CSCI-580 Advanced High Performance Computing

Lecture 3: Parallel Programming Models

Bo Wu
Colorado School of Mines

Part of slide resources from: Katherine Yelick (UC Berkeley), Cris Cecka (Harvard), Scott Baden (UCSD) and Andreas Moshovos (UOfToronto)
Consider two processors that write to different locations mapping to different parts of the same cache block.
False Sharing

- P0 writes a location
- Assuming we have a write-through cache memory is updated
False Sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory
False Sharing

- P1’s write updates main memory
- Cache coherence protocol invalidates the corresponding block in P0’s cache
Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache.
Data Race

- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
- The accesses are concurrent (not synchronized) so they could happen simultaneously

```
shared int s = 0;
Core 1
s += a[0];
Core 2
s += a[1];
```

Is the result $s = a[0] + a[1]$?
shared int s = 0;

Core 1
load s to r1
load a[0] to r0
r2 = r0 + r1
store r2 to s

Core 2
load s to r1
load a[1] to r0
r2 = r0 + r1
store r2 to s

The result is correct if?
Parallel programing with threads
Shared Memory Programming

- PTHREADS is the POSIX Standard
  - POSIX: Portable Operating System Interface for UNIX
  - Relatively low level
  - Portable but possibly slow
    - https://computing.llnl.gov/tutorials/pthreads/

- OpenMP is relatively new
  - Support for scientific programming on shared memory
  - De facto standard for SMP scientific codes
    - http://www.openMP.org
OpenMP: an example

for (i=0; i<N; i++)
  C[i] = A[i] + B[i];

#pragma omp parallel for
for (i=0; i<N; i++)
  C[i] = A[i] + B[i];

Compilation: gcc -o VecAdd -fopenmp VecAdd.c
Execution: ./VecAdd
Discussion on Forking Threads

omp_set_num_threads(8);
#pragma omp parallel for
for (i=0; i<N; i++)
    C[i] = A[i] + B[i];

......
#pragma omp parallel for
for (i=0; i<N; i++)
    F[i] = D[i] + E[i];
A Programmer’s View of OpenMP

- **OpenMP will:**
  - Allow a programmer to separate a program into serial regions and parallel regions
  - Hide stack management
  - Provide synchronization constructs

- **OpenMP will not:**
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races
Programming distributed memory machines with message passing
Programming With MPI

- MPI is a library
  - MPI: Message Passing Interface
  - Several implementations: MPICH, Open MPI
  - All operations are performed with routine calls
  - Basic definitions in
    - mpi.h for C
    - mpif.h for Fortran 77 and 90
MPI: Hello World

```c
/*
 * "Hello World" MPI Test Program
 */
#include <mpi.h>
#include <stdio.h>
#include <string.h>
#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[])
{
    char idstr[32];
    char buff[BUFSIZE];
    int numprocs;
    int myid;
    int i;
    MPI_Status stat;
    /* MPI programs start with MPI_Init; all 'N' processes exist thereafter */
    MPI_Init(&argc, &argv);
    /* find out how big the SMPD world is */
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    /* and this process's rank is */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    /* At this point, all programs are running equivalently, the rank distinguishes the roles of the programs in the SMPD model, with rank 0 often used specially... */
    if(myid == 0)
    {
        printf("Hello World!
");
        MPI_Send(buff, BUFSIZE, MPI_CHAR, 1, TAG, MPI_COMM_WORLD);
    }
    for(i=1;i<numprocs;i++)
    {
        MPI_Recv(buff, BUFSIZE, MPI_CHAR, 1, TAG, MPI_COMM_WORLD, &stat);
        printf("Hello World!
");
        MPI_Send(buff, BUFSIZE, MPI_CHAR, 1, TAG, MPI_COMM_WORLD, &stat);
    }
    /* receive from rank 0; */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    printf(idstr, "Processor ", myid);
    strcat(idstr, " reporting for duty ", BUFSIZE-1);
    MPI_Send(buff, BUFSIZE-1, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
    /* MPI programs end with MPI_Finalize; this is a weak synchronization point */
    MPI_Finalize();
    return 0;
}
```

example from Wiki
MPI: Hello World

```c
/*
"Hello World" MPI Test Program
*/

#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[]) {
    char idstr[32];
    char buff[BUFSIZE];
    int numprocs;
    int myid;
    int i;
    MPI_Status stat;
    /* MPI programs start with MPI_Init; all 'N' processes exist thereafter */
    MPI_Init(&argc,&argv);
    /* find out how big the SPMD world is */
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    /* and this processes' rank is */
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```
MPI: Hello World

```c
/* At this point, all programs are running equivalently, the rank
distinguishes the roles of the programs in the SPMD model, with
rank 0 often used specially... */
if (myid == 0)
{
  printf("%d: We have %d processors\n", myid, numprocs);
  for(i=1;i<numprocs;i++)
  {
    sprintf(buff, "Hello %d!", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
  }
  for(i=1;i<numprocs;i++)
  {
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
    printf("%d: %s\n", myid, buff);
  }
}
else
{
  /* receive from rank 0: */
  MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
  sprintf(idstr, "Processor %d", myid);
  strcat(buff, idstr, BUFSIZE-1);
  strcat(buff, "reporting for duty", BUFSIZE-1);
  /* send to rank 0: */
  MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
}

/* MPI programs end with MPI Finalize; this is a weak synchronization point */
MPI_Finalize();
return 0;
```
MPI: Hello World

```c
/*
 * 'Hello World' MPI Test Program
 */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[])
{
    char idstr[32];
    char buf[BUFSIZE];
    int numprocs;
    int myid;
    int i;
    MPI_Status status;
    /* MPI programs start with MPI_Init; all 'N' processes exist thereafter */
    MPI_Init(&argc, &argv);
    /* find out how big the SHMD world is */
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    /* and this process's rank is */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    /* At this point, all programs are running equivalently, the rank 
       distinguishes the roles of the programs in the SHMD model, with 
       rank 0 often used specially... */
    if (myid == 0) {
        printf("Hello World\n");
        for (i = 1; i < numprocs; i++) {
            printf("Hello %d\n", i);
            MPI_Send(buf, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
        }
    }
    else {
        /* receive from rank 0; */
        MPI_Recv(buf, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &status);
        printf(idstr, Processor %d, myid);
        streambuf(buf, idstr, BUFSIZE-1);
        streambuf(buf, "reporting for duty", BUFSIZE-1);
        /* send to rank 0 */
        MPI_Send(buf, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
    }
    /* MPI programs end with MPI_Finalize; this is a weak synchronisation point */
    MPI_Finalize();
    return 0;
}
```

Compilation: mpicc -o hello hello.c
Execution: mpirun -np 4 hello
 MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - MPI_INIT
  - MPI_FINALIZE
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
  - MPI_SEND
  - MPI_RECV
Send andRecv

○ Point-to-point communication

○ When Send( ) returns, the message is “in transit”
  – A return doesn’t tell us if the message has been received
  – Somewhere in the system

○ Receive( ) blocks until the message has been received
Buffering

- Where does the data go when you send it? One possibility is:

  ![Diagram showing data flow through local buffers and the network between processes 0 and 1]
Causality

- If a process sends multiple messages to the same destination, then the messages will be received in the order sent.
No guarantee with different senders

- If different processes send messages to the same destination
  - The order of receipt is defined from a single source
  - The order of receipt is not defined across multiple sources
Learn More about MPI

○ The Standard itself:
  – at http://www.mpi-forum.org
  – All MPI official releases, in both postscript and HTML

○ Other information on Web:
  – at http://www.mcs.anl.gov/mpi
  – pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

○ The reference book from Norm Matloff has a nice introduction of MPI
Computing with Graphics Processing Units (GPUs)
Design Philosophy of GPU

- Branch prediction
- Out-of-order execution
- Speculative execution

None
NVIDIA Kepler GPU
GPU v.s. CPU (throughput)

We have two!

We have four!

from NVIDIA
GPU v.s. CPU (memory bandwidth)

Theoretical GB/s

- CPU
- GeForce GPU
- Tesla GPU

from NVIDIA
How to Get High Performance

- How to keep the compute engines busy?
  - Needs a lot of independent calculations
Some Things are Naturally Parallel
Sequential Execution Model

```plaintext
int a[N]; // N is large
for (i = 0; i < N; i++)
    a[i] = a[i] * fade;
```

Flow of control / Thread
One instruction at the time
Optimizations possible at the machine level
Data Parallel Model

```c
int a[N]; // N is large
for all elements do in parallel
    a[index] = a[index] * fade;
```
int a[N]; // N is large
for all elements do in parallel
    if (a[i] > threshold) a[i] *= fade;

The model used in today’s Graphics Processors
Programmer’s View

- GPU as a co-processor

CPU

GPU

Memory

6.4GB/sec – 31.92GB/sec
8B per transfer

3GB/s – 8GB/s

141GB/sec

GPU Memory
Target Applications

```c
int a[N]; // N is large
for all elements of a compute
    a[i] = a[i] * fade
```

- Lots of independent computations
  - GPU threads need not be independent
Programmer’s View of the GPU

○ GPU: a compute device that:
  – Is a coprocessor to the CPU or host
  – Has its own DRAM (device memory)
  – Runs many threads in parallel

○ Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
CPU threads vs. GPU threads

- GPU threads are extremely lightweight
  - Very little creation overhead
  - In the order of microseconds
  - All done in hardware

- GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few
Execution Timeflow

1. Copy to GPU mem
2. Launch GPU Kernel
2'. Synchronize with GPU
3. Copy from GPU mem
Programmer’s View

- Prepare data in CPU memory
Programmer’s View

- Copy data to GPU
Programmer’s View

- GPU starts a computation -> runs a kernel
- CPU can also continue
Programmer’s View

- CPU and GPU synchronize
Programmer’s View

- Copy results back to CPU
CUDA exposes the hardware to the programmer
Programmer must manually partition the work appropriately
Think of data as an array
Computation Partitioning — Kernel

○ Computation Grid: 2D case

○ Threads in the same block can communicate
  – Run on the same multiprocessor

○ Threads across blocks can not communicate
  – Should not touch each other’s data
  – Behavior undefined
Grids of blocks of threads

Host

Kernel 1

Device

Grid 1

Block (0, 0)
Block (0, 1)
Block (0, 2)

Block (1, 0)
Block (1, 1)
Block (1, 2)

Grid 2

Execution

Kernel 2

Block (1, 1)

Thread (0, 0)  Thread (0, 1)  Thread (0, 2)  Thread (0, 3)
Thread (1, 0)  Thread (1, 1)  Thread (1, 2)  Thread (1, 3)
## Block and Thread IDs

- **Threads and blocks have IDs**
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D or 3D
  - Thread ID: 1D or 2D or 3D

- **IDs are accessible through predefined variables**
  - blockDim.x
  - threadIdx.y

---

### Diagram

![Diagram showing block and thread IDs]

<table>
<thead>
<tr>
<th>Grid 1</th>
<th>Block (0, 0)</th>
<th>Block (1, 0)</th>
<th>Block (2, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block (0, 1)</td>
<td>Block (1, 1)</td>
<td>Block (2, 1)</td>
<td></td>
</tr>
<tr>
<td>Thread (0, 0)</td>
<td>Thread (1, 0)</td>
<td>Thread (2, 0)</td>
<td>Thread (3, 0)</td>
</tr>
<tr>
<td>Thread (0, 1)</td>
<td>Thread (1, 1)</td>
<td>Thread (2, 1)</td>
<td>Thread (3, 1)</td>
</tr>
<tr>
<td>Thread (0, 2)</td>
<td>Thread (1, 2)</td>
<td>Thread (2, 2)</td>
<td>Thread (3, 2)</td>
</tr>
</tbody>
</table>
Execution Model: Ordering

- Execution order of threads is undefined
- Do not assume and use:
  - Block 0 executes before block 1
  - Thread 10 executes after thread 2
  - And any other ordering
- Why?
  - More flexible hardware options
Execution Model Summary

- Grid of blocks of threads
  - 1D/2D/3D grid of blocks
  - 1D/2D/3D blocks of threads
- All blocks are identical:
  - same structure and # of threads
- Block execution order is undefined
- Same block threads:
  - can synchronize and share data fast (shared memory)
- Threads from different blocks:
  - Cannot cooperate
  - Communication through global memory
- Threads and Blocks have IDs
  - Simplifies data indexing
  - Can be 1D, 2D, or 3D (threads)