Implementation of Parallelization

OpenMP, PThreads and MPI

Jascha Schewtschenko

Institute of Cosmology and Gravitation, University of Portsmouth

May 9, 2018

Outline

OpenMP

OpenMP - Goals

OpenMP - Goals

Standardization provide standard for vatiety of platforms/shared-mem architectures

OpenMP - Goals

Standardization provide standard for vatiety 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 vatiety 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 vatiety 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

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:

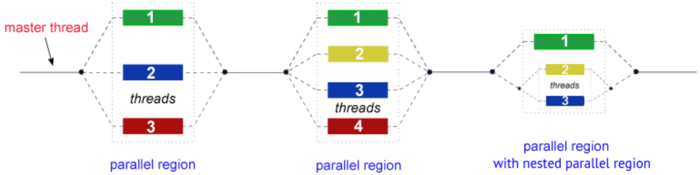
  e.g. `gcc -fopenmp`

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

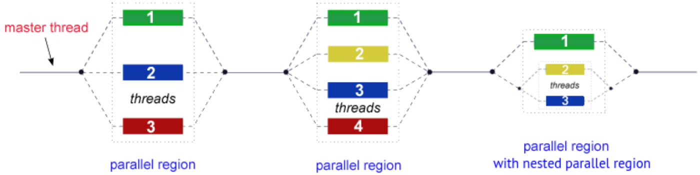


OpenMP - Structure & Implementations

- 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

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)
```

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

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

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)

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.

OpenMP - Environment variables

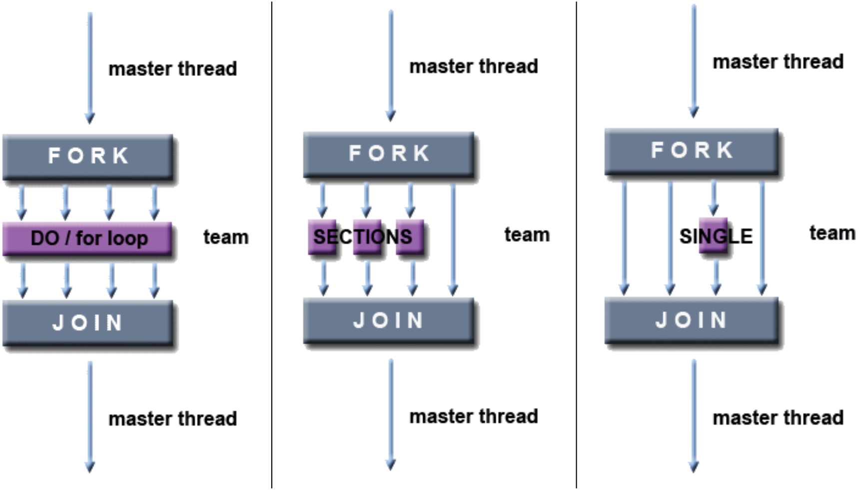
OpenMP - Environment variables

- 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`).

OpenMP - Environment variables

- 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



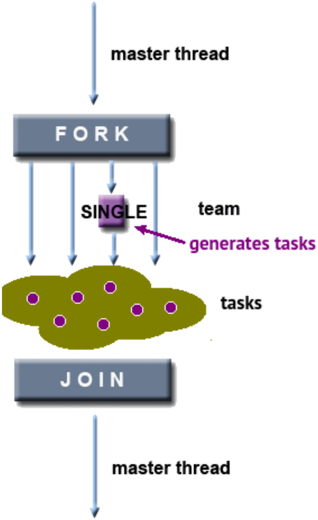
OpenMP - Worksharing (examples)

```
#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 for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }   /* end of parallel region */

}
```

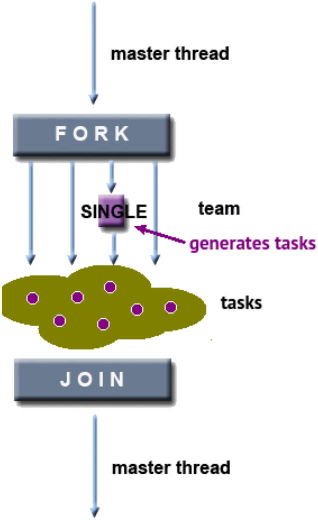
```
#include <omp.h>
#define N 1000

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

int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel shared(a,b,c,d) 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

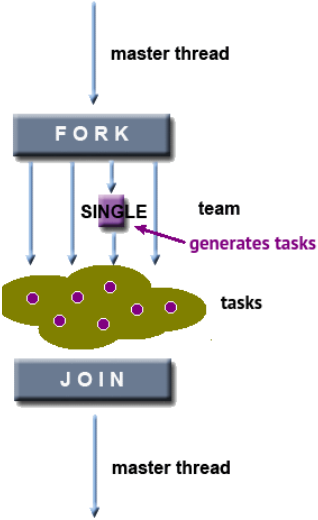


OpenMP - advanced Worksharing



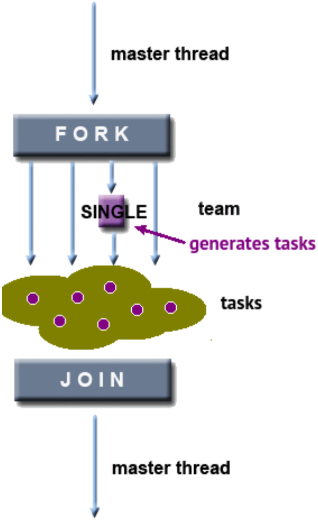
- defines explicit tasks similar to sections that are generated (usually by a single task) and then deferred to any thread in the team via a queue/scheduler

OpenMP - advanced Worksharing



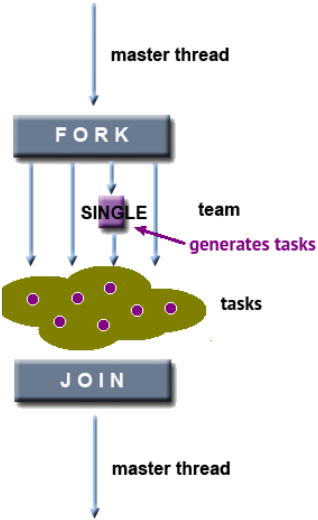
- defines explicit tasks similar to sections that are generated (usually by a single task) and then deferred to any thread in the team via a queue/scheduler
- tasks are not necessarily tied to a single thread, can be e.g. postponed or migrated to other threads

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)




```
#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;
}
```

OpenMP - Synchronization / Flow control

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:

OpenMP - Synchronization / Flow control

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`

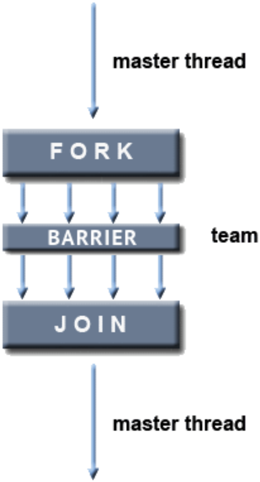
OpenMP - Synchronization / Flow control

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:

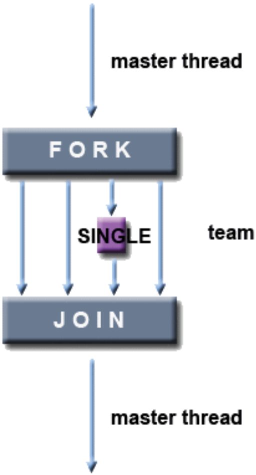
- Compiler Directives:
  - (for general parallel regions) e.g. `cancel`,`single`,`master`,`critical`,`atomic`, `barrier`
  - (for loops) `ordered`
  - (for tasks) `taskwait`, `taskyield`
- Runtime Library Routines: `omp_set_lock`,`omp_unset_lock`,`omp_test_lock`

OpenMP - Synchronization / Flow control (RESTRICTION)

BARRIER

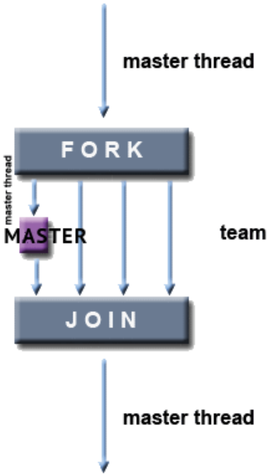


SINGLE



implicit BARRIER

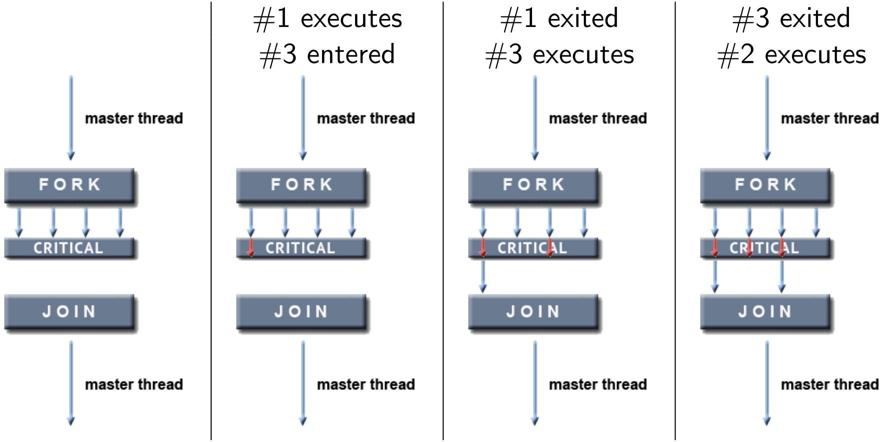
MASTER



NO implicit BARRIER

OpenMP - Synchronization / Flow control (MUTEX)

CRITICAL / ATOMIC



- CRITICAL,ATOMIC exclusive for ALL threads, not just team
- CRITICAL regions can be named, regions with same name treated as same region

OpenMP - Memory management (CLAUSES)

- 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`)

OpenMP - Memory management (CLAUSES)

- 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

OpenMP - Memory management (CLAUSES)

- 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

OpenMP - Memory management (CLAUSES)

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

```
#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)

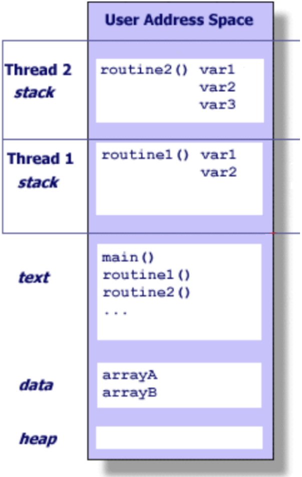
OpenMP - Memory management (FLUSHING DATA)

- 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

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.

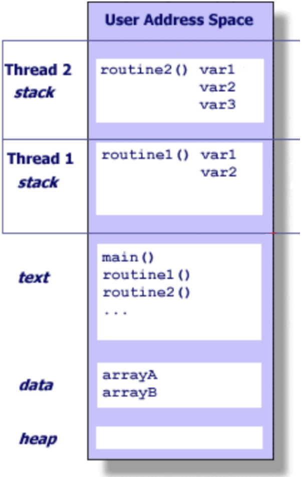
OpenMP - Memory management (STACK)



OpenMP - Memory management (STACK)

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

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

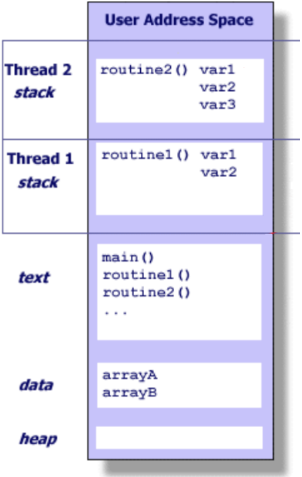


OpenMP - Memory management (STACK)

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

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

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

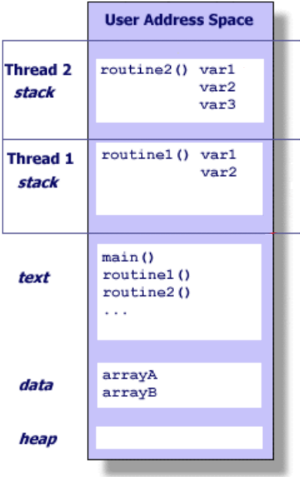


OpenMP - Memory management (STACK)

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

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

- 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 an 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

PThreads - 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

PThreads - 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
- historically implementations focused on Unix as OS, but implementations also exist now for others e.g. for Windows

PThreads - Compiling & Running

PThreads - Compiling & Running

- Like for OpenMP, POSIX Threads are included in most recent compiler suites by default

PThreads - Compiling & Running

- Like for OpenMP, POSIX Threads are included in most recent compiler suites by default
- To enable these included libraries, use e.g.

```
icc -pthread
gcc -pthread
```

`icc -pthread` for INTEL (Linux)
`gcc -pthread` for GNU (Linux)

PThreads - API

PThreads - API

- 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 - API

- 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

PThreads - Thread management: Creation & Termination

- 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)

PThreads - Thread management: Creation & Termination

- 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)

PThreads - Thread management: Example 1

```
#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);
}
```

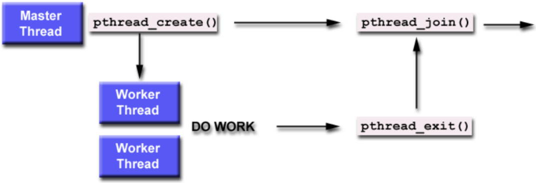
PThreads - Thread management: Joining & Detaching

PThreads - Thread management: Joining & Detaching

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

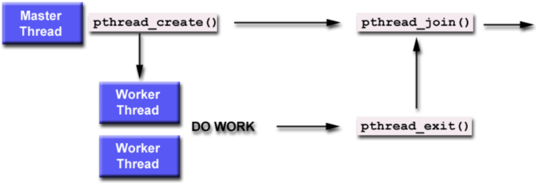
PThreads - Thread management: Joining & Detaching

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



PThreads - Thread management: Joining & Detaching

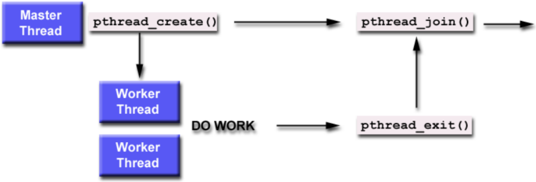
- "Joining" threads allows the master thread to synchronize with its worker threads on completion of their task



- threads can be declared "joinable" on creation

PThreads - Thread management: Joining & Detaching

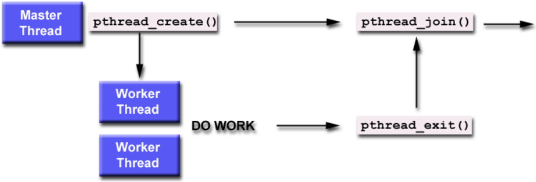
- "Joining" threads allows the master thread to synchronize with its worker threads on completion of their task



- 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

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

PThreads - Mutexes

- Mutexes work in similar way as the OpemMP locks: once claimed by one thread, other threads encountering it will be hold until the mutex released again.

PThreads - Joining & Mutexes: Example

```
#define NUMTHRDS 4
#define VECLEN 100000
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

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 %d to %d:  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));
```

```
[...]

pthread_mutex_init(&mutexsum, NULL);

/* Create threads to perform the dotproduct  */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for(i=0;i<NUMTHRDS;i++)
  {
  /* Each thread works on a different set of data.
   * The offset is specified by 'i'. The size of
   * the data for each thread is indicated by VECLEN.
   */
   pthread_create(&callThd[i], &attr, dotprod, (void *)i);
   }

pthread_attr_destroy(&attr);
/* Wait on the other threads */

for(i=0;i<NUMTHRDS;i++) {
  pthread_join(callThd[i], &status);
  }
/* After joining, print out the results and cleanup */

printf ("Sum =  %f \n", dotstr.sum);
free (a);
free (b);
pthread_mutex_destroy(&mutexsum);
pthread_exit(NULL);
}
```

PThreads - Condition variables

PThreads - Condition variables

- Conditions variables control the flow of threads like Mutexes

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 send a signal (`pthread_cond_signal()`) through the condition variable to continue.

PThreads - Synchronization: Barriers

PThreads - Synchronization: Barriers

- POSIX Threads also feature a synchronization barrier similar to OpenMP.

PThreads - Synchronization: Barriers

- 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.

PThreads - Memory management

PThreads - Memory management

- As for OpenMP, POSIX does not dictate the (default) stack size for a thread and thus can vary greatly.

PThreads - Memory management

- 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

PThreads - Memory management

- 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.

MPI

MPI - History/Goals

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

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

MPI - Compiling & Running

MPI - Compiling & Running

- 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	
Intel MPI	**C**	mpicc/mpigcc	mpicc/mpiicc
	C++	mpi{CC,c++,cxx}/mpigxx	mpi{CC,c++,cxx}/mpiicpc
	Fortran	mpifort	mpifort*/mpiifort

MPI - Compiling & Running

- 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	
Intel MPI	**C**	mpicc/mpigcc	mpicc/mpiicc
	C++	mpi{CC,c++,cxx}/mpigxx	mpi{CC,c++,cxx}/mpiicpc
	Fortran	mpifort	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
```

MPI - Init & Finalize

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

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 - 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)`)

MPI - Example 1

```
#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

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

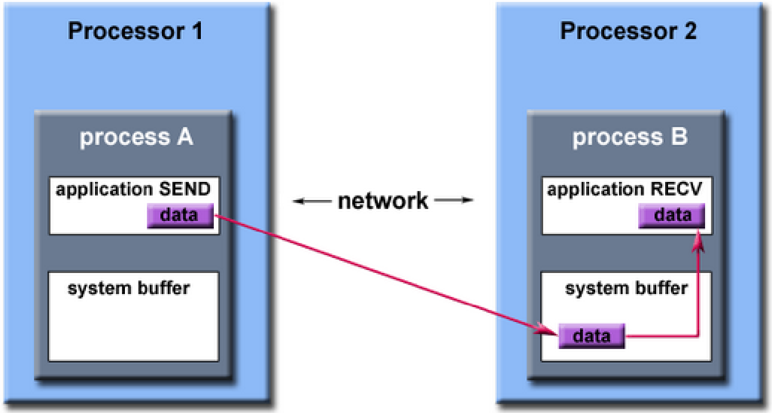
MPI - Communication

- 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

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*

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

MPI - Communication: Syntax

C Data Types	
MPI_CHAR	char
MPI_WCHAR	wchar_t - wide character
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT MPI_LONG_LONG	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_C_COMPLEX MPI_C_FLOAT_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_C_BOOL	_Bool
MPI_INT8_T MPI_INT16_T MPI_INT32_T MPI_INT64_T	int8_t int16_t int32_t int64_t
MPI_UINT8_T MPI_UINT16_T MPI_UINT32_T MPI_UINT64_T	uint8_t uint16_t uint32_t uint64_t
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

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

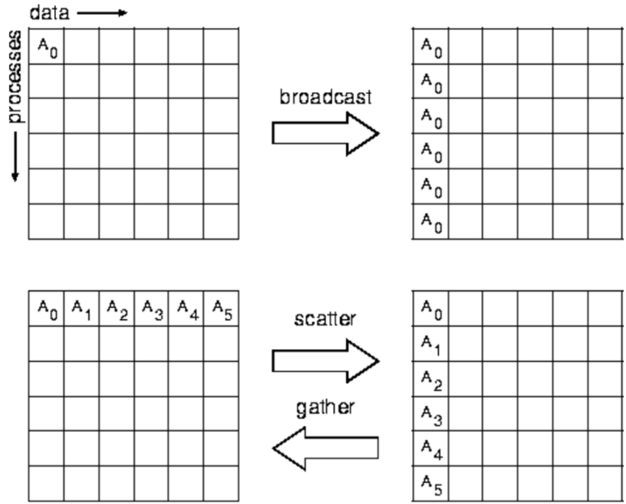
with

- **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
- **status** allocated status structure containing source & tag for receive routines

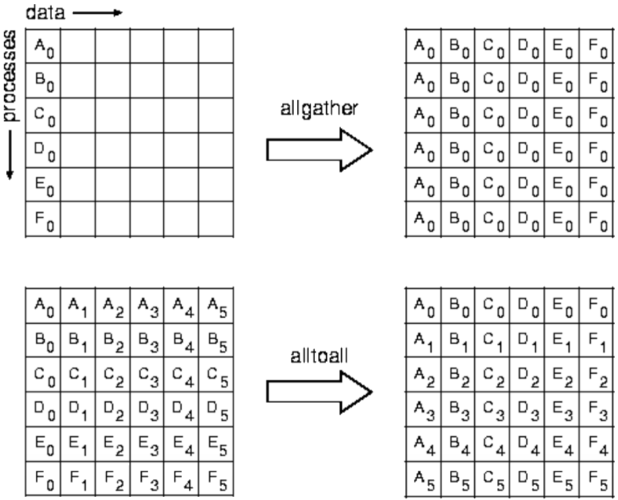
MPI - Collective Communication

- 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)

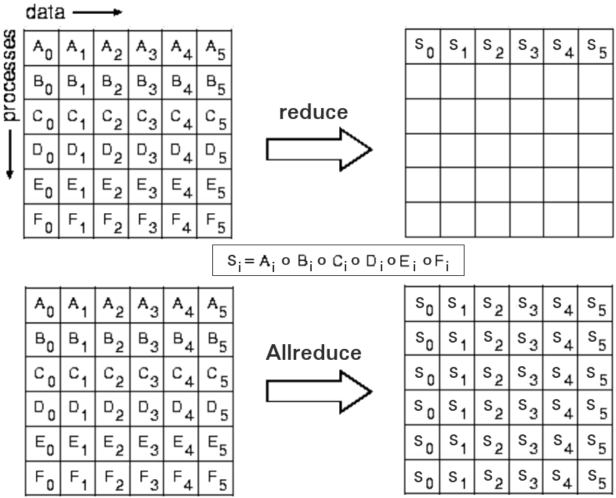
MPI - Collective Comm. [Broadcast/Scatter/Gather]



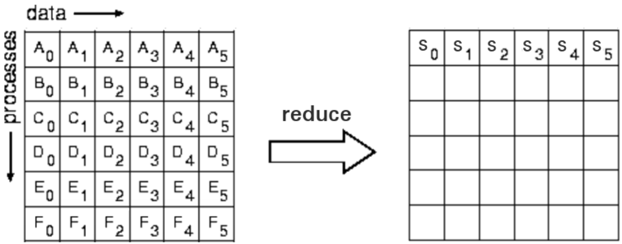
MPI - Collective Communication [Allgather,Alltoall]



MPI - Collective Communication [Reduce,Allreduce]



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 Reduction Operation		C Data Types
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double
MPI_MINLOC	min value and location	float, double and long double

MPI - Collective Communication: Example

```
#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<100; 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();
}
```

MPI - Multithreading

- 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

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)

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.

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