Implementation of Parallelization
OpenMP, PThreads and MPI

Jascha Schewtschenko
Institute of Cosmology and Gravitation, University of Portsmouth

May 9, 2018
Outline

1. OpenMP
2. POSIX Threads
3. MPI
4. Debugging
OpenMP
OpenMP - Goals

Standardization provide standard for variety of platforms/shared-memory architectures

Lean and Mean simple and limited set of directives, very few uses of directives needed

Ease of Use can incrementally parallelize program (source stays the same except for added directives), supports both coarse-grain and fine-grain parallelism

Portability public API, implementations for C, C++, Fortran

JAS (ICG, Portsmouth)
OpenMP - Goals

**Standardization** provide standard for variety of platforms/shared-mem architectures
OpenMP - Goals

**Standardization**  provide standard for variety of platforms/shared-mem architectures

**Lean and Mean**  simple and limited set of directives, very few uses of directives needed
OpenMP - Goals

**Standardization** provide standard for variety of platforms/shared-mem architectures

**Lean and Mean** simple and limited set of directives, very few uses of directives needed

**Ease of Use** can incrementally parallelize program (source stays the same except for added directives), supports both coarse-grain and fine-grain parallelism
OpenMP - Goals

**Standardization** provide standard for variety of platforms/shared-mem architectures

**Lean and Mean** simple and limited set of directives, very few uses of directives needed

**Ease of Use** can incrementally parallelize program (source stays the same except for added directives), supports both coarse-grain and fine-grain parallelism

**Portability** public API, implementations for C, C++, Fortran
OpenMP - Structure & Implementations

Supported/shipped with various compilers for various platforms (e.g. Intel and GNU compilers for Linux), i.e. to compile simply add option:

e.g. gcc -fopenmp

uses a form-join model comprised of 3 API components:

- Compiler Directives
- Runtime Library routines
- Environment Variables
OpenMP - Structure & Implementations

- Supported/shipped with various compilers for various platforms (e.g. Intel and GNU compilers for Linux), i.e. to compile simply add option:
  
  e.g. gcc -fopenmp
OpenMP - Structure & Implementations

- Supported/shipped with various compilers for various platforms (e.g. Intel and GNU compilers for Linux), i.e. to compile simply add option:
  
  e.g. gcc -fopenmp

- uses a form-join model

![Diagram of OpenMP parallel regions and threads]

- Master thread
- Parallel regions
- Threads

Parallel region with nested parallel region
OpenMP - Structure & Implementations

- Supported/shipped with various compilers for various platforms (e.g. Intel and GNU compilers for Linux), i.e. to compile simply add option:
  
  e.g. gcc -fopenmp

- uses a form-join model

- comprised of 3 API components:
  - Compiler Directives
  - Runtime Library routines
  - Environment Variables
OpenMP - Compiler Directives

We will focus here on C/C++ syntax, FORTRAN syntax slightly different:

```
#pragma omp <directive name> [clauses]
```
(C/C++)

```
!$OMP [END] <directive name> [clauses]
```
(Fortran)

Used for:
- Defining parallel regions / spawning threads
- Distributing loop iterations or sections of code between threads
- Serializing sections of code (e.g. for access to I/O or shared variables)
- Synchronizing threads

You can find a reference sheet for the C/C++ API for OpenMP 4.0 in the source code archive for this workshop.
OpenMP - Compiler Directives

We will focus here on C/C++ syntax, FORTRAN syntax slightly different:

```c
#pragma omp <directive name> [clauses] (C/C++)
```

```fortran
$OMP [END] <directive name> [clauses] (Fortran)
```

Used for:
- Defining parallel regions / spawning threads
- Distributing loop iterations or sections of code between threads
- Serializing sections of code (e.g. for access to I/O or shared variables)
- Synchronizing threads

You can find a reference sheet for the C/C++ API for OpenMP 4.0 in the source code archive for this workshop.
OpenMP - Compiler Directives

We will focus here on C/C++ syntax, FORTRAN syntax slightly different:

```
#pragma omp <directive name> [<clauses>] (C/C++)
!$OMP [END] <directive name> [<clauses>] (Fortran)
```

Used for:
- Defining parallel regions / spawning threads
- Distributing loop iterations or sections of code between threads
- Serializing sections of code (e.g. for access to I/O or shared variables)
- Synchronizing threads

You can find a reference sheet for the C/C++ API for OpenMP 4.0 in the source code archive for this workshop.
OpenMP - Runtime Library Routines

These routines are provided by the openmp library are used to configuring and monitoring the multithreading during execution: e.g. `omp get num threads` returns number of threads in current team, `omp in parallel` check if in parallel regions, `omp set schedule` modify scheduler policy. There are further routines for locks for synchronization/access control (see later) as well as timing routines for recording elapsed time for each thread.
These routines are provided by the openmp library are used to configuring and monitoring the multithreading during execution: e.g.

- `omp_get_num_threads` returns number of threads in current team
- `omp_in_parallel` check if in parallel regions
- `omp_set_schedule` modify scheduler policy
These routines are provided by the openmp library are used to configuring and monitoring the multithreading during execution: e.g. 

- `omp_get_num_threads` returns number of threads in current team
- `omp_in_parallel` check if in parallel regions
- `omp_set_schedule` modify scheduler policy

There are further routines for locks for synchronization/access control (see later)
These routines are provided by the openmp library are used to configuring and monitoring the multithreading during execution: e.g.

- `omp_get_num_threads` returns number of threads in current team
- `omp_in_parallel` check if in parallel regions
- `omp_set_schedule` modify scheduler policy

There are further routines for locks for synchronization/access control (see later)

as well as timing routines for recording elapsed time for each thread.
Like for most programs in the UNIX world, environmental variables are used to store configurations needed for running the program. In OpenMP, they are used for setting, e.g., the number of threads per team (\texttt{OMP_NUM_THREADS}), maximum number of threads (\texttt{OMP_THREAD_LIMIT}) or the scheduler policy (\texttt{OMP_SCHEDULE}).

While most of these settings can also be done using clauses in the compiler directives of runtime library routines, environmental variables provide a user an easy way to change these crucial settings without the need of an additional config file (parsed by your program) or even rewriting/recompiling the openmp-enhanced program.
Like for most programs in the UNIX world, environmental variables are used to store configurations needed for running the program. In OpenMP, they are used for setting e.g. the number of threads per team (OMP_NUM_THREADS), maximum number of threads (OMP_THREAD_LIMIT) or the scheduler policy (OMP_SCHEDULE).
Like for most programs in the UNIX world, environmental variables are used to store configurations needed for running the program. In OpenMP, they are used for setting e.g. the number of threads per team (OMP_NUM_THREADS), maximum number of threads (OMP_THREAD_LIMIT) or the scheduler policy (OMP_SCHEDULE).

While most of these settings can also be done using clauses in the compiler directives of runtime library routines, environmental variables provide a user an easy way to change these crucial settings without the need of an additional config file (parsed by your program) or even rewritting/recompiling the openmp-enhanced program.
OpenMP - Worksharing

Diagram showing the flow of control in OpenMP worksharing:

1. **FORK**
   - **DO / for loop**
   - **JOIN**
2. **FORK**
   - **SECTIONS**
   - **JOIN**
3. **FORK**
   - **SINGLE**
   - **JOIN**

Each diagram represents a different approach to parallelization, with the master thread initiating tasks and the threads joining back to the master thread.
#include <omp.h>
define N 1000
define CHUNKSIZE 100

main(int argc, char *argv[]) {
    int i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++)
                c[i] = a[i] + b[i];

            #pragma omp section
            for (i=0; i < N; i++)
                d[i] = a[i] * b[i];
        } /* end of sections */
    } /* end of parallel region */
}

#include <omp.h>
define N 1000
define CHUNKSIZE 100

main(int argc, char *argv[]) {
    int i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel for 
    shared(a,b,c,chunk) private(i) 
    schedule(static,chunk)
    for (i=0; i < n; i++)
        c[i] = a[i] + b[i];
}
OpenMP - advanced Worksharing

defines explicit tasks similar to sections that are generated (usually by a single task) and then deferred to any thread in the team via a queue/scheduler.

tasks are not necessarily tied to a single thread, can be e.g. postponed or migrated to other threads.

allows for defining dependencies among tasks (e.g. task X has to finish before any thread can work on task Y).
defines explicit tasks similar to sections that are generated (usually by a single task) and then deferred to any thread in the team via a queue/scheduler.
OpenMP - advanced Worksharing

- defines explicit tasks similar to sections that are generated (usually by a single task) and then deferred to any thread in the team via a queue/scheduler.
- tasks are not necessarily tied to a single thread, can be e.g. postponed or migrated to other threads.
OpenMP - advanced Worksharing

- defines explicit tasks similar to sections that are generated (usually by a single task) and then deferred to any thread in the team via a queue/scheduler
- tasks are not necessarily tied to a single thread, can be e.g. postponed or migrated to other threads
- allows for defining dependencies among tasks (e.g. task X has to finish before any thread can work on task Y)
OpenMP - advanced Worksharing (example)

```c
#include <omp.h>

float sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }

    // recursive case
    size_t half = n / 2;
    float x, y;

    #pragma omp parallel shared(x,y)
    #pragma omp single nowait
    {
        #pragma omp task shared(x)
        x = sum(a, half);
        #pragma omp task shared(y)
        y = sum(a + half, n - half);
        #pragma omp taskwait
        x += y;
    }

    return x;
}
```
In the ’Introduction to Parallelization’, we discussed the need of controlling the execution of threads at certain points to e.g. synchronize them to exchange intermediate results or to protect resources from getting accessed simultaneously with non-deterministic outcome (’race condition’). OpenMP provides two ways to do this:
In the 'Introduction to Parallelization', we discussed the need of controlling the execution of threads at certain points to e.g. synchronize them to exchange intermediate results or to protect resources from getting accessed simultaneously with non-deterministic outcome ('race condition'). OpenMP provides two ways to do this:

- **Compiler Directives:**
  - (for general parallel regions) e.g. cancel, single, master, critical, atomic, barrier
  - (for loops) ordered
  - (for tasks) taskwait, taskyield
In the ’Introduction to Parallelization’, we discussed the need of controlling the execution of threads at certain points to e.g. synchronize them to exchange intermediate results or to protect resources from getting accessed simultaneously with non-deterministic outcome (’race condition’). OpenMP provides two ways to do this:

- **Compiler Directives:**
  - (for general parallel regions) e.g. cancel, single, master, critical, atomic, barrier
  - (for loops) ordered
  - (for tasks) taskwait, taskyield

- **Runtime Library Routines:**
  - `omp_set_lock`, `omp_unset_lock`, `omp_test_lock`
OpenMP - Synchronization / Flow control

(RESTRICTION)

BARRIER

SINGLE

MASTER

FORK

BARRIER

JOIN

TEAM

master thread

implicit BARRIER

NO implicit BARRIER

JAS (ICG, Portsmouth) Implementation of Parallelization May 9, 2018 14 / 48
CRITICAL, ATOMIC exclusive for ALL threads, not just team

CRITICAL regions can be named, regions with same name treated as same region
Certain clauses for compiler directives allow us to specify how data is shared (e.g. shared, private, threadprivate) and how they are initialized (e.g. firstprivate, copyin)
Certain clauses for compiler directives allow us to specify how data is shared (e.g. shared, private, threadprivate) and how they are initialized (e.g. firstprivate, copyin).

Others like copyprivate allow for broadcasting the content of private variables from one thread to all others.
Certain clauses for compiler directives allow us to specify how data is shared (e.g. shared, private, threadprivate) and how they are initialized (e.g. firstprivate, copyin).

Others like copyprivate allow for broadcasting the content of private variables from one thread to all others.

Similarly, the reduction clause provides an elegant way to gather private data from the threads when joining them.
Similarly, the *reduction* clause provides an elegant way to gather private data from the threads when joining them.

```c
#include <omp.h>

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

    int i, n, chunk;
    int a[100], b[100], result;

    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    #pragma omp parallel for  \ 
    default(shared) private(i) \ 
    schedule(static,chunk)  \ 
    reduction(+:result)

    for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);

    printf("result= %d\n",result);
}
```
OpenMP - Memory management (FLUSHING DATA)

even if shared, sometimes variable may not be updated in the "global" view, e.g. if kept in a register or cache of a CPU instead of the shared memory.

while many directives (e.g. `for`, `section`, `critical`) implicitly flush variable to synchronize them with other threads, sometimes explicit flushing using the `flush` may be necessary.
even if shared, sometimes variable may not be updated in the "global" view, e.g. if kept in a register or cache of a CPU instead of the shared memory
even if shared, sometimes variable may not be updated in the “global” view, e.g. if kept in a register or cache of a CPU instead of the shared memory

while many directives (e.g. for, section, critical) implicitly flush variable to synchronize them with other threads, sometimes explicit flushing using the flush may be necessary.
OpenMP - Memory management (STACK)

The OpenMP standard does not specify a default stack size for each thread. So it depends on the compiler, e.g.,

- Compiler: icc/ifort (Linux)
  - Approx Stack Limit: 4 MB
- Compiler: gcc/gfort (Linux)
  - Approx Stack Limit: 2 MB

If stack allocation exceeds the limit, it may result in a seg fault or worse, data corruption.

Environment variable `OMP_STACKSIZE` allows to set the stacksize prior to execution. If your program needs a significant amount of data on the stack, make sure to adapt the stacksize this way!

---

**Diagram:**

- **User Address Space**
  - `text`
    - `main()`
    - `routine1()`
    - `routine2()`
    - ...
  - `data`
    - `arrayA`
    - `arrayB`
- `Thread 1`
  - `stack`
    - `routine1()` var1 var2
- `Thread 2`
  - `stack`
    - `routine2()` var1 var2 var3

---

JAS (ICG, Portsmouth)  Implementation of Parallelization  May 9, 2018
OpenMP standard does not specify a default stack size for each thread. So it depends on the compiler e.g.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Approx Stack Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc/ifort (Linux)</td>
<td>4 MB</td>
</tr>
<tr>
<td>gcc/gfort (Linux)</td>
<td>2 MB</td>
</tr>
</tbody>
</table>

If stack allocation exceeds these limits, it may result in a segmentation fault or worse, data corruption. The environment variable `OMP_STACKSIZE` allows setting the stack size prior to execution. If your program needs a significant amount of data on the stack, make sure to adapt the stack size this way!
OpenMP - Memory management (STACK)

- OpenMP standard does not specify a default stack size for each thread. So depends on the compiler e.g.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Approx Stack Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc/ifort (Linux)</td>
<td>4 MB</td>
</tr>
<tr>
<td>gcc/gfort (Linux)</td>
<td>2 MB</td>
</tr>
</tbody>
</table>

- if stack allocation exceeded, may result in seg fault or (worse) data corruption.
OpenMP standard does not specify a default stack size for each thread. So it depends on the compiler e.g.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Approx Stack Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc/ifort (Linux)</td>
<td>4 MB</td>
</tr>
<tr>
<td>gcc/gfort (Linux)</td>
<td>2 MB</td>
</tr>
</tbody>
</table>

If stack allocation exceeded, may result in seg fault or (worse) data corruption.

Env. variable `OMP_STACKSIZE` allows to set stacksize prior to execution. So if your program needs a significant amount of data on the stack, make sure to adapt the stacksize this way!
POSIX Threads
PThreads - History/Goals
PThreads - History/Goals

- standardized API for multithreading to allow for portable threaded applications
PTThreads - History/Goals

- standardized API for multithreading to allow for portable threaded applications
- first defined in IEEE POSIX standard 1003.1c in 1995, but undergoes continuous evolution/revision
standardized API for multithreading to allow for portable threaded applications

first defined in IEEE POSIX standard 1003.1c in 1995, but undergoes continuous evolution/revision

historically implementations focused on Unix as OS, but implementations also exist now for others e.g. for Windows
Like for OpenMP, POSIX Threads are included in most recent compiler suites by default. To enable these included libraries, use e.g. `icc -pthread` for INTEL (Linux) or `gcc -pthread` for GNU (Linux).
Like for OpenMP, POSIX Threads are included in most recent compiler suites by default.
Like for OpenMP, POSIX Threads are included in most recent compiler suites by default.

To enable these included libraries, use e.g.

- `icc -pthread` for INTEL (Linux)
- `gcc -pthread` for GNU (Linux)
The subroutines defined in the API can be classified into four major groups:

1. **Thread management**: For creating new threads, checking their properties and joining/destroying them and the end of their lifecycle (`pthread`, `pthread_attr`).

2. **Mutexes**: For creating mutex locks to control access to exclusive resources (`pthread_mutex`, `pthread_mutexattr`).

3. **Condition variables**: Routines for managing condition variables to allow for easy communication between threads that share a mutex (`pthread_cond`, `pthread_condattr`).

4. **Synchronization barriers, read/write locks**: (`pthread_barrier`, `pthread_rwlock`).
The subroutines defined in the API can be classified into four major groups:

**Thread management** For creating new threads, checking their properties and joining/destroying them and the end of their lifecycle (`pthread_`, `pthread_attr_`)

**Mutexes** For creating mutex locks to control excess to exclusive resources (`pthread_mutex_`, `pthread_mutexattr_`)

**Condition variables** routines for managing condition variable to allow for easy communication between threads that share a mutex (`pthread_cond_`, `pthread_condattr_`)

**Synchronization** barriers, read/write locks (`pthread_barrier_`, `pthread_rwlock_`)
The subroutines defined in the API can be classified into four major groups:

**Thread management** For creating new threads, checking their properties and joining/destroying them and the end of their lifecycle (pthread_, pthread_attr_)

**Mutexes** For creating mutex locks to control excess to exclusive resources (pthread_mutex_, pthread_mutexattr_)

**Condition variables** routines for managing condition variable to allow for easy communication between threads that share a mutex (pthread_cond_, pthread_condattr_)

**Synchronization** barriers, read/write locks (pthread_barrier_, pthread_rwlock_)
PThreads - Thread management: Creation & Termination

POSIX threads (pthread) are created explicitly using the
pthread_create(thread, attr, start routine, arg)
where
▶ attr is a thread attribute structure containing settings for creating/running thread
▶ start routine is a procedure that works as a starting point for the thread
▶ arg is a pointer to the argument for the starting routine (can be pointing to a single data element, an array or a custom data structure)

They terminate when finishing their starting routine, calling pthread_exit(status) to return a status flag, by another thread by calling pthread_cancel(thread) with thread pointing to them or the host process finishing first (without pthread_exit() call)
POSIX threads (pthread_t) are created explicitly using the `pthread_create(thread, attr, start_routine, arg)` where
- `attr` is a thread attribute structure containing settings for creating/running thread
- `start_routine` is a procedure that works as a starting point for the thread
- `arg` is a pointer to the argument for the starting routine (can be pointing to a single data element, an array or a custom data structure)
POSIX threads (pthread_t) are created explicitly using the `pthread_create(thread, attr, start_routine, arg)` where

- attr is a thread attribute structure containing settings for creating/running thread
- start_routine is a procedure that works as a starting point for the thread
- arg is a pointer to the argument for the starting routine (can be pointing to a single data element, an array or a custom data structure)

They terminate when finishing their starting routine, calling `pthread_exit(status)` to return a status flag, by another thread by calling `pthread_cancel(thread)` with thread pointing to them or the host process finishing first (without `pthread_exit()` call)
```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #\%ld\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread \%ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is \%d\n", rc);
            exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
PThreads - Thread management: Joining & Detaching

“Joining” threads allows the master thread to synchronize with its worker threads on completion of their task. Threads can be declared “joinable” on creation. The data (Thread Control Block) remains in memory after completion of a thread until `pthread_join` is called on this dead thread and the clean-up is triggered. “Detached” threads do not keep such (potentially unnecessary) data, i.e. get cleaned up directly on completion.
“Joining” threads allows the master thread to synchronize with its worker threads on completion of their task.
PThreads - Thread management: Joining & Detaching

- “Joining” threads allows the master thread to synchronize with its worker threads on completion of their task.

```
Master Thread
  → pthread_create() → pthread_join()
  ↓
Worker Thread
  --- DO WORK ---
  ↓
Worker Thread
  → pthread_exit()
```

Thread data (Thread Control Block) remains in memory after completion of a thread until `pthread_join()` is called on this dead thread and the clean-up is triggered.

“Detached” threads do not keep such (potentially unnecessary) data, i.e. get cleaned up directly on completion.
“Joining” threads allows the master thread to synchronize with its worker threads on completion of their task.

threads can be declared “joinable” on creation.
“Joining” threads allows the master thread to synchronize with its worker threads on completion of their task.

- threads can be declared “joinable” on creation

- data (Thread Control Block) remains in memory after completion of a thread until `pthread_join` is called on this dead thread and the clean-up is triggered.
PThreads - Thread management: Joining & Detaching

- “Joining” threads allows the master thread to synchronize with its worker threads on completion of their task.

- threads can be declared “joinable” on creation.

- data (Thread Control Block) remains in memory after completion of a thread until `pthread_join` is called on this dead thread and the clean-up is triggered.

- “detached” threads do not keep such (potentially unnecessary) data, i.e. get cleaned up directly on completion.
Mutexes work in similar way as the OpenMP locks: once claimed by one thread, other threads encountering it will be hold until the mutex released again.
//define NUMTHRDS 4
//define VECLEN 1000000

void *dotprod(void *arg)
{
    mysum = 0;
    for (i=start; i<end; i++)
    {
        mysum += (x[i] * y[i]);
    }

    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    printf("Thread %ld did %ld to %ld: mysum=%f global sum=%f\n", offset, start, end, mysum, dotstr.sum);
    pthread_mutex_unlock (&mutexsum);

    pthread_exit((void*) 0);
}

int main (int argc, char *argv[])
{
    long i;
    double *a, *b;
    void *status;
    pthread_attr_t attr;

    a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
    b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

    [...]
PThreads - Condition variables

Conditions variables control the flow of threads like Mutexes. Instead of claiming a lock, it allows threads to wait (`pthread_cond_wait()`) until another thread sends a signal (`pthread_cond_signal()`) through the condition variable to continue.
PThreads - Condition variables

- Conditions variables control the flow of threads like Mutexes
Conditions variables control the flow of threads like Mutexes. Instead of claiming a lock, it allows threads to wait \( \texttt{pthread\_cond\_wait()} \) until another thread send a signal \( \texttt{pthread\_cond\_signal()} \) through the condition variable to continue.
POSIX Threads also feature a synchronization barrier similar to OpenMP. Since there are no "team" structure like in OpenMP, on creation a number of threads is defined, that has to reach the barrier before any of them is allowed to pass.
POSIX Threads also feature a synchronization barrier similar to OpenMP.
POSIX Threads also feature a synchronization barrier similar to OpenMP.

Since there are no "team" structure like in OpenMP, on creation a number of threads is defined, that has to reach the barrier before any of them is allowed to pass.
As for OpenMP, POSIX does not dictate the (default) stack size for a thread and thus can vary greatly. So better explicitly allocate enough stack to provide portability and avoid segmentation faults or data corruption. Use `pthread_attr_setstacksize` to set the desired stacksize in the attribute object used for creating the thread.
As for OpenMP, POSIX does not dictate the (default) stack size for a thread and thus can vary greatly. So better explicitly allocate enough stack to provide portability and avoid segmentation faults or data corruption use pthread_attr_setstacksize to set the desired stacksize in the attribute object used for creating the thread.
As for OpenMP, POSIX does not dictate the (default) stack size for a thread and thus can vary greatly. So better explicitly allocate enough stack to provide portability and avoid segmentation faults or data corruption.
As for OpenMP, POSIX does not dictate the (default) stack size for a thread and thus can vary greatly.

So better explicitly allocate enough stack to provide portability and avoid segmentation faults or data corruption.

use `pthread_attr_set-stacksize` to set the desired stacksize in the attribute object used for creating the thread.
MPI
Goals:

- Standardization: De-facto industry "standard" for message passing.
- Portability: Runs on a huge variety of platforms, allows for parallelization on very heterogeneous clusters.

First presented at supercomputing conference in 1993, initial releases in 1994 (MPI-1), 1998 (MPI-2), 2012 (MPI-3).

Many popular implementations e.g. OpenMPI (free), Intel MPI, MPICH.
Goals:

**Standardization**  De-facto industry “standard” for message passing.

**Portability**  Runs on a huge variety of platforms, allows for parallelization on very heterogeneous clusters.
MPI - History/Goals

- **Goals:**
  - **Standardization**  De-facto industry “standard” for message passing.
  - **Portability**  Runs on a huge variety of platforms, allows for parallelization on very heterogeneous clusters

- First presented at supercomputing conference in 1993, initial releases in 1994 (MPI-1), 1998 (MPI-2), 2012 (MPI-3)
MPI - History/Goals

- Goals:
  - **Standardization**: De-facto industry “standard” for message passing.
  - **Portability**: Runs on a huge variety of platforms, allows for parallelization on very heterogeneous clusters
- First presented at supercomputing conference in 1993, initial releases in 1994 (MPI-1), 1998 (MPI-2), 2012 (MPI-3)
- Many popular implementations e.g. OpenMPI (free), Intel MPI, MPICH
For compiling MPI programs, each implementation comes with specific "wrapper" scripts for the compilers, e.g.,

- GNU
- Intel
- OpenMPI

  - C:  `mpicc`
  - C++: `mpiCC` / `mpic++` / `mpicxx`
  - Fortran: `mpifort`

For Intel MPI

  - C:  `mpicc` / `mpigcc`
  - C++: `mpi` {`CC`, `c++`, `cxx`} / `mpigxx` / `mpiicpc`

For Fortran: `mpifort` / `mpiifort`

For running a MPI program, we use `mpirun`, which starts as many copies of the program as requested on nodes provided by the batch system, e.g.,

`mpirun -np 4 my program`
For compiling MPI programs, each implementation comes with specific “wrapper” scripts for the compilers, e.g.

<table>
<thead>
<tr>
<th></th>
<th>GNU</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMPI</td>
<td>C</td>
<td>mpicc</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>mpiCC/mpiC++/mpicxx</td>
</tr>
<tr>
<td></td>
<td>Fortran</td>
<td>mpifort</td>
</tr>
<tr>
<td>Intel MPI</td>
<td>C</td>
<td>mpicc/mpiGcc</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>mpi{CC,c++,cxx}/mpiGxx</td>
</tr>
<tr>
<td></td>
<td>Fortran</td>
<td>mpifort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mpi{CC,c++,cxx}/mpiicpc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mpifort*/mpiifort</td>
</tr>
</tbody>
</table>
For compiling MPI programs, each implementation comes with specific “wrapper” scripts for the compilers, e.g.

<table>
<thead>
<tr>
<th></th>
<th>GNU</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMPI</td>
<td>C</td>
<td>mpicc</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>mpiCC/mpi++/mpicxx</td>
</tr>
<tr>
<td></td>
<td>Fortran</td>
<td>mpifort</td>
</tr>
<tr>
<td>Intel MPI</td>
<td>C</td>
<td>mpicc/mpiGcc</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>mpi{CC,c++,cxx}/mpicxx</td>
</tr>
<tr>
<td></td>
<td>Fortran</td>
<td>mpifort</td>
</tr>
</tbody>
</table>

For running a MPI program, we use mpirun, which starts as many copies of the program as requested on nodes provided by the batch system, e.g.

mpirun -np 4 my_program
Before using any MPI routine (or better as early as possible), the MPI framework must be initialized by calling `MPI_Init(&argc, &argv)`.

At the end of your program, always call `MPI_Finalize()` to properly terminate/clean up the MPI execution environment.
Before using any MPI routine (or better as early as possible), the MPI framework must be initialized by calling `MPI_Init(&argc,&argv)`, which also broadcast the command line arguments to all processes.
MPI - Init & Finalize

- Before using any MPI routine (or better as early as possible), the MPI framework must be initialized by calling `MPI_Init(&argc,&argv)`, which also broadcast the command line arguments to all processes.
- At the end of your program, always call `MPI_Finalize()` to properly terminate/clean up the MPI execution environment.
MPI - Communicators

MPI uses communicators to define which processes may communicate with each other - in many cases, the predefined MPI COMM WORLD, which includes all MPI processes. Each process has a unique rank within the communicator. You can get the rank for a process with the command `MPI_Comm_rank(comm,&rank)` as well as the total size of the communicator (`MPI_Comm_size(comm,&size)`).
MPI - Communicators

MPI uses *communicators* to define which processes may communicate with each other - in many cases, the predefined MPI_COMM_WORLD, which includes all MPI processes.
MPI - Communicators

- MPI uses *communicators* to define which processes may communicate with each other - in many cases, the predefined MPI_COMM_WORLD, which includes all MPI processes.
- each process has a unique *rank* within the communicator. You can get the rank for a process with the command `MPI_Comm_rank(comm,&rank)` as well as the total size of the communicator (`MPI_Comm_size(comm,&size)`)
```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
  int numtasks, rank, len, rc;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  // initialize MPI
  MPI_Init(&argc,&argv);

  // get number of tasks
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

  // get my rank
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // this one is obvious
  MPI_Get_processor_name(hostname, &len);
  printf("Number of tasks= %d My rank= %d Running on %s\n", numtasks, rank, hostname);

  // do some work with message passing

  // done with MPI
  MPI_Finalize();
}
MPI - Communication

In MPI there are routines for Point-to-Point communication (i.e. from one process to exactly one other) as well as for collective communication. A Point-to-Point communication always consists of a send and a matching receive (or combined send/recv) routines. Those routines can be blocking and non-blocking, non-synchronous and synchronous.
In MPI there are routines for Point-to-Point communication (i.e. from one process to exactly one other) as well as for collective communication.
MPI - Communication

- In MPI there are routines for Point-to-Point communication (i.e. from one process to exactly one other) as well as for collective communication.
- A Point-to-Point communication always consists of a send and a matching receive (or combined send/recv) routines.
In MPI there are routines for Point-to-Point communication (i.e. from one process to exactly one other) as well as for collective communication.

- A Point-to-Point communication always consists of a `send` and a matching `receive` (or combined `send/recv`) routines.
- Those routines can be `blocking` and `non-blocking`, `non-synchronous` and `synchronous`.
MPI - Communication: Buffering

Path of a message buffered at the receiving process
MPI - Communication: Blocking vs Non-Blocking

**Blocking** A blocking send (`MPI_Send(...)`) waits until message is processed by local MPI (does not mean, that message has been received by other processes!), for waiting for confirmed processing by recipient, use synchronous blocking send (`MPI_Ssend(...)`); a blocking receive waits until data is received and ready for use.

**Non-Blocking** Non-blocking send/receive routines (`MPI_Isend(...)`, `MPI_Irecv(...)`, `MPI_Issend(...)`) work like their blocking counter-parts, but only request the operation and do not wait for its completion. Instead they return a *request* object that can be used to test/wait (e.g. `MPI_Wait(...)`, `MPI_Probe(...)`) until operation has been processed/certain status is reached for one or more request simultaneously.
MPI - Communication: Syntax

MPI_Isend(&buffer, count, datatype, dest, tag, comm, &request)

MPI_Recv(&buffer, count, datatype, src, tag, comm, &status)
MPI - Communication: Syntax

MPI_Isend(&buffer,count,datatype,dest,tag,comm,&request)

MPI_Recv(&buffer,count,datatype,src,tag,comm,&status)

with

buffer Memory block to send/receive data from/to
MPI - Communication: Syntax

MPI_Isend(&buffer, count, datatype, dest, tag, comm, &request)
MPI_Recv(&buffer, count, datatype, src, tag, comm, &status)

with

buffer Memory block to send/receive data from/to
count Number of data elements to be sent/maximum number to be received (see MPI_Get_count() for received amount)
**MPI - Communication: Syntax**

MPI\_Isend(&buffer,count,datatype,dest,tag,comm,&request)

MPI\_Recv(&buffer,count,datatype,src,tag,comm,&status)

with

- **buffer** Memory block to send/receive data from/to
- **count** Number of data elements to be sent/maximum number to be received (see MPI\_Get\_count() for received amount)
- **datatype** One of the predefined elementary MPI data types or derived data types

---

JAS (ICG, Portsmouth)  Implementation of Parallelization  May 9, 2018  40 / 48
### C Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>MPI_WCHAR</td>
<td>wchar_t - wide character</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG_INT</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</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_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG_LONG</td>
<td>unsigned long long int</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>
<tr>
<td>MPI_C_COMPLEX</td>
<td>float_Complex</td>
</tr>
<tr>
<td>MPI_C_FLOAT_COMPLEX</td>
<td>double_Complex</td>
</tr>
<tr>
<td>MPI_C_DOUBLE_COMPLEX</td>
<td>double_Complex</td>
</tr>
<tr>
<td>MPI_C_LONG_DOUBLE_COMPLEX</td>
<td>long double_Complex</td>
</tr>
<tr>
<td>MPI_C_BOOL</td>
<td>_Bool</td>
</tr>
<tr>
<td>MPI_INT8_T</td>
<td>int8_t</td>
</tr>
<tr>
<td>MPI_INT16_T</td>
<td>int16_t</td>
</tr>
<tr>
<td>MPI_INT32_T</td>
<td>int32_t</td>
</tr>
<tr>
<td>MPI_INT64_T</td>
<td>int64_t</td>
</tr>
<tr>
<td>MPI_UINT8_T</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MPI_UINT16_T</td>
<td>uint16_t</td>
</tr>
<tr>
<td>MPI_UINT32_T</td>
<td>uint32_t</td>
</tr>
<tr>
<td>MPI_UINT64_T</td>
<td>uint64_t</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>8 binary digits</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>data packed or unpacked with MPI_Pack() / MPI_Unpack</td>
</tr>
</tbody>
</table>
MPI - Communication: Syntax

```
MPI_Isend(&buffer,count,datatype,dest,tag,comm,&request)
MPI_Recv(&buffer,count,datatype,src,tag,comm,&status)
```

with

- **buffer**: Memory block to send/receive data from/to
- **count**: Number of data elements to be sent/maximum number to be received (see `MPI_Get_count()` for received amount)
- **datatype**: One of the predefined elementary MPI data types or derived data types
- **dest/src**: Rank of the communication partner (within the used shared communicator); wildcard `MPI_ANY_SOURCE`
MPI - Communication: Syntax

MPI_Isend(&buffer, count, datatype, dest, tag, comm, &request)

MPI_Recv(&buffer, count, datatype, src, tag, comm, &status)

with

- **buffer**: Memory block to send/receive data from/to
- **count**: Number of data elements to be sent/maximum number to be received (see MPI_Get_count() for received amount)
- **datatype**: One of the predefined elementary MPI data types or derived data types
- **dest/src**: Rank of the communication partner (within the used shared communicator); wildcard MPI_ANY_SOURCE
- **tag**: Arbitrary non-negative (short) integer; same for send & receive (unless wildcard MPI_ANY_TAG used for recv)
MPI - Communication: Syntax

MPI_Isend(&buffer, count, datatype, dest, tag, comm, &request)
MPI_Recv(&buffer, count, datatype, src, tag, comm, &status)

with

buffer Memory block to send/receive data from/to
count Number of data elements to be sent/maximum number to be received (see MPI_Get_count() for received amount)
datatype One of the predefined elementary MPI data types or derived data types
dest/src Rank of the communication partner (within the used shared communicator); wildcard MPI_ANY_SOURCE
tag arbitrary non-negative (short) integer; same for send & receive (unless wildcard MPI_ANY_TAG used for recv)
comm communicator
MPI - Communication: Syntax

MPI_Isend(&buffer, count, datatype, dest, tag, comm, &request)
MPI_Recv(&buffer, count, datatype, src, tag, comm, &status)

with

- **buffer**: Memory block to send/receive data from/to
- **count**: Number of data elements to be sent/maximum number to be received (see MPI_Get_count() for received amount)
- **datatype**: One of the predefined elementary MPI data types or derived data types
- **dest/src**: Rank of the communication partner (within the used shared communicator); wildcard MPI_ANY_SOURCE
- **tag**: Arbitrary non-negative (short) integer; same for send & receive (unless wildcard MPI_ANY_TAG used for recv)
- **comm**: Communicator
- **request**: Allocated request structure used to communicate progress of comm. process for non-blocking routines
MPI - Communication: Syntax

MPI_Send(&buffer, count, datatype, dest, tag, comm, &request)
MPI_Recv(&buffer, count, datatype, src, tag, comm, &status)

with

- **buffer**: Memory block to send/receive data from/to
- **count**: Number of data elements to be sent/maximum number to be received (see MPI_Get_count() for received amount)
- **datatype**: One of the predefined elementary MPI data types or derived data types
- **dest/src**: Rank of the communication partner (within the used shared communicator); wildcard MPI_ANY_SOURCE
- **tag**: Arbitrary non-negative (short) integer; same for send & receive (unless wildcard MPI_ANY_TAG used for recv)
- **comm**: Communicator
- **request**: Allocated request structure used to communicate progress of comm. process for non-blocking routines
- **status**: Allocated status structure containing source & tag for receive routines
MPI - Collective Communication

- More efficient data exchange with multiple processes
- always involves all processes in one communicator
- can only used with predefined datatypes
- can be blocking or non-blocking (since MPI-3)
MPI - Collective Communication [Allgather, Alltoall]

The diagrams illustrate the concepts of Allgather and Alltoall in MPI. The left side shows the data distribution among processes, while the right side shows the transformed data after the collective communication operations.

- **Allgather**: Each process gathers data from all other processes.
- **Alltoall**: Each process exchanges data with all other processes.

The diagrams use different symbols to represent the data and processes before and after the operations.
MPI - Collective Communication [Reduce, Allreduce]

\[
S_i = A_i \circ B_i \circ C_i \circ D_i \circ E_i \circ F_i
\]
MPI - Collective Communication [Reduce, Allreduce]

\[ S_i = A_i \odot B_i \odot C_i \odot D_i \odot E_i \odot F_i \]

<table>
<thead>
<tr>
<th>MPI Reduction Operation</th>
<th>C Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bit-wise AND</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>bit-wise OR</td>
</tr>
<tr>
<td>MPI_XOR</td>
<td>logical XOR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max value and location</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min value and location</td>
</tr>
</tbody>
</table>
#include "mpi.h"
#include <stdio.h>
#include <math.h>

main(int argc, char *argv[]) {
    int numtasks, rank, n, i, root, chunk;
    int a[100], b[100], result, final_result;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    root = 0;
    result = 0;
    n = 100;
    chunk = ceil(n / numtasks);

    for (i=rank*chunk; i<n; i++) {
        result = result + (a[i] * b[i]);
    }

    MPI_Reduce(&result,&final_result,1,MPI_INT,MPI_SUM, root,MPI_COMM_WORLD);

    if (rank == root) {
        printf("result = %d\n",final_result);
    }

    MPI_Finalize();
}
As shown in the 'Introduction' talk, you can combine Multithreading and Multiprocessing, **BUT** ...

- you have to check whether your MPI implementations is thread-safe. MPI libraries vary in their level of thread support:
  - `MPI_THREAD_SINGLE` no multithreading supported
  - `MPI_THREAD_FUNNELED` only main thread may make MPI calls
  - `MPI_THREAD_SERIALIZED` MPI calls are serialized i.e. cannot be processed concurrently
  - `MPI_THREAD_MULTIPLE` thread-safe
Debugging
Debugging

As for analyzing and tuning parallel program performance, debugging can be much more challenging for parallel programs than for serial programs (in particular for MPI programs) and again, unfortunately, covering this topic in any detail would go beyond the scope of this introduction to parallel program.
As for analyzing and tuning parallel program performance, debugging can be much more challenging for parallel programs than for serial programs (in particular for MPI programs).
Debugging

- As for analyzing and tuning parallel program performance, debugging can be much more challenging for parallel programs than for serial programs (in particular for MPI programs).
- And again, unfortunately, covering this topic in any detail would go beyond the scope of this introduction to parallel program.
As for analyzing and tuning parallel program performance, debugging can be much more challenging for parallel programs than for serial programs (in particular for MPI programs).

And again, unfortunately, covering this topic in any detail would go beyond the scope of this introduction to parallel program.

While popular open source debuggers like gdb provide facilities for debugging multi-threaded programs, MPI debugging relies on commercial solutions like DDT or TotalView.