COMP 322: Fundamentals of Parallel Programming

Lecture 38: Comparison of Programming Models

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Acknowledgments

• “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003, and accompanying slides
  — http://www-users.cs.umn.edu/~karypis/parbook/

• Slides from COMP 422 course at Rice University
  — http://www.clear.rice.edu/comp422/


• Slides from OpenMP tutorial given by Ruud van der Paas at HPCC 2007
  — http://www.tlc2.uh.edu/hpcc07/Schedule/OpenMP

• “Towards OpenMP 3.0”, Larry Meadows, HPCC 2007 presentation
  — http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc07_Larry.ppt

• Pthreads: A Brief Introduction, CSCI 8530 lecture, University of Nebraska Omaha

• “Principles of Parallel Programming”, Calvin Lin & Lawrence Snyder
  — Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html

• Tim Warburton, Rice University, “Introduction to GPGPU Programming”
  — 5-day course taught at Danish Technical University (DTU) in May 2011

Parallel Programming is a Cross-Cutting Concern

Developer Pyramid (not drawn to scale!)

Software Stack

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain-Specific langs.</td>
</tr>
<tr>
<td>Middleware</td>
</tr>
<tr>
<td>Programming</td>
</tr>
<tr>
<td>Runtime Systems</td>
</tr>
<tr>
<td>Compilers</td>
</tr>
<tr>
<td>System Libraries</td>
</tr>
<tr>
<td>OS and Hypervisors</td>
</tr>
</tbody>
</table>

Parallel Programming

Application
Developers

Infrastructure
Developers

System Programmers
Different Parallel Programming Models for different Levels of Developer Pyramid and Software Stack

Software Stack

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain-Specific Langs.</td>
</tr>
<tr>
<td>Middleware</td>
</tr>
<tr>
<td>Programming</td>
</tr>
<tr>
<td>Runtime Systems</td>
</tr>
<tr>
<td>Compilers</td>
</tr>
<tr>
<td>System Libraries</td>
</tr>
<tr>
<td>OS and Hypervisors</td>
</tr>
</tbody>
</table>

Application Developers

Infrastructure Developers

System Programmers

LabView
Matlab
Chapel, X10
Habanero-Java
Habanero-C
Java threads
OpenMP
MPI
CUDA
OpenCL
Pthreads

COMP 322, Spring 2012 (V. Sarkar)
Outline

- Pthreads
- OpenMP
- CUDA
POSIX Thread API (Pthreads)

• Standard user threads API supported by most vendors
• Library interface, intended for system programmers
• Concepts behind Pthreads interface are broadly applicable
  — largely independent of the API
  — useful for programming with other thread APIs as well
    — Windows threads
    — Solaris threads
    — Java threads
    — ...
• Threads are peers, unlike Linux/Unix processes
  — no parent/child relationship
PThread Creation

Asynchronously invoke `thread_function` in a new thread

```c
#include <pthread.h>
int pthread_create(
    pthread_t *thread_handle, /* returns handle here */
    const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg); /* single argument; perhaps a structure */
```

Attribute created by `pthread_attr_init`

Contains details about
- whether scheduling policy is inherited or explicit
- scheduling policy, scheduling priority
- stack size, stack guard region size

Can use NULL for `pthread_attr_init` for default values
Pthread Termination

- A thread terminates by calling the function `pthread_exit()`. A single argument, a pointer to a void* object, is supplied as the argument to `pthread_exit`. This value is returned to any thread that has blocked while waiting for this thread to exit.

- Suspend parent thread until child thread terminates

```c
#include <pthread.h>
int pthread_join ( 
    pthread_t thread, /* thread id */
    void **ptr); /* ptr to location for return code a terminating 
                  thread passes to pthread_exit */
```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 32
void *compute_pi (void *);
...
int main(...) {
  ...
  pthread_t p_threads[NUM_THREADS];
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  for (i=0; i< NUM_THREADS; i++) {
    hits[i] = i;
    pthread_create(&p_threads[i], &attr, compute_pi,
                   (void*) &hits[i]);
  }
  for (i=0; i< NUM_THREADS; i++) {
    pthread_join(p_threads[i], NULL);
    total_hits += hits[i];
  }
  ...
}
Example of Implementing a Reduction Using Mutex Locks

```c
pthread_mutex_t cost_lock;
...
int main() {
...
    pthread_mutex_init(&cost_lock, NULL);
...
}
void *find_best(void *list_ptr) {
...
    pthread_mutex_lock(&cost_lock);    /* lock the mutex */
    if (my_cost < best_cost) {
        best_cost = my_cost;
    }
    pthread_mutex_unlock(&cost_lock); /* unlock the mutex */
}
```

use default (normal) lock type

Critical section

COMP 322, Spring 2012 (V. Sarkar)
Pthread’s Condition Variable API

/* initialize or destroy a condition variable */
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

/* block until a condition is true */
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *wtime);

/* signal one or all waiting threads that condition is true */
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

wake one

abort wait if time exceeded

wake all
Condition Variable Producer-Consumer

(main)

```c
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */
main() {
    /* declarations and initializations */
task_available = 0;
pthread_init();
pthread_cond_init(&cond_queue_empty, NULL);
pthread_cond_init(&cond_queue_full, NULL);
pthread_mutex_init(&task_queue_cond_lock, NULL);
/* create and join producer and consumer threads */
}
```
Producer Using Condition Variables

```c
void *producer(void *producer_thread_data) {
    int inserted;
    while (!done()) {
        create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 1)
            pthread_cond_wait(&cond_queue_empty,
                               &task_queue_cond_lock);
        insert_into_queue();
        task_available = 1;
        pthread_cond_signal(&cond_queue_full);
        pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

- Note loop
- Reacquires mutex when woken
- Releases mutex on wait
Consumer Using Condition Variables

```c
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait(&cond_queue_full,
                               &task_queue_cond_lock);
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```
Composite Synchronization Constructs

• Pthreads provides only basic synchronization constructs

• Build higher-level constructs from basic ones e.g., barriers
  — Pthreads extension includes barriers as synchronization objects (available in Single UNIX Specification)
    - Enable by #define _XOPEN_SOURCE 600 at start of file
  — Initialize a barrier for count threads
    - int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrier attr_t *attr, int count);
  — Each thread waits on a barrier by calling
    - int pthread_barrier_wait(pthread_barrier_t *barrier);
  — Destroy a barrier
    - int pthread_barrier_destroy(pthread_barrier_t *barrier);

• Java threads and HJ worker threads are also implemented as pthreads
Summary of key features in Pthreads

<table>
<thead>
<tr>
<th>Pthreads construct</th>
<th>Related HJ/Java constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create()</td>
<td>HJ's async; Java's &quot;new Thread&quot; and &quot;Thread.start()&quot;</td>
</tr>
<tr>
<td>pthread_join()</td>
<td>HJ's finish &amp; future get(); Java's &quot;Thread.join()&quot;</td>
</tr>
<tr>
<td>pthread_mutex_lock()</td>
<td>HJ's begin-isolated, actors; Java's begin-synchronized, and lock() libray calls</td>
</tr>
<tr>
<td>pthread_mutex_unlock()</td>
<td>HJ's end-isolated, actors; Java's begin-synchronized, and lock() libray calls</td>
</tr>
<tr>
<td>pthread_cond_signal()</td>
<td>Deterministic use: HJ's phasers; Nondeterministic use: j.u.c.locks.condition</td>
</tr>
<tr>
<td>pthread_cond_wait()</td>
<td>Deterministic use: HJ's phasers; Nondeterministic use: j.u.c.locks.condition</td>
</tr>
</tbody>
</table>
Outline

• Pthreads
• OpenMP
• CUDA
What is OpenMP?

- Well-established standard for writing shared-memory parallel programs in C, C++ Fortran

- Programming model is expressed via
  - Pragmas/directives (not language extensions)
  - Runtime routines
  - Environment variables

- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)
  - Latest specification: Version 3.0 (May 2008)
  - Previous specification: Version 2.5 (May 2005)
A first OpenMP example

For-loop with independent iterations

```
for (i = 0; i < n; i++)
c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for
  shared(n, a, b, c)\n  private(i)
for (i = 0; i < n; i++)
c[i] = a[i] + b[i];
```

```
% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 4
% a.out
```
The OpenMP Execution Model

Fork and Join Model

Master Thread

Parallel region

Worker Threads

Synchronization

Parallel region

Worker Threads

Synchronization
Terminology

- **OpenMP Team** := Master + Workers

- A **Parallel Region** is a block of code executed by all threads simultaneously
  - The master thread always has thread ID 0
  - Thread adjustment (if enabled) is only done before entering a parallel region
  - Parallel regions can be nested, but support for this is implementation dependent
  - An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially

- A **work-sharing construct** divides the execution of the enclosed code region among the members of the team; in other words: they split the work
Parallel Region

```
#pragma omp parallel [clause[[], clause] ...] 
{
    "this is executed in parallel"
} (implied barrier)
```

A parallel region is a block of code executed by multiple threads simultaneously, and supports the following clauses:

- **if** *(scalar expression)*
- **private** *(list)*
- **shared** *(list)*
- **default** *(nonelshared)* *(C/C++)
- **default** *(nonelshared/private)* *(Fortran)*
- **reduction** *(operator: list)*
- **copyin** *(list)*
- **firstprivate** *(list)*
- **num_threads** *(scalar_int_expr)*
Work-sharing constructs in a Parallel Region

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless nowait is specified)
- A work-sharing construct does not launch any new threads

```c
#pragma omp parallel
#pragma omp for
for (...) { .... }
```
Legality constraints for work-sharing constructs

- Each worksharing region must be encountered by all threads in a team or by none at all.
- The sequence of worksharing regions and barrier regions encountered must be the same for every thread in a team.

```c
#pragma omp parallel
{
  do {
    // c1 and c2 may depend on the OpenMP thread-id
    boolean c1 = ... ; boolean c2 = ... ;
    ...
    if (c2) {
      // Start of work-sharing region with no wait clause
      #pragma omp ...
      . . . // Worksharing statement
    } // if (c2)
  } while (! c1);
}
```

==> No OpenMP implementation checks for conformance with this rule
Example of work-sharing “omp for” loop

```c
#pragma omp parallel default(none)\ 
    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];

} /*-- End of parallel region --*/

(implicit barrier)
```
Reduction Clause in OpenMP

• The reduction clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.

• The syntax of the reduction clause is as follows
  — reduction (operator: variable list).

• The variables in the list are implicitly specified as being private to threads.

• The operator can be one of +, *, -, &, |, ^, &&, and ||.

```c
#pragma omp parallel reduction(+: sum) num_threads(8) {
    /* compute local instances of sum here */
}
/*sum here contains sum of all local instances of sum */
```
“single” and “master” constructs in a parallel region

Only one thread in the team executes the code enclosed

```c
#pragma omp single [clause[, clause] ...]
{
    <code-block>
}
```

Only the master thread executes the code block

```c
#pragma omp master
{
    <code-block>
}
```

- Single and master are useful for computations that are intended for single-processor execution e.g., I/O and initializations
- There is no implied barrier on entry or exit of a single or master construct
task Construct

```c
#pragma omp task [clause[,],clause] ... ]
structured-block

where clause can be one of:

if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none )
```
Example – parallel pointer chasing using tasks

1. `#pragma omp parallel`  
2. `{  
3.   `#pragma omp single private(p)`  
4.     {  
5.       `p = listhead ;`  
6.       `while (p) {`  
7.       `#pragma omp task`  
8.       `process (p);`  
9.       `p = p->next ;`  
10.     }  
11. }  
12.}`

---

 spawned call to `process(p)`

implicit finish at end of parallel region
Example – parallel pointer chasing on multiple lists using tasks (nested parallelism)

1. #pragma omp parallel
2. {
3.   #pragma omp for private(p)
4.   for (int i = 0; i < numlists; i++) {
5.     p = listheads[i];
6.     while (p) {
7.       #pragma omp task
8.       process(p)
9.       p = next(p);
10.     }
11.   }
12. }
Example: postorder tree traversal

```c
void postorder(node *p) {
    if (p->left)
        #pragma omp task
        postorder(p->left);
    if (p->right)
        #pragma omp task
        postorder(p->right);
    #pragma omp taskwait // wait for child tasks
    process(p->data);
}
```

- Parent task suspended until children tasks complete
# Summary of key features in OpenMP

<table>
<thead>
<tr>
<th>OpenMP construct</th>
<th>Related HJ/Java constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel region</td>
<td>HJ forall (forall iteration = OpenMP thread)</td>
</tr>
<tr>
<td>#pragma omp parallel</td>
<td></td>
</tr>
<tr>
<td>Work-sharing constructs:</td>
<td>No direct analogy in HJ or Java</td>
</tr>
<tr>
<td>parallel loops, parallel sections</td>
<td></td>
</tr>
<tr>
<td>Barrier</td>
<td>HJ forall-next on implicit phaser</td>
</tr>
<tr>
<td>#pragma omp barrier</td>
<td></td>
</tr>
<tr>
<td>Single</td>
<td>HJ's forall-next-single on implicit phaser (but HJ does not support single + nowait)</td>
</tr>
<tr>
<td>#pragma omp single</td>
<td></td>
</tr>
<tr>
<td>Reduction clauses</td>
<td>HJ's finish accumulators (in forall)</td>
</tr>
<tr>
<td>Critical section</td>
<td>HJ's isolated statement</td>
</tr>
<tr>
<td>#pragma omp critical</td>
<td></td>
</tr>
<tr>
<td>Task creation</td>
<td>HJ's async statement</td>
</tr>
<tr>
<td>#pragma omp task</td>
<td></td>
</tr>
<tr>
<td>Task termination</td>
<td>HJ's finish statement</td>
</tr>
<tr>
<td>#pragma omp taskwait</td>
<td></td>
</tr>
</tbody>
</table>
Outline

- Pthreads
- OpenMP
- CUDA
CPUs and GPUs have fundamentally different design philosophies

**GPU = Graphics Processing Unit**

**Single CPU core**
- Control
- ALU
- ALU
- Cache
- DRAM

**Multiple GPU processors**
- Control
- ALU
- ALU
- Streaming Multiprocessor
- DRAM

GPUs are provided to accelerate graphics, but they can also be used for non-graphics applications that exhibit large amounts of data parallelism and require large amounts of “streaming” throughput.
Process Flow of a CUDA Kernel Call (Compute Unified Device Architecture)

- Data parallel programming architecture from NVIDIA
  - Execute programmer-defined kernels on extremely parallel GPUs
  - CUDA program flow:
    1. Push data on device
    2. Launch kernel
    3. Execute kernel and memory accesses in parallel
    4. Pull data off device
- Device threads are launched in batches
  - Blocks of Threads, Grid of Blocks
- Explicit device memory management
  - cudaMalloc, cudaMemcpy, cudaFree, etc.

Figure source: Y. Yan et. al “JCUDA: a Programmer Friendly Interface for Accelerating Java Programs with CUDA.” Euro-Par 2009.
Execution of a CUDA program

- **Integrated host+device application**
  - Serial or modestly parallel parts on CPU host
  - Highly parallel kernels on GPU device

  [Diagram showing host and device code with serial and parallel parts]

  **Host Code**
  (small number of threads)

  **Device Kernel**
  (large number of threads)

  [Diagram showing host and device code with serial and parallel parts]

  **Host Code**
  (small number of threads)

  **Device Kernel**
  (large number of threads)

  [Diagram showing host and device code with serial and parallel parts]
Matrix multiplication kernel code in CUDA (SPMD model with index = threadIdx)

```c
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k) {
        float Mdelement = Md[ty * Width + k];
        float Ndelement = Nd[k * Width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```
Host Code in C for Matrix Multiplication

1. void MatrixMultiplication(float* M, float* N, float* P, int Width) {
2.   int size = Width*Width*sizeof(float); // matrix size
3.   float* Md, Nd, Pd; // pointers to device arrays
4.   cudaMalloc((void**)&Md, size); // allocate Md on device
5.   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice); // copy M to Md
6.   cudaMalloc((void**)&Nd, size); // allocate Nd on device
7.   cudaMemcpy(Nd, M, size, cudaMemcpyHostToDevice); // copy N to Nd
8.   cudaMalloc((void**)&Pd, size); // allocate Pd on device
9.   dim3 dimBlock(Width, Width); dim3 dimGrid(1, 1);
10.  // launch kernel (equivalent to “async at(GPU), forall, forall”
11.   MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
12.  cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost); // copy Pd to P
13.  // Free device matrices
14.   cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
15. }

38

COMP 322, Spring 2012 (V. Sarkar)
Organization of a CUDA grid (Figure 4)
CUDA Host-Device Data Transfer

- cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
- copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- The memory areas may not overlap
- Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior.
CUDA Storage Classes

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory

- Host code can:
  - Transfer data to/from per-grid global and constant memories
CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>local</strong> int LocalVar;</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__, __shared__, or __constant__

- **Automatic variables** without any qualifier reside in a *register*
  - Except arrays that reside in local memory

- **Pointers** can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    __global__ void KernelFunc(float* ptr)
  - Obtained as the address of a global variable:
    float* ptr = &GlobalVar;
CUDA Storage Classes

- **Local Memory**: per-thread
  - Private per thread
  - Auto variables, register spill
- **Shared Memory**: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- **Global Memory**: per-application
  - Shared by all threads
  - Inter-Grid communication
## Summary of key features in CUDA

<table>
<thead>
<tr>
<th>CUDA construct</th>
<th>Related HJ/Java constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel invocation, &lt;&lt;&lt;...&gt;&gt;&gt;</td>
<td>async at(gpu-place)</td>
</tr>
<tr>
<td>1D/2D grid with 1D/2D/3D blocks of threads</td>
<td>Outer 1D/2D forall with inner 1D/2D/3D forall</td>
</tr>
<tr>
<td>Intra-block barrier, __syncthreads()</td>
<td>HJ forall-next on implicit phaser for inner forall</td>
</tr>
<tr>
<td>cudaMemcpy()</td>
<td>No direct equivalent in HJ/Java (can use System.arraycopy() if needed)</td>
</tr>
<tr>
<td>Storage classes: local, shared, global</td>
<td>No direct equivalent in HJ/Java (method-local variables are scalars)</td>
</tr>
</tbody>
</table>