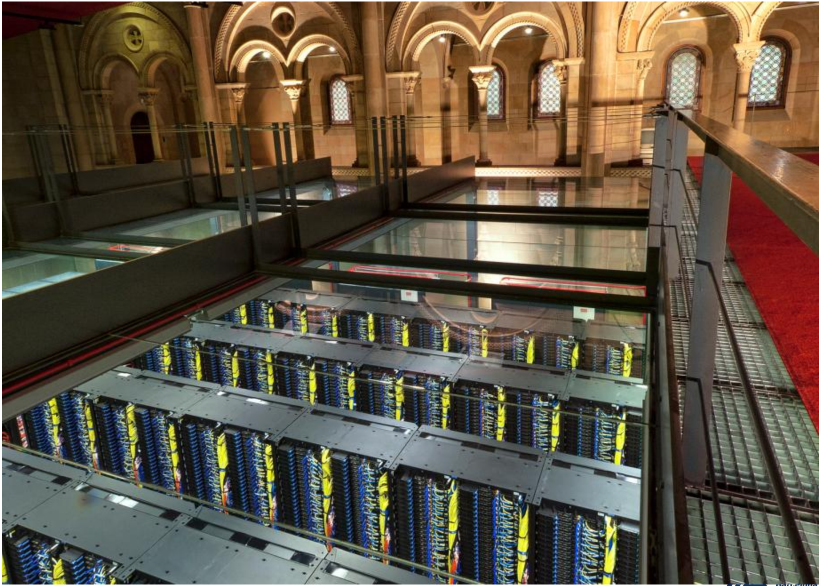
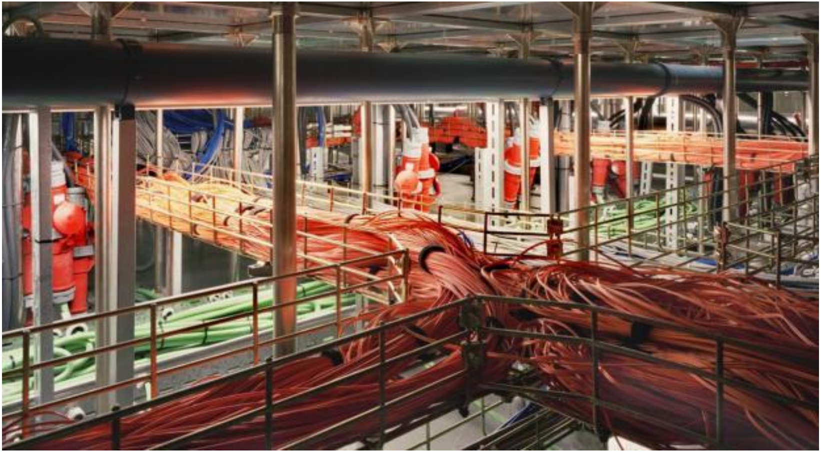
POSIX Threads and OpenMP tasks

Jimmy Aguilar Mena

February 16, 2018

Barcelona Supercomputing Center
Centro Nacional de Supercomputación





Two simple schemas

Independent functions

```
#include<pthread.h>
#include<stdio.h>

void function1() {
  [...];
}

void function2() {
  [...];
}

int main(int argc, char **argv) {
  function1();
  function2();

  return 0;
}
```

Recursion

```
#include<pthread.h>
#include<stdio.h>

int function(int x, int y) {
  [...]
  int a = function(x+y);
  int b = function(x-y);
  [...]
  return 0;
}

int main(int argc, char **argv) {
  [...]
  function(a, b);

  return 0;
  }
```

Threads vs Processes

Start

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

https://computing.llnl.gov/tutorials/pthreads/

Threads vs Processes

Communication

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

https://computing.llnl.gov/tutorials/pthreads/

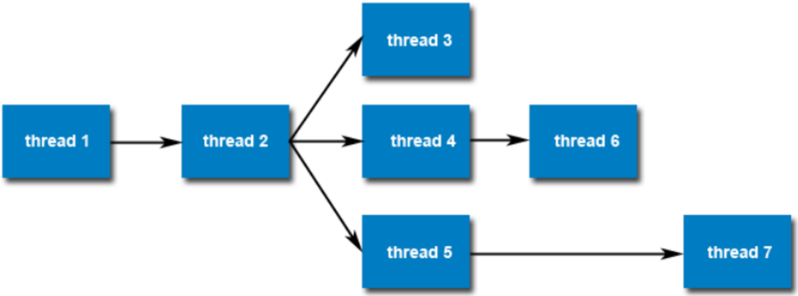
POSIX Threads

POSIX Threads

A POSIX thread is a single flow of control within a process. It shares a virtual address space with other threads in the same process. The following are per-thread attributes:

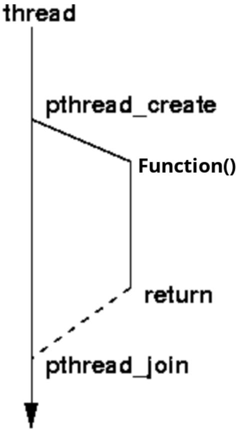
- execution stack (within shared address space)
- set of blocked signals (the signal mask)
- set of signals pending for the thread
- scheduling policy and priority
- errno value
- thread-specific key-to-attribute mapping

Threads workflow



PTHREADS

C interface

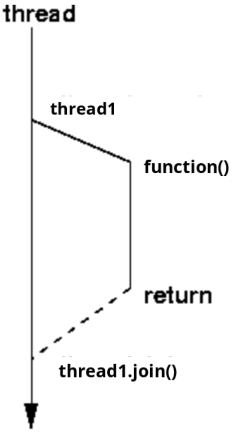



```c
#include<pthread.h>
#include<stdio.h>

// create the Function to be executed as a thread
void *Function(void *ptr)
{
  int type = (int) ptr;
  printf("Thread-%d\n",type);
  return  ptr;
}

int main(int argc, char **argv)
{
  pthread_t thread1; // create the thread objs

  int thr = 1;        // input

  pthread_create(&thread1, NULL, *Function, (void *) thr);

  printf("Thread-Master\n");

  pthread_join(thread1,NULL); // wait for threads to finish

  return 0;
}
```

Build: gcc pthread.c -o pthread.x -lpthread

PTHREADS

C++11 interface




```
#include <thread>
#include <cstdio>
#include <cstdlib>

using namespace std;
void function(int i)
{
  cout << "Thread-"<<i<< endl;
}

int main(int argc, char** argv)
{
  int  thr = 1;  //input

  thread thread1(function, i);

  cout << "Thread-Master"<< endl;

  threadlist[i].join();

  return 0;
}
```

Build: g++ -std=c++11 thread.cxx -o thread.x -pthread

Create Threads

pthread_create

```
int pthread_create(pthread_t *new_thread_ID,
            const pthread_attr_t *attr,
            void * (*func)(void *), void *arg);
```

Needs to specify a place to store the ID of the new thread, the procedure that the thread should execute, optionally some thread creation attributes, and optionally an argument for the thread.

Attributes

Creating attributes to default values

```
pthread_attr_t tattr;
int ret;

// initialize an attribute to the default
ret = pthread_attr_init(&tattr);

// set the thread detach state (for example)
ret = pthread_attr_setdetachstate(&tattr,
                                  PTHREAD_CREATE_DETACHED);

    ...

// destroy attribute
ret = pthread_attr_destroy(&tattr);
```

Joining threads

```
#include <pthread.h>
int pthread_join(pthread_t target_thread,
                 void **status);
```

By default, threads are created joinable. This means that some other thread is required to call pthread_join to collect a terminated thread, in a fashion similar to the requirement for a parent process to collect status for terminated child processes.

POSIX threads C API

Protecting critical sections (Mutex)

Mutex usage

```
pthread_mutex_t M;
pthread_mutex_lock (&M);
... critical section ...
pthread_mutex_unlock (&M);
```

A mutex is a memory-based data object that is used to implement mutual exclusion. The intent it so provide the kind of protection that is needed to implement a monitor. Mutexes are designed to provide the mutual needed for a monitor.

Once operations

pthread_once_t

```
void init_routine () {
  pthread_mutex_init (&M, NULL);
}

void initialize () {
  static pthread_once_t init_flag = PTHREAD_ONCE_INIT;
  // is initialized at process start time
  pthread_once (&init_flag , init_routine);
  pthread_mutex_lock (&M);
  // now initialize other global data
  pthread_mutex_unock (&M);
}
```

The variable init_flag is used to indicate whether the initialization has been done yet. The difference is that the function pthread_once is guaranteed to atomically test and modify the flag.

Programming with pthreads

1. Prevent overhead even if it is small.
2. As usual, programming for the the architecture improves performance.
3. You can create as many threads as you need, but the machine has a fix number of cores.
4. Memory affinity (NUMA) and core affinity (taskset) cares.
5. Before doing a serious application READ THE DOCUMENTATION. Don't reinvent the wheel!!

Tasks vs Threads

Task

A task is something you want done.

Threads

A thread is one of the many possible workers which performs that task.

OpenMP specification version 3.0 introduced a new feature called tasking. Tasks are generated dynamically in recursive structures or while loops.

The task construct can be placed anywhere in the program; whenever a thread encounters a task construct, a new task is generated.

Task Execution

- The task construct defines a section of code
- Inside a parallel region, a thread encountering a task construct will package up the task for execution
- Some thread in the parallel region will execute the task at some point in the future
- If task execution is deferred, then the task is placed in a pool of tasks.
- Tasks can be nested: i.e. a task may itself generate tasks
- The OMP parallel construct creates “implicit” tasks.

Syntax

C++

```
#pragma omp task [clauses]
structured-block
```

Fortran

```
!$OMP TASK [clauses]
structured block
!$OMP END TASK
```

Clauses

- if(scalar-expression) Creates conditionally
- final(scalar-expression) The generated task will be a final task
- untied Any thread in the team can resume the task region after a suspension
- default(shared | none) As expected.
- mergeable the generated task is a mergeable task.
- private(list) As expected.
- firstprivate(list) As expected. (makes a copy during the generation.)
- shared(list) As expected.
- depend(dependence-type : list) Dependency with variables.
- priority(priority-value) Hint for the priority of the generated task

Hello World

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>

void function1() {
  printf("Hello␣from␣function␣1\n");
  sleep(5);
  printf("Bye␣from␣function␣1\n");
}

void function2() {
  printf("Hello␣from␣function␣2\n");
  sleep(5);
  printf("Bye␣from␣function␣1\n");
}

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

  #pragma omp parallel
  #pragma omp single
  {
    #pragma omp task
    function1();

    #pragma omp task
    function2();
  }
  return 0;
}
```

Tasks synchronization

Thread barriers

Applies to all tasks generated in the current parallel region up to the barrier.

Taskwait directive

Wait until all tasks defined in the current task have completed.

C	`#pragma omp taskwait`
Fortran	`!$OMP TASKWAIT`

- This is specially useful in recursive applications.
- Like barrier, it must not be executed conditionally.
- You are recommended **not** to use barrier with active tasks

Nested tasks

```
#pragma omp task private(B)
{
  B = ...
  #pragma omp task shared (B)
  {
    compute(B);
  }
  ...
  #pragma omp taskwait
}
```

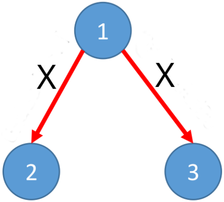
- Every outer task has its own copy of B
- All inner tasks use their parent task's copy of B
- Taskwait ensures these don't go out of scope

Parallel task loading

```
#pragma omp parallel
{
  #pragma omp for private(p)
  for ( int i =0; i < N ; ++i) {
      #pragma omp task firstprivate(p)
      {
      task_code_function(i);
      }
    }
  }
}
```

Dependencies

```
void process_in_parallel()
{
  #pragma omp parallel
  #pragma omp single
  {
    int x = 1;
    #pragma omp task shared(x, ...) depend(out:x)
    Task1(...);
    #pragma omp task shared(x, ...) depend(in:x)
    Task2_dependent(...);
    #pragma omp task shared(x, ...) depend(in:x)
    Task3_dependent(...);
  }
}
```



Using Tasks

1. Getting the data attribute scoping right can be quite tricky
   - default scoping rules different from other constructs
   - as ever, using default(none) is a good idea
2. Don't use tasks for things already well supported by OpenMP
   - e.g. standard do/for loops
   - the overhead of using tasks is greater
3. Don't expect miracles from the runtime
   - best results usually obtained where the user controls the number and granularity of tasks