overruns:0 carrier:0
collisions:0 txqueuelen:1024
RX bytes:398541677901981 (362.4 TiB) TX bytes:396246181843935 (360.3 TiB)

listening on port 38524.
received connection request.
received message: message from active/client side with pid 40882
connected. posting send...
send completed successfully.
peer disconnected.

address resolved.
route resolved.
connected. posting send...
send completed successfully.
received message: message from passive/server side with pid 17220
disconnected.
RDMA EXAMPLE - SERVER

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define TEST N(x) do { if (!x) die("error: \" x \" failed (returned non-zero).\"; } while (0)
define TEST Z(x) do { if (!x) die("error: \" x \" failed (returned zero/null).\"; } while (0)

const int BUFFER_SIZE = 1024;

struct context {
    struct ibv context *ctx;
    struct ibv_pd *pd;
    struct ibv_cq *cq;
    struct ibv_comp_channel *comp_channel;
    pthread_t cq_poller_thread;
};

struct connection {
    struct ibv_qp *qp;
    struct ibv_mr *recv_mr;
    struct ibv_mr *send_mr;
    char *recv_region;
    char *send_region;
};

static void die(const char *reason);

static void build_context(struct ibv_context *verbs);
static void build_qp_attr(struct ibv_qp_init_attr *qp_attr);
static void *poll_cq(void *);
static void post_recv(struct connection *conn);
static void register_memory(struct connection *conn);
static void on_completion(struct ibv_wc *wc);
static int on_connect_request(struct rdma_cm_id *id);
static int on_disconnect(struct rdma_cm_event *event);
static struct context *s_ctx = NULL;

int main(int argc, char **argv)
{
    struct sockaddr in6 addr;
    struct rdma_cm_event *event = NULL;
    struct rdma_cm_id *listener = NULL;
    struct rdma_event_channel *ec = NULL;
    uint16_t port = 0;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET6;
    TEST Z(0 = rdma create event channel());
    TEST NZ(rdma create id(ec, &listener, NULL, RDMA_PS_TCP));
    TEST NZ(rdma bind_addr(listener, (struct sockaddr *)&addr));
    TEST NZ(rdma_listen(listener, 10)); /* backlog=10 is arbitrary */
    port = ntohs(rdma get src port(listener));

    printf("listening on port %d.%d: port\n");
    while ((rdma get_cm event(ec, &event) == 0) {
        struct rdma_cm_event event_copy;
        nercopy(&event_copy, event, sizeof(*event));
        rdma_cm_event(events);
        if (on_event(&event_copy))
            break;
    }
    rdma destroy id(listener);
    rdma destroy event channel(ec);
    return 0;
}

void die(const char *reason)
{
    fprintf(stderr, "%s\n", reason);
    exit(EXIT_FAILURE);
}

void build_context(struct ibv_context *verbs)
{
    if (s_ctx->ctx->verbs)
        die("cannot handle events in more than one context.\";
    return;
}

s_ctx = (struct context *)malloc(sizeof(struct context));

s_ctx->ctx = verbs;

TEST Z(s_ctx->pd = ibv alloc pd(s_ctx->ctx));
TEST Z(s_ctx->comp_channel = ibv create comp channel(s_ctx->ctx));
TEST Z(s_ctx->cq = ibv create cq(s_ctx->ctx, 10, NULL, s_ctx->comp_channel, 0)); /* cqm=10 is arbitrary */
TEST NZ(rma_reg notify_cq(s_ctx->cq, 0));

TEST NZ(pthread create(s_ctx->cq_poller_thread, NULL, poll_cq, NULL));

void build_qp_attr(struct ibv_qp_init_attr *qp_attr)
{
    memset(qp_attr, 0, sizeof(*qp_attr));
    qp_attr->send_q = s_ctx->cq;
    qp_attr->recv_q = s_ctx->cq;
    qp_attr->qp_type = IBV_QP_TYPE;
    qp_attr->cap.max_send_wr = 10;
    qp_attr->cap.max_send_q = 1;
    qp_attr->cap.max_recv_q = 1;
}

void *poll_cq(void *ct)
{
    struct ibv_ctx *ctx;
    struct ibv_cq *cq;
    struct ibv_wc wc;
```
int on_connection(void *context)
{
    struct connection *conn = (struct connection *)context;
    struct ibv_send_wr wr, *bad_wr = NULL;
    struct ibv_sge sge;
    snprintf(conn->send_region, BUFFER_SIZE, "message from passive/server side with pid %d", getpid());
    printf("connected. posting send...\n");
    memset(&wr, 0, sizeof(wr));
    wr.opcode = IBV_WR_SEND;
    wr.sg_list = &sge;
    wr.num_sge = 1;
    wr.send_flags = IBV_SEND_SIGNALED;
    sge.addr = (uintptr_t)conn->send_region;
    sge.length = BUFFER_SIZE;
    sge.lkey = conn->send_mr->lkey;
    TEST_NZ(ibv_post_send(conn->qp, &wr, &bad_wr));
    return 0;
}

int on_disconnect(struct rdma_cm_id *id)
{
    struct connection *conn = (struct connection *)id->context;
    printf("peer disconnected.\n");
    rdma_destroy_qp(id);
    ibv_dereg_mr(conn->send_mr);
    ibv_dereg_mr(conn->recv_mr);
    free(conn->send_region);
    free(conn->recv_region);
    free(conn);
    rdma_destroy_id(id);
    return 0;
}

int on_event(struct rdma_cm_event *event)
{
    int r = 0;
    if (event->event == RDMA_CM_EVENT_CONNECT_REQUEST)
    r = on_connect_request(event->id);
    else if (event->event == RDMA_CM_EVENT_ESTABLISHED)
    r = on_connection(event->id->context);
    else if (event->event == RDMA_CM_EVENT_DISCONNECTED)
    r = on_disconnect(event->id);
    else
    die("on_event: unknown event.");
    return r;
}
RDMA EXAMPLE - CLIENT

```c
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <rdma/rdma_cm.h>

#define TEST_NZ(x) do { if ( (x) ) die("error: " #x " failed (returned non-zero)." ); } while (0)
#define TEST_Z(x) do { if (! (x) ) die("error: " #x " failed (returned zero/null)." ); } while (0)

const int BUFFER_SIZE = 1624;
const int TIMEOUT_IN_MS = 500; /* ms */

struct context {
    struct ibv_context *ctx;
    struct ibv_pd *pd;
    struct ibv_qp *cq;
    struct ibv_comp_channel *comp_channel;
    pthread_t cm_poller_thread;
};

struct connection {
    struct rdma_cm_id *id;
    struct ibv_qp *qp;
    struct ibv_mr *recv_mr;
    struct ibv_mr *send_mr;
    char *recv_region;
    char *send_region;
    int num_completions;
};

static void die(const char *reason);
static void build_context(struct ibv_context *verbs);
static void build_qp_attr(struct ibv_qp_init_attr *qp_attr);
static void *poll_cq(void *);
static void post_receives(struct connection *conn);
static void register_memory(struct connection *conn);
static int on_addr_resolved(struct rdma_cm_id *id);
static void on_completion(struct ibv_wc *wc);
static int on_connection(void *context);
static int on_disconnect(struct rdma_cm_id *id);
static int on_event(struct rdma_cm_event *event);
static int on_route_resolved(struct rdma_cm_id *id);

static struct context *s_ctx = NULL;

int main(int argc, char **argv)
{
    struct addrinfo *addr;
    struct rdma_cm_event *event = NULL;
    struct rdma_cm_id *conn = NULL;
    struct rdma_event_channel *ec = NULL;

    if (argc != 3)
        die("usage: client <server-address> <server-port>\n");

    TEST_NZ(getaddrinfo(argv[1], argv[2], NULL, &addr));
```
void *poll_cq(void *ctx)
{
    struct ibv_cq *cq;
    struct ibv_wc wc;

    while (1) {
        TEST_NZ(ibv_get_cq_event(s_ctx->comp_channel, &cq, &ct);
        ibv_nck_cq_events(cq, 1);
        TEST_NZ(ibv_req_notify_cq(cq, 0));
        while (ibv_poll_cq(cq, 1, &wc))
            on_completion(&wc);
    }
    return NULL;
}

void post_receives(struct connection *conn)
{
    struct ibv_recv_wr wr, *bad_wr = NULL;
    struct ibv_sge sge;

    wr.wr_id = (uintptr_t)conn;
    wr.next = NULL;
    wr.sg_list = &sge;
    wr.num_sge = 1;
    sge.addr = (uintptr_t)conn->recv_region;
    sge.length = BUFFER_SIZE;
    sge.lkey = conn->recv_mr->lkey;
    TEST_NZ(ibv_post_recv(conn->qp, &wr, &bad_wr));
}

void register_memory(struct connection *conn)
{
    conn->send_region = malloc(BUFFER_SIZE);
    conn->recv_region = malloc(BUFFER_SIZE);

    TEST_NZ(conn->send_mr = ibv_reg_mr(
        s_ctx->pd,
        conn->send_region,
        BUFFER_SIZE,
        IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_WRITE));
    TEST_NZ(conn->recv_mr = ibv_reg_mr(
        s_ctx->pd,
        conn->recv_region,
        BUFFER_SIZE,
        IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_WRITE));
}
int on_addr_resolved(struct rdma_cm_id *id)
{
    struct ibv_qp_init_attr qp_attr;
    struct connection *conn = malloc(sizeof(struct connection));
    conn->id = id;
    conn->qp = id->qp;
    conn->num_completions = 0;
    register_memory(conn);
    post_receives(conn);
    TEST_NZ(rdma_resolve_route(id, TIMEOUT_MS));
    return 0;
}

void on_completion(struct IBV_WR *wr)
{
    struct connection *conn = (struct connection *)(uintptr_t)wr->wr_id;
    if (wr->status != IBV_WR_SUCCESS)
        die("on_completion: status is not IBV_WR_SUCCESS.");
    if (wr->opcode & IBV_WR_SEND)
        die("on_completion: completion isn't a send or a receive.");
    TEST_NZ(conn->num_completions++);
    rdma_disconnect(conn->id);
}

int on_connection(void *context)
{
    struct connection *conn = (struct connection *)context;
    struct ibv_send_wr wr, *bad_wr = NULL;
    struct ibv_sge sge;
    snprintf(conn->send_region, BUFFER_SIZE, "message from active/client side with pid %d", getpid());
    printf("connected. posting send...
");
    memset(&wr, 0, sizeof(wr));
    wr.wr_id = (uintptr_t)conn;
    wr.opcode = IBV_WR_SEND;
    wr.sg_list = &sge;
    wr.num_sge = 1;
    wr.send_flags = IBV_SEND_SIGNALED;
    sge.addr = (uintptr_t)conn->send_region;
    sge.length = BUFFER_SIZE;
    sge.lkey = conn->send_mr->lkey;
    TEST_NZ(ibv_post_send(conn->qp, &wr, &bad_wr));
    return 0;
}

int on_disconnect(struct rdma_cm_id *id)
{
    struct connection *conn = (struct connection *)id->context;
    printf("disconnected.
");
    rdma_destroy_qp(id);
    ibv_dereg_mr(conn->send_mr);
    ibv_dereg_mr(conn->recv_mr);
    free(conn->send_region);
    free(conn->recv_region);
    free(conn);
    rdma_destroy_id(id);
    return 1; /* exit event loop */
}

int on_event(struct rdma_cm_event *event)
{
    int r = 0;
    if (event->event == RDMA_CM_EVENT_ADDR_RESOLVED)
        r = on_addr_resolved(event->id);
    else if (event->event == RDMA_CM_EVENT_ROUTE_RESOLVED)
        r = on_route_resolved(event->id);
    else if (event->event == RDMA_CM_EVENT_ESTABLISHED)
        r = on_connection(event->id->context);
    else if (event->event == RDMA_CM_EVENT_DISCONNECTED)
        r = on_disconnect(event->id);
    else
        die("on_event: unknown event.");
    return r;
}

int on_route_resolved(struct rdma_cm_id *id)
{
    struct rdma_conn_param cm_params;
    printf("route resolved.
");
    memset(&cm_params, 0, sizeof(cm_params));
    TEST_NZ(rdma_connect(id, &cm_params));
    return 0;
}
IB TRANSPORT LAYER DEBUGGING

Application ("Consumer of InfiniBand message services"):  
- posts 'work requests' to a queue  
- each work request represents a message...a unit of work

Channel interface (verbs):  
- allows the consumer to request services

InfiniBand Channel interface provider:  
- Maintains work queues  
- Manages address translation  
- Provides completion and event mechanisms

IB Transport:  
- Packetizes messages  
- Provides transport service  
  - reliable/unreliable, connected/unconnected, datagram  
- Implements RDMA protocol  
  - send/receive, RDMA r/w, Atomics  
- Implements end-to-end reliability  
- Assures reliable delivery

DEBUGGING OVER IB VERBS
NOW LETS LOOK AT A RDMA EXAMPLE RUN ON DISCOVERY CLUSTER

• PARALLEL-IB QUEUE RDMA/IPOIB – FDR 56Gbps
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
  ```bash
  mpiexec -np 2 cuda-memcheck ./myapp <args>
  ```
- Problem: output of different processes is interleaved
  - Use save, log-file command line options and launcher script
    ```bash
    #!/bin/bash
    LOG=$1.$OMPI_COMM_WORLD_RANK
    LOG=$1.$MV2_COMM_WORLD_RANK
    cuda-memcheck --log-file $LOG.log --save $LOG.memcheck $*
    ```
    ```bash
    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
  ```bash
  > 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
  ```c
  if ( rank == 0 ) {
      int i=0;
      printf("rank %d: pid %d on %s ready for attach\n", rank, getpid(),name);
      while ( i < 1 ) {
          sleep(5);
      }
  }
  ```
  ```bash
  > mpiexec -np 2 ./jacobi_mpi+cuda
  rank 0: pid 30034 on ge107 ready for attach
  > ssh ge107
  cuda-gdb --pid 30034
  ```

DDT: THREAD LEVEL DEBUGGING
- Allinea DDT debugger
- Totalview
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
Load Balancing

• Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time.

• Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.

![Load Balancing Diagram]

How to Achieve Load Balance:

• Equally partition the work each task receives
  o For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
  o For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  o If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances. Adjust work accordingly.

• Use dynamic work assignment
  o Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
    o Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".
    o Adaptive grid methods - some tasks may need to refine their mesh while others don't.
    o N-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.
  o When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a scheduler - task pool approach. As each task finishes its work, it queues to get a new piece of work.
  o It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.
Synchronization

- Managing the sequence of work and the tasks performing it is a critical design consideration for most parallel programs.

- Can be a significant factor in program performance (or lack of it)

- Often requires "serialization" of segments of the program.

Types of Synchronization:

- **Barrier**
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized
  - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.

- **Lock / semaphore**
  - Can involve any number of tasks
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking

- **Synchronous communication operations**
  - Involves only those tasks executing a communication operation
  - When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.
Debugging/Profiling MPI with MPE
TOOLS FOR LOAD BALANCING / SYNCHRONIZATION

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
THANK YOU - QUESTIONS?

- TAU: http://www.cs.uoregon.edu/research/tau/docs.php
- HPCToolkit: http://hpctoolkit.org/documentation.html
- OpenSpeedshop: http://www.openspeedshop.org/wp/
- Vampir/Vampirtrace: http://vampir.eu/
- Valgrind: http://valgrind.org/
- PAPI: http://icl.cs.utk.edu/papi/
- mpitrace: https://computing.llnl.gov/tutorials/bgq/index.html#mpitrace
- mpiP: http://mpip.sourceforge.net/
- memP: http://memp.sourceforge.net/