Parallel Software

Parallel Computing – COSC 3P93

Course Outline

- Introduction
- Parallel Hardware
- Parallel Software
- Performance
- Parallel Programming Models
- Patterns
- SMP with Threads and OpenMP
- Distributed-Memory Programming with MPI
- Algorithms
- Tools
- Parallel Program Design

Today Class

- Parallel Software
  - Caveats
  - Coordinating the processes/threads
  - Shared-memory
  - Distributed-memory
  - Programming hybrid systems

Parallel Software

- Just particular class of systems are parallel
  - Operating systems, DBMS, Web servers
- Very little commodity software making extensive use of parallel hardware
- We cannot rely on hardware and compilers for performance
  - Applications need to be designed to exploit shared- and distributed memory architectures
- Terminology
  - Shared-memory systems → forking **threads**
  - Distributed-memory systems → **processes** and **tasks**
  - When in both systems (hybrid) → processes/threads

MIMD and SIMD

- MIMD
  - APIs of GPU grow at rapid pace → they are different from MIMD APIs
- SIMD / SPMD
  - Instead of running different programs on each core
    - Single executable behaving differently according to data
      - Use of conditional branches

```
if ( I'm thread/process 0)
    do this;
else
    do that;
```

- Note that
  - Program is **task parallel** → parallelism is by dividing tasks among threads or processes

Coordinating Processes/threads

- High performance → trivial in a few cases
  - Example: addition of 2 arrays

```
double x [ n ], y [ n ];
. . .
for (int i = 0; i < n ; i ++)
    x [ i ] += y [ i ];
```

- Parallelizing it
  - Just simple assignment of array elements to processes
    - Process 0 → 0,..., n/(p-1)
  1. Splitting the work
     - Same (even) amount of work
       - **Load balancing** → not knowing in advance the workload
       - Converting serial to parallel design → **parallelization**
         - Programs that are turned parallel by simply dividing work → **embarassingly parallel**
           - Usually parallelization is more complex
             - Requiring steps (2) and (3)
     - Minimizing communication
  2. Arrange processes/threads to synchronize
  3. Arrange communication among them

Shared Memory

- Variables/data can be shared or private
  - **Shared**
    - Read or written by any thread
  - **Private/local**
    - Only one thread accesses it
- Shared variables are essential for communication/interactions among threads
  - Implicit rather than explicit

Dynamic and Static Threads

- **Dynamic threads**
  - Master/slave paradigm
    - Master thread with a collection of empty worker threads
    - Master waiting for work requests
      - Upon a request → master forks a worker thread
      - Upon work termination → worker thread joins the master
    - Efficient use of resources
      - Only being used/reserved while the thread is running

Dynamic and Static Threads

- **Static threads**
  - master/slave paradigm
    - Master thread a with pool of forked (ready to go) threads
      - Threads run while the whole work is finished
      - Threads join master → master may do some cleanup
    - Less efficient use of resources
      - Threads reserve resources while idling
    - It avoids the time-consuming forking and joining operations
- If resources are available
  - Static paradigm is more efficient → less delays
  - Very close to the mindset of distributed-memory programming
    - Static paradigm is preferred

- ThreadPoolServer

Nondeterminism

- MIMD → asynchronous processor executions
  - Nondeterminism
    - Input can result in different outputs
    - Multiple independent thread runs:
      - Completing statements at different rates
        - Example → two threads and variable my_x
          - `printf ("Thread%d > my_val = %d\n", my_rank, my_x);`
            - Outputs
              - `Thread 0 > my val = 7`
              - `Thread 1 > my val = 19`
            - Or
              - `Thread 1 > my val = 19`
              - `Thread 0 > my val = 7`

Nondeterminism

- Nondetermism is not a problem in many cases
  - Ordering is not a problem or can be corrected
- In many other cases → specially shared-memory
  - It leads to disastrous results → program errors
- Example → threads computing an int
  - Values stored in private var. my_val and shared var. x (0)
    - `my_val = Compute_val ( my_rank );`
    - `x += my_val ;`
  - One possible execution → not as expected

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

  - From attempting to update the memory location x at the same time

- Nondeterminism.BadThreads

Race Condition

- Threads and processes in a “horse race”
  - Outcome depends on the ordering → who wins it
    - In the example → x += my_val
- **Critical section**
  - A block of code that can only be executed by one thread at a time
- **Mutually exclusive** access to the section
  - Program design’s responsibility
- Mechanisms → **mutual exclusion lock**, **mutex**, or **lock**
  - Support of underlying hardware
  - Critical section protected by a lock
    - Lock should be obtained for granting access to the section

Mutex Example

- Locks could be applied to the race condition example
  - **my_val = Compute_val (my_rank);**
  - **lock (& add my_val_lock);**
  - **x += my_val ;**
  - **unlock (& add_my_val_lock);**
- There is no predetermined order imposed to the threads
  - It just guarantees mutual exclusion
- Mutex downside
  - **Serialization** → one thread accessing at a time
    - **Parallelization contention**
- Thus
  - A code should have as few critical sections as possible
    - Critical sections as short as possible

Busy Waiting

- A loop whose sole purpose is to test a condition
  - Example
    - my_val = Compute_val (my_rank);
    - if (my_rank == 1)
      - while (! ok for 1); /∗ Busy−wait loop ∗/
    - x += my_val; /∗ Critical section ∗/
    - if (my_rank == 0)
      - ok for 1 = true ; /∗ Let thread 1 update x ∗/
-
- Wasteful method → waste of system resources for testing
  - No useful work is done and core runs thread

Semaphores

- A method for signalling processes and threads
  - Granting access to critical sections
- Different from mutexes
  - Semantically similar
- **Semaphores** allow types o thread synchronization that are easier than using mutexes
- **Monitor**
  - Higher-level mutual exclusion
  - Object-like entity whose methods can only be executed by one thread at a time

Transactions

- Commonly seen in DBMS
- To guarantee consistency and integrity
- All or nothing
  - A transactions is successful
    - If all involved steps are successful
- An error
  - **Rollback** the transaction
- **Transactional memory**
  - In shared-memory programs as transactions
    - Either a thread succeeded completely in a critical section
    - Or any partial results are rolled back
      - Critical section is repeated
- Example → Two-phase locking

Thread Safety

- **Thread safe**
  - Block of code that does not lead to unexpected results when used in multithread program → at any circumstance
  - Threads are nondeterministic by nature
- Example
  - In many cases, parallel programs can call functions developed for use in serial programs
    - Functions with declared local variables are allocated in stack
      - A thread has its own call stack → these are private variables
  - However → C allows declaration of static var. in functions
    - They persist from one call to the next
      - Shared among function callers
  - Practical example → strtok in C library (split string in tokens)
    - Static char* for subsequent calls
    - strtok is **not thread safe** → (strtok_example.cpp)

Distributed Memory

- In distributed memory
  - Cores directly access their own, private memories
- Sharing data → APIs
  - **Message passing**
- Distributed-memory APIs can be used in shared-memory hardware
  - Logically partition memory in private address spaces for threads
  - A library implement needed communication
- Distributed-memory programs composed of processes rather than threads
  - Usually such programs spans over independent CPUs with independent operating systems
- message_passing

Message Passing

- API at minimum → **send** and **receive** functions
- Processes using unique IDs → ranks (0, 1, ..., p-1)
  - Example of message passing

```
char message [100];
. . .
my rank = Get rank ();
if ( my rank == 1) {
    sprintf ( message , "Greetings from process 1");
    Send ( message , MSG CHAR , 100, 0);
} else if ( my rank == 0) {
    Receive ( message , MSG CHAR , 100, 1);
    printf ("Process 0 > Received: %s\n", message );
}
```

- Note
  - The program segment is SPMD
    - Two processes following the same executable

Message Passing

- Behaviour of **Send** and **Receive**
  - It depends on the API implementation
  - The simplest
    - A **blocking** *Send* call
      - It blocks until *Receive* starts
  - **Nonblocking** Send
    - Contents of the message placed into storage
  - Commonly → *Receive* **blocks** until caller receives
- Additional functions
  - **Broadcast** → collective communication
  - **Reduction** → combining results of multiple processes
- Widely used API → **Message-Passing Interface** (MPI)
  - Powerful and versatile → used by most powerful computers
  - Very low level → programmer needs to handle a lot of details
    - Called → the assembly language of parallel programming

One-sided Communication

- Usual communication
  - Two acting parts (send and receive)
- **One-sided communication** → remote memory access
  - Single process calling a function
    - Updating or retrieving value from memory
      - Local or remote memory
  - A simplification → just one process
    - Less overhead → less synchronization
    - Less function calls
- Example
  - Process 0 copies value into memory of process 1
    - 0 has ways to know it is safe to copy → overwriting
      - Synchronization of the processes
    - 1 has ways to know about the update
      - Flag variable → process 1 **polling** the flag
        - Busy checking it until flags shows update is complete
- **No direct interactions**
  - Overhead and introduce errors (hard to trace)

Partitioned Global Address Space Languages

- Some may prefer shared-memory programming
  - Instead of message passing or one-sided communication
- Shared memory techniques in programming distributed-memory hardware
  - **Not simple** → example
    - Just treating multiple distributed memory systems as a large memory → poor or unpredictable performance
      - Access remote memory may take hundreds or thousands times longer than local memory

```
shared int n = ...;
shared double x[n], y[n];
private int i, my_first_element, my_last_element;
my first element = ...;
my last element = ...;
/* Initialize x and y */
. . .
for ( i = my_first_element; i <= my_last_element; i++)
    x[i] += y[i];
```

- All x in core 0 and all y in core 1

Partitioned Global Address Space Languages

- **Partitioned global address space**
  - With tools to avoid delays and performance loss
    - Private variables allocated in local memory of the core running the code
    - Shared data structures used for programmer to control

Programming Hybrid Systems

- Merging/mixing shared-memory and distributed-memory systems
- Allying the best of both worlds
  - For programs requiring **high performance levels**
- **High complexity** → extremely difficult design
  - Application-specific

Input and Output

- Input and Output calls
  - **Major concern in parallel computing**
- High performance computing
  - **Minimizing I/O calls**
  - Or dealing with **little amount of data**
    - Not the case of Big Data Analytics
- Even the limited use of I/O functions cause problems
  - I/O language functions
    - No details about how they are performed
    - Example → printf and scanf in C, which is a serial language
  - A single process' threads share stdin, stdout, and stdrr
    - Simultaneous access to I/O
      - Nondeterministic outcome → not predictable

Class Recap

- Parallel Software
  - Caveats
  - Coordinating the processes/threads
  - Shared-memory
  - Distributed-memory
  - Programming hybrid systems

Next Class

- Performance
  - Speed up