Advanced OpenMP

Other APIs

What's wrong with OpenMP?

- OpenMP is designed for programs where you want a fixed number of threads, and you always want the threads to be consuming CPU cycles.
  - cannot arbitrarily start/stop threads
  - cannot put threads to sleep and wake them up later

- OpenMP is good for programs where each thread is doing (more-or-less) the same thing.

- Although OpenMP supports C++, it's not especially OO friendly
  - though it is gradually getting better.

- OpenMP doesn't support Java

- Lacks some of the higher level functionality of OpenMP

- Performance of Java may not match that of Fortran or C

- OO model doesn't suit some types of scientific computation very well

- Some HPC systems don't support Java
  - there is no standardised interface for message passing in Java

What are the alternatives?

- POSIX threads

- C++ threads

- Intel TBB

- Cilk

- OpenCL

(not an exhaustive list!)

POSIX threads

- POSIX threads (or Pthreads) is a standard library for shared memory programming without directives.
  - Part of the ANSI/IEEE 1003.1 standard (1996)

- Interface is a C library
  - no standard Fortran interface
  - can be used with C++, but not OO friendly

- Widely available
  - even for Windows
  - typically installed as part of OS
  - code is pretty portable

- Lots of low-level control over behaviour of threads

- Execution model similar to Java threads

Thread forking

```
#include <pthread.h>


int pthread_create(

        pthread_t *thread,

        const pthread_attr_t *attr,

        void*(*start_routine, void*),

        void *arg)
```

- Creates a new thread:
  - first argument returns a pointer to a thread descriptor.
  - can set attributes.
  - new thread will execute `start_routine(arg)`
  - return value is error code.

Thread joining

```
#include <pthread.h>


int pthread_join(

        pthread_t thread,

        void **value_ptr)
```

- Waits for the specified thread to finish.
  - thread finishes when `start_routine` exits
  - second argument holds return value from `start_routine`

Synchronisation

- Barriers

- Mutex locks
  – Behaviour is essentially the same as the OpenMP lock routines.

- Condition variables
  – Behaviour is essentially the same as wait/notify in Java

Hello World

```c
#include <pthread.h>

#define NTHREADS 5

int i, threadnum[NTHREADS];

pthread_t tid[NTHREADS];


for (i=0; i<NTHREADS; i++) {

  threadnum[i]=i;

  pthread_create(&tid[i], NULL, hello, &threadnum[i]);

}


for (i=0; i<NTHREADS; i++)

   pthread_join(tid[i], NULL);
```

```
void* hello (void *arg) {

    int myid;


    myid = *(int *)arg;

    printf("Hello world from thread %d\n", myid);


    return (0);

}
```

C++11 threads

- Library for multithreaded programming built in to C++11 standard

- Similar functionality to POSIX threads
  - but with a proper OO interface
  - based quite heavily on BOOST threads library

- Portable
  - depends on C++11 support, OK in gcc, Intel, clang, MS

- Threads are C++ objects

- Synchronisation
  - mutex locks
  - condition variables
  - C++11 atomics

Hello world

```cpp
#include <thread>
#include <iostream>
#include <vector>

void hello(){
    std::cout << "Hello from thread " << std::this_thread::get_id() <<
    std::endl;
}
int main(){
    std::vector<std::thread> threads;
    for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread(hello));
    }
    for(auto& thread : threads){
        thread.join();
    }
}
```

Intel Thread Building Blocks (TBB)

- C++ library for multithreaded programming

- Offers somewhat higher level of abstraction that POSIX/C++11 threads
  - notion of tasks rather that explicit threads
  - support for parallel loops and reductions
  - support for concurrency on containers

- Moderately portable
  - support for Intel and gcc compilers on Linux and Mac OS X, Intel and Visual C++ on Windows
  - no build required to install

Hello World

```cpp
#include <iostream>
#include <tbb/parallel_for.h>

using namespace tbb;

class Hello
{
public:
void operator()(int x) const {
std::cout << "Hello world\n";
}
};

int main()
{
// parallelizing:
// for(int i = 0; i < 2; ++i) { ... }
parallel_for(0, 2, 1, Hello());

return 0;
}
```

Cilk

- Very minimal API which supports spawning and joining of tasks
  - C/C++ with a few extra keywords

- Commercial implementation by Intel
  - Intel Cilk Plus, built in to Intel C++ compiler
  - not very portable

- Support for parallel loops and reductions
  - No locks, but can use pthread or TBB mutexes.

- Still unclear whether it is really useful for real-world applications!

Hello World

```c
#include <stdio.h>

#include <cilk/cilk.h>


static void hello(){

    printf("Hello ");

}


int main(){

    cilk_spawn hello();

    cilk_sync;

}
```

OpenCL

- API designed for programming heterogeneous systems (GPUs, DSPs, etc).
  - but can also execute on regular CPUs

- Open standard administered by Khronos Group

- Based on C99 with some extra keywords, large set of runtime library routines

- CPU implementations from Intel, IBM

- Very low level (c.f. CUDA), lots of boiler-plate code required

- Performance (and performance portability) not convincingly demonstrated....

Hello World

```
__kernel void hello(__global char* string)

{

string[0] = 'H';

string[1] = 'e';

string[2] = 'l';

string[3] = 'l';

string[4] = 'o';

string[5] = ',';

string[6] = ' ';

string[7] = 'W';

string[8] = 'o';

string[9] = 'r';

string[10] = 'l';

string[11] = 'd';

string[12] = '!';

string[13] = '\0';

}
```

```c
#include <stdio.h>

#include <stdlib.h>

#include <CL/cl.h>


#define MEM_SIZE (128)

#define MAX_SOURCE_SIZE (0x100000)


int main()

{

cl_device_id device_id = NULL;

cl_context context = NULL;

cl_command_queue command_queue = NULL;

cl_mem memobj = NULL;

cl_program program = NULL;

cl_kernel kernel = NULL;

cl_platform_id platform_id = NULL;

cl_uint ret_num_devices;

cl_uint ret_num_platforms;

cl_int ret;


char string[MEM_SIZE];


FILE *fp;

char fileName[] = "./hello.cl";

char *source_str;

size_t source_size;
```

```c
/* Load the source code containing
the kernel*/
fp = fopen(fileName, "r");
if (!fp) {
fprintf(stderr, "Failed to load
kernel.\n");
exit(1);
}
source_str =
(char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1,
MAX_SOURCE_SIZE, fp);
fclose(fp);
/* Get Platform and Device Info */
ret = clGetPlatformIDs(1,
&platform_id, &ret_num_platforms);
ret = clGetDeviceIDs(platform_id,
CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);

/* Create OpenCL context */
context = clCreateContext(NULL, 1,
&device_id, NULL, NULL, &ret);
/* Create Command Queue */
command_queue =
clCreateCommandQueue(context,
device_id, 0, &ret);
/* Create Memory Buffer */
memobj = clCreateBuffer(context,
CL_MEM_READ_WRITE,MEM_SIZE *
sizeof(char), NULL, &ret);
/* Create Kernel Program from the
source */
program =
clCreateProgramWithSource(context, 1,
(const char **)&source_str,
(const size_t *)&source_size, &ret);
```

```c
/* Create OpenCL context */
context = clCreateContext(NULL, 1,
&device_id, NULL, NULL, &ret);
/* Create Command Queue */
command_queue =
clCreateCommandQueue(context,
device_id, 0, &ret);
/* Create Memory Buffer */
memobj = clCreateBuffer(context,
CL_MEM_READ_WRITE,MEM_SIZE *
sizeof(char), NULL, &ret);
/* Create Kernel Program from the
source */
program =
clCreateProgramWithSource(context, 1,
(const char **)&source_str,
(const size_t *)&source_size, &ret);

/* Build Kernel Program */
ret = clBuildProgram(program, 1,
&device_id, NULL, NULL, NULL);
/* Create OpenCL Kernel */
kernel = clCreateKernel(program,
"hello", &ret);
/* Set OpenCL Kernel Parameters */
ret = clSetKernelArg(kernel, 0,
sizeof(cl_mem), (void *)&memobj);
/* Execute OpenCL Kernel */
ret = clEnqueueTask(command_queue,
kernel, 0, NULL,NULL);
/* Copy results from the memory buffer
*/
ret =
clEnqueueReadBuffer(command_queue,
memobj, CL_TRUE, 0,
MEM_SIZE * sizeof(char),string, 0,
NULL, NULL);
```

```c
/* Display Result */

puts(string);

/* Finalization */

ret = clFlush(command_queue);

ret = clFinish(command_queue);

ret = clReleaseKernel(kernel);

ret = clReleaseProgram(program);

ret = clReleaseMemObject(memobj);

ret =
clReleaseCommandQueue(command_queue);

ret = clReleaseContext(context);

free(source_str);

return 0;

}
```