Creative Commons License

This work is under a Creative Commons Attribution 4.0 Unported License

The details of this license are publicly available at https://creativecommons.org/licenses/by-nc-nd/4.0
Table of Contents

• Introduction
• Parallel Programming Models
  • Let’s code!!!
• Parallelism: different Points-of-view
• Tradeoffs
• Approaches
  • Inter-Process Communication
Introduction to Parallelism

• Parallelism is the capability to perform several actions at the same time

• Parallel computing: multiple computations are executed at the same time...
  • ...but what sort of computations? ... and what about communications?

• Summary of the “levels of parallelism”
  • Bit-Level: the finest grain parallelism (e.g. one 64-bit vs two 32-bit calculations)
  • Instruction-Level: multiple instructions per cycle (e.g. 1 ALU and 1 FPU instructions at a time)
  • Data-Level: same computation to multiple data (e.g. 3D-graphics processing, vectors, simd)
  • Task-Level: different computations to the same or different datasets (e.g. complex applications)

• This lesson focuses on Data and Task levels of parallelism
Generic Use Cases of Parallel Programming

- **Parallelism exploitation**
  - Workload can be split into multiple workload subsets that can be processed in parallel
    - with no or few dependences
  - Multiple independent inputs can be processed in parallel
    - with no or few dependences

- **Task encapsulation**
  - Task specific modules focused on a part of an application, to improve performance
    - e.g. AI and physics engines in videogames

- **I/O efficiency**
  - Decouple I/O management from the main execution workload

- **Service request pipelining (sustain required Quality-of-Service)**
  - Enhance distribution of pipeline processing stages over hardware resources towards improve performance
Sequential vs Parallel Execution

• **Sequential Execution**: code can only be executed by a single Sw thread. It is independent of Hw resource availability.

![Sequential Execution Diagram]

• **Parallel Execution**: multiple Sw threads can execute code at the same time. The code may show occasional dependences or synchronization barriers.

![Parallel Execution Diagram]
Parallelism vs Concurrency

• **Parallelism**: N Sw threads run at a given time in N Hw threads
  
  Time (each CPU executes one process)

| CPU0 | Proc. 0 | Proc. 1 | Proc. 2 | Proc. 0 | Proc. 1 | Proc. 2 | Proc. 0 | Proc. 1 | Proc. 2 | ...
|------|---------|---------|---------|---------|---------|---------|---------|---------|---------|--------|

• **Concurrency**: N Sw threads could be potentially executed in parallel, but there are not enough Hw resources to do it
  
  • The OS selects what Sw thread can run and what Sw thread has to wait

Time (CPU is shared among processes)
Table of Contents

• Introduction
• Parallel Programming Models
  • Let’s code!!!
• Parallelism: different Points-of-view
• Tradeoffs
• Approaches
  • Inter-Process Communication
Fork-Join Model

- Parallel programs show code snippets that can be parallelized
  - **Fork**: A piece of code branches off to be executed by multiple threads
  - **Join**: At the end of the parallel code, threads are synchronized to resume sequential execution

- Master thread spawns multiple worker threads in parallel regions
  - Master thread participates as a worker
Parallel Programming Models

• This lesson focuses on
  • **OpenMP** (Open specification for Multi Processing)
    • Standard API to write shared memory parallel programs (C, C++ and Fortran)
    • [http://www.openmp.org](http://www.openmp.org)
  • Pthreads (POSIX Threads)
    • POSIX (Portable Operating System Interface) API and parallel execution model
    • Supported by multiple OSs

• Spawn multiple threads that share memory (access to all data)
Difficulties of Sharing Data

• The correct behavior of the program MUST NOT depend on the sequence of processes or threads execution (a.k.a. race condition)
  • It is out of our control

• Shared data MUST BE properly protected to guarantee data consistency
  • Critical sections (code that accesses to shared data) have to be correctly protected and accessed
  • Does a given critical section really need to be protected???

• We have to avoid deadlocks: two or more threads wait for each other forever

• Over protection of parallel code degrades performance
  • We have protected code that is not necessary

• It is difficult to totally guarantee the correct behavior of multithreaded codes
  • Possible collisions among threads depend on the sequence of processes/threads execution
    • E.g. Debug deterministic sequences
The Big Issue: Sharing Data

- Threads can potentially access to any shared data
- Critical Section
  - Code that accesses to shared data

```c
Func_assign(int value){
    ...
    pos = pos_free;
    vector[pos] = value;
    pos_free++;
    ...
}

Th0

Func_assign(10){
    ...
    pos = pos_free;
    vector[pos] = 10;
    pos_free++;
    ...
}

CRITICAL SECTION

Th1

Func_assign(20){
    ...
    pos = pos_free;
    vector[pos] = 20;
    pos_free++;
    ...
}
The Big Issue: Sharing Data

- We cannot control when multiple threads may collide in a shared data
- Shared data MUST BE properly protected to guarantee data consistency

```
//at this moment: “pos_free == 2”

Func_assign(int value){
  ...
  pos = pos_free;
  vector[pos] = value;
  pos_free++;
  ...
}

Th0

Func_assign(10){
  ...
  pos = pos_free;//2?? 3???
  vector[pos] = 10;
  pos_free++; //3?? 4???
  ...
}

Th1

Func_assign(20){
  ...
  pos = pos_free;//2?? 3???
  vector[pos] = 20;
  pos_free++; //3?? 4???
  ...
}

// at this moment we cannot guarantee the values of “pos_free”, “vector[2]”, and “vector[3]”
```
Synchronization Mechanisms

• **Mutex** (MUTual EXclusion)
  - The running thread waits till the lock (mutex) is available to get its ownership

• **Barriers**
  - All threads wait at a given point, till the last one arrives

• **Semaphores**
  - A common resource can be accessed by multiple threads

• **Spinlocks**
  - **Active** checking for a lock

• **Hardware support**
  - Atomically read and modify a memory location
Synchronization: Mutual Exclusion

• Only a single thread can be in a critical section at a time

• Utilization
  
  **Directive [lock name]**

• Limitations
  
  • It is not exception safe
  
  • It is not allowed to break out
  
  • All non-named criticals wait for each other
    
    • Lock_name helps

• Alternatives
  
  • Lock based alternatives

```
Time

Th0
Func_assign(10) {
  ...
  Directive
  {
    pos = pos_free;
    vector[pos] = 10;
    pos_free++;
  }
  ...
}

Th1
Func_assign(20) {
  ...
  Directive
  {
    pos = pos_free;
    vector[pos] = 10;
    pos_free++;
  }
  ...
}

Thread is waiting
```
Synchronization: Barriers

• All threads wait for the remaining threads to reach the checkpoint
• There are implicit barriers at the end of parallel regions

• Utilization

```c
{
    ...
    Barrier Directive
    ...
}
```
Table of Contents

• Introduction
• Parallel Programming Models
  • Let’s code!!!
• Parallelism: different Points-of-view
• Tradeoffs
• Approaches
  • Inter-Process Communication
OpenMP

- OpenMP: open source API that supports shared memory multiprocessing programming
  - From OpenMP-1 (end of ’90s) to version 5.0 (released on 2018)

- Directives
  - Additional information for the compiler: how to process directive annotated code
    - `#pragma omp` directive [clause [ clause ...] ]
  - Clause example: data scoping
    - Private: each thread has its own copy of the variable
    - Shared: threads share a single copy of the specified variable
    - ...

- Format of Runtime Library Routines & Environment Variables
  - `omp_...` & `OMP_...`
OpenMP Components
OpenMP key extensions

- Parallel control structures
  - governs flow of control in the program
- Work sharing
  - distributes work among threads
- Data environment
  - scopes variables
- Synchronization
  - coordinates thread execution
- Runtime functions, environment variables
  - runtime environment

OpenMP language extensions
Example: Hello World

```c
int main() {
    ... 
    sprintf(buf, "Hello!\n");
    write(1, buf, strlen(buf));
    ... 
}
```

Output
Hello!
Parallel Control Structures

• GCC flag: "-fopenmp" → Enables handling of OpenMP directives
  • gcc -fopenmp -o hello hello.c

```c
#include <omp.h>
int main() {
    ...
    #pragma omp parallel
    {
        char buf[200];
        sprintf(buf, "Hello!\n");
        write(1, buf, strlen(buf));
    }
    ...
}
```

Directive

Parallel Region
buf is explicitly privatized
# Runtime Environment

- Environment variables
  - OMP_NUM_THREADS
    - E.g.: export OMP_NUM_THREADS=4

```c
#include <omp.h>
int main() {
  ...
  #pragma omp parallel
  {
    char buf[200];
    sprintf(buf, "Hello!\n");
    write(1, buf, strlen(buf));
  }
  ...
}
```

Output
Hello!
Hello!
Hello!
Hello!
Runtime Functions

• Runtime Library Routines
  • `omp_get_thread_num()`
    • Returns the ID of the current thread

```c
#include <omp.h>
int main() {
  ...
#pragma omp parallel num_threads(4)
  {
    int thid = omp_get_thread_num();
    char buf[300];
    sprintf(buf, "Hello %d!\n", thid);
    write(1, buf, strlen(buf));

  }
  ...
}
```

Output
Hello 0!
Hello 3!
Hello 1!
Hello 2!
Work Sharing

• Parallelizing regions (e.g. for, task, sections,...)

```c
#include <omp.h>

int main() {
    ...
    #pragma omp parallel
    {
        #pragma omp single
        {
            int thid = omp_get_thread_num();
            numthreads = omp_get_num_threads();
        }
        #pragma omp for
        for (i=0; i<10; i++) {
            sprintf(buf, "Hello i %d th %d\n", i, thid);
            write(1, buf, strlen(buf));
        }
    }
    ...
}
```

Output
Hello i 0 th 0!
Hello i 1 th 0!
Hello i 2 th 1!
Hello i 3 th 1!
Hello i 4 th 1!
Hello i 5 th 2!
Hello i 6 th 2!
Hello i 7 th 2!
Hello i 8 th 3!
Hello i 9 th 3!
Data Environment

• **private**: Each thread has its own copy of the data
  • Other threads cannot access this data
  • Changes are only visible to the thread owning the data
  • By default, the loop iteration counters are private

• **firstprivate**: private initialized with original value

• **shared**: Threads share a single copy of the data
  • Other threads can access this data
  • Threads can read and write the data *simultaneously*
  • By default, all variables in a work sharing region are shared except the loop iterator

• **default (shared|none)**: explicitly determines the default data sharing

• The default data scoping is … depends 😊

• Utilization
  • `#pragma omp parallel shared(x,y) private(thid)`
Synchronization Mechanisms

- **Mutex (MUTual EXclusion)**
  - The running thread waits till the lock (mutex) is available to get its ownership
- **Barriers**
  - A thread waits for other threads to reach a given point
- **Semaphores**
  - A common resource can be accessed by multiple threads
- **Spinlocks**
  - Active checking for a lock
- **Hardware support**
  - Atomically read and modify a memory location
Some Synchronization Clauses

- **Critical**
  - Restricts execution of the associated structured block to a single thread at a time
  - \#pragma omp critical [(name) [hint (hint-expression)]]
  - structured-block

- **Barrier**
  - Specifies an explicit barrier at the point at which the construct appears
  - \#pragma omp barrier

- **Taskwait**
  - Specifies a wait on the completion of child tasks of the current task
  - \#pragma omp taskwait

- **Atomic**
  - Ensures that a specific storage location is accessed atomically
  - \#pragma omp atomic
  - expression
Synchronization: Mutual Exclusion

- Only a single thread can be in a critical section at a time

**Utilization**

`Directive [lock name]`

**Limitations**

- It is not exception safe
- It is not allowed to break out
- All non-named criticals wait for each other
  - Lock_name helps

**Alternatives**

- Lock based alternatives

```
Th0
Func_assign(10) {
  ... Directive
  pos = pos_free;
  vector[pos] = 10;
  pos_free++;
}
...
```

```
Th1
Func_assign(20) {
  ... Directive
  pos = pos_free;
  vector[pos] = 10;
  pos_free++;
}
...
```
Synchronization: Mutual Exclusion

• Only a single thread can be in a critical section at a time

• Utilization
  Directive [lock name]

• Limitations
  • It is not exception safe
  • It is not allowed to break out
  • All non-named criticals wait for each other
    • Lock_name helps

• Alternatives
  • Lock based alternatives

```
Th0
Func_assign(10) {
  ...
  #pragma omp critical
  {
    pos = pos_free;
    vector[pos] = 10;
    pos_free++;
  }
  ...
}

Th1
Func_assign(20) {
  ...
  #pragma omp critical
  {
    pos = pos_free;
    vector[pos] = 10;
    pos_free++;
  }
  ...
}
Thread is waiting
```
Table of Contents

• Introduction
• Parallel Programming Models
  • Let’s code!!!
• Parallelism: different Points-of-view
• Tradeoffs
• Approaches
Parallelism: Hardware point of view

• Number of instructions executed per cycle
  • ILP vs TLP

• Resource limitations
  • E.g.: #processors, #cores, #HwThreads
  • Structural hazards
    • A planned instruction cannot be executed because the resource is still being used by others

• Flynn’s taxonomy
  • [Link to Flynn's taxonomy](https://en.wikipedia.org/wiki/Flynn's_taxonomy)
Parallelism: Software point of view

• Code execution is inherently limited by dependences
  • Data dependency
    • A given value of the current instruction depends on the outcome of another instruction
  • Control dependency
    • The execution flow path depends on the outcome of a given instruction

• Compiler / Library support
  • Optimizations at compile-time
  • Parallel programming models
Parallelism: Software point of view

• Degrees of Parallelism
  • Coarse-grained parallelism (thousands of instructions or independent programs; ~seconds)
    • Parts of a program, Tasks
    • More difficult detection, less communication requirements
    • Low communication and synchronization overhead, but difficult for load balancing
  • Medium-grained parallelism (<10000 instructions; ~milliseconds)
    • Procedures, Routines
  • Fine-grained parallelism (10s-1000s instructions; ~microseconds)
    • Loops, Groups of instructions
    • Assisted by compiler (difficult to detect by programmers) and parallel programming models
    • Suitable for shared memory approaches and easy for load balancing

• Developer design decisions may have important consequences!!!!
Parallelism: Operating System point of view

• Process and thread management
  • User-Level Threads vs Kernel-Level Threads → Important impact on scheduling

• Scheduling policies
  • Affinity based, priority based

• Load balancing across hardware resources
  • Dealing with dependences and shared-resource contention

• Communication and synchronization capabilities
  • Inter-process communication (IPC)
    • Addressed in this lesson
Tradeoffs of Parallelism

• Shared-Memory vs Distributed-Memory Computing
  • Shared-memory
    • suitable for threaded programs
  • Distributed-memory
    • suitable for computational demanding workloads

• Hybrid programming techniques
  • combine the advantages of both of them
Tradeoffs of Parallelism

• Processes vs Threads
  • Multi-processing:
    • a global problem is split into several processes that can run on shared or distributed memory devices. Every process has access to its own memory space
    • E.g.: MPI – Message Passing Interface
  • Multi-threading:
    • a code snippet is split into several flows that can run on a shared memory device. Every thread has access to all data
    • E.g.: OpenMP – Open MultiProcessing
Utilization of resources (Processes)

- Multi-process approach
  - No memory sharing in this case, no complexity on the memory management
Utilization of resources (Threads)

• Multi-thread approach
  • Allow to exploit parallelism inside each process
  • Require thread management support
    • From programming language libraries and OS
Table of Contents

• Introduction
• Parallel Programming Models
  • Let’s code!!
• Parallelism: different Points-of-view
• Tradeoffs
• Approaches
Inter-Process Communication (IPC)

• Cooperating processes/threads need communication and synchronization
  • Data Exchange / Event Notification
  • Blocking (synchronous) / Non-blocking (asynchronous) communication
  • Direct / Indirect communication

• Processes can run on a single host or on several remote hosts

• IPC Method selection based on...
  • Latency, bandwidth, type of data exchanged

• Bugs really difficult to find and solve!!!!
Inter-Process Communication (IPC)

• Communication Mechanisms
  • Signals
  • Pipes/named pipes
  • Shared Memory
  • Memory-Mapped files
  • Sockets
  • Message Passing libraries
  • Others...
Inter-Process Communication (IPC)

- Signals
  - Event notification
  - NO data exchange
  - Change execution flow
  - Programmable actions
Inter-Process Communication (IPC)

• Pipes/Named Pipes
  • FIFO memory buffer
  • N:M end-points
  • Synchronization mechanisms

![Diagram of process A and process B communicating through Unix pipes and kernel space.](image-url)
Inter-Process Communication (IPC)

• Shared Memory
  • Multiple processes share virtual memory space
  • Very fast
  • Needs explicit synchronization
Inter-Process Communication (IPC)

• Memory-Mapped files
  • A file allocated in virtual memory
  • Can be shared among processes

Inter-Process Communication (IPC)

• Sockets
  • Full-duplex communication
  • Between two processes
    • Alternative implementations
  • Synchronization mechanisms
  • Multiple socket types

Process A

sendto(sw...)

Kernel Space

recvfrom(sr...)

User Space

Process B

Kernel Space
Inter-Process Communication (IPC)

- **Message Passing**
  - Implemented through syscalls
    - Overhead
  - Message Queues
  - Structured messages

- **MPI**
  - Message Passing Interface
  - Library specification
  - Aim at parallel computing

- **Open source implementations of MPI**
  - OpenMPI, [https://www.open-mpi.org](https://www.open-mpi.org)
  - MPICH, [https://www.mpich.org](https://www.mpich.org)
Message based Communication

• Basic idea: divide the work into multiple communicated tasks
  • Blocking vs non-blocking; Synchronous vs asynchronous
  • System buffer (data in transit) vs app buffer (user managed address space)

• Point-to-Point communication
  • Messages between only two MPI tasks

• Collective communication
  • All processes of a given scope
Other Parallel Programming Models such as...

• CUDA (for GPGPU)
  • NVIDIA’s general purpose parallel computing architecture & programming model
    • GPU accelerates applications other than 3D graphics
  • Scale code to hundreds of cores running thousands of threads
    • E.g.: 1 Tesla: 240 cores; 128 threads per core → 30720 threads total

• MapReduce
  • Suitable for data-intensive parallel processing
  • Data is partitioned across the cluster in a distributed file system
  • Move computation to data and compute across nodes in parallel
Bibliography

• OpenMPI
  • https://www.open-mpi.org/

• OpenMP
  • https://www.openmp.org
  • OpenMP 5 reference guide C/C++ summary