CS6868 - Concurrent Programming

Introduction

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Quiz 1 = 10 marks, Quiz 2 = 10, Final = 40 marks.
- Programming assignments: Six. Total 40 marks.
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a good question, answer a chosen question, make a good point! Take 0.5 marks each. Max one mark per day per person.
- Attendance requirement – as per institute norms. Non compliance will lead to 'W' grade.
  - If you come to the class after 5 minutes - don't.
  - Proxy attendance - is not a help; actually a disservice.
- Plagiarism - A good word to know. A bad act to own.
  - Students Welfare and Disciplinary committee.

Contact (Anytime) :
Instructor: Krishna, Email: nvk@cse.iitm.ac.in, Office: BSB 352.
TA: Aman, Saurabh, Anchu:{amannoug,saurabhk,anchu}@cse, Office: PACE Lab.

Course outline

A rough outline (we may not strictly stick to this).

- Introduction
- Abstractions
- Memory Consistency models
- Design Patterns
- Languages: OpenMP, MPI, CUDA
- Optimizing parallel programs

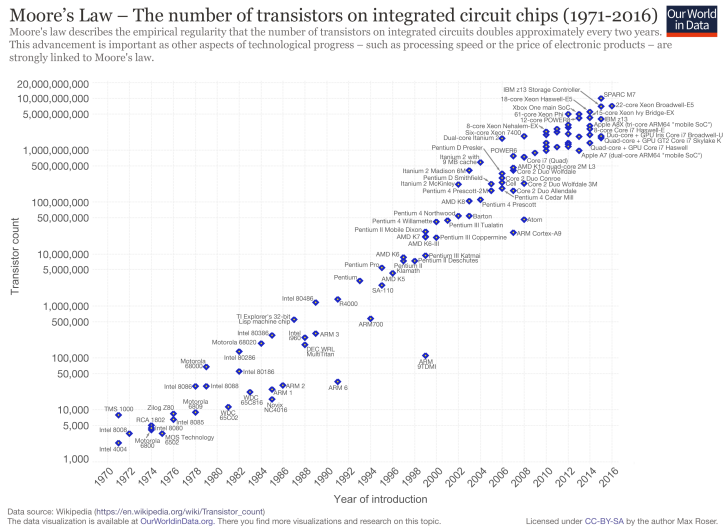
Books:

1. The Art of Multiprocessor Programming by Maurice Herlihy and Nir Shavit
2. OpenMP application Program interface (language reference)
3. The Complete Reference Java
4. MPI - the complete reference by Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, Jack Dongarra

Lecture schedule

1. Introduction
2. Critical sections
3. Java concurrency features
4. Data races (determinacy and atomicity)
5. OpenMP concurrency features
6. Synchronization (barriers, clocks, rendezvous, semaphores)
7. Mutual exclusion/Critical sections variations (h/w and s/w solutions), atomics, single, isolated
8. Recursive task parallelism
9. Deadlocks and livelocks
10. MPI concurrency features
11. Efficiency in parallel programs.
12. Patterns for multicore systems.

Why Multicores?



Focus on increasing the number of computing cores.

What, When Multicores? **Why not** Multiprocessors

- **What** A multi-core processor is composed of two or more independent cores. Composition involves the interconnect, memory, caches.
- **When** IBM POWER4, the world's first dual-core processor, released in 2001.
- **Why not Multi-processors**
  - An application can be "threaded" across multiple cores, but not across multi-CPUs – communication across multiple CPUs is fairly expensive.
  - Some of the resources can be shared. For example, on Intel Core Duo: L2 cache is shared across cores, thereby reducing further power consumption.
  - Less expensive: A single CPU board with a dual-core CPU Vs a dual board with 2 CPUs.

So what's the big deal?

- In the old world - processor speed used double every 1.5 years.
- Implication: increase the complexity of the program, wait for 1.5 years - same performance.
- New world: Unless there is concurrency in the program: you cannot speedup.
- Even if there is concurrency: parallelisation and synchronization do not come for free.
- Q: Say I get 8 parallel cores - will I get 8x speedup?

Challenges Involved

- Harnessing parallelism
  - How to map parallel activities to different cores? How to distribute data?
- Locality: Data and threads
- Minimizing the communication overhead
- Exploring fine grain parallelism (SIMDization), coarse grain parallelism (SPMDization).
- Dynamic code profiling and optimizations.
- Unpredictable performance Why?
- Programmability issues.

Programmability issues

- With hardware becoming increasingly multi-core, software developed without attention to parallel processing capabilities of the hardware will typically under-utilize the hardware - Example?
- When software is designed to operate in a multi-threaded or multi-processed manner, how the threads are mapped to the cores becomes an important issue - Why?
- Software that is critically dependent on multi-threading is always based on assumptions regarding the thread-safety of the function calls - Why?
- Multi-threading of software is generally very important to applications that involve human interactivity.
- Understanding different levels of parallelism.

A simple example: thread safety (more details later)

```
function int Withdraw(int amount){
     if (balance > amount) {
          balance = balance - amount;
          return SUCCESS;
     }
     return FAIL;
}
```

- Say `balance` = 100.
- Two parallel threads executing `Withdraw`(80)
- At the end of the execution, it may so happen that both of the withdrawals are successful. Further `balance` can still be 20!

Race freedom is enough?

```
void deposit(int amt) {
   acquire(m);
   balance = balance+amt;
   release(m);
}
int read_balance() {
   int t;
   acquire(m);
   t = balance;
   release(m);
   return t;
}
```

```
int withdraw(int amt) {
   int t = read_balance();
   acquire(m);
   if (t <= amt) {
      balance = 0;
   } else {
      balance = balance-amt;
      t = amt;
   }
   release(m);
   return t;
}
```

```
// Initial balance = 10.
fork  withdraw(10); ;
fork  deposit(10); ;
```

```
// Thread 1
// Thread 2
```

Example taken from Flanagan and Qadeer TLDI 2003.

Parallelism types

Instruction level parallelism.

- Parallelism at the machine-instruction level.
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 20 years.

Thread level parallelism.

- This is parallelism on a more coarser scale.
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP. Multicores are the way out to exploit the TLP.

What type of applications benefit from Multi-cores?

- Nearly All !
- Database servers
- Web servers (Web commerce)
- Compilers
- Multimedia applications
- Scientific applications, CAD/CAM
- In general, applications with Thread-level parallelism (as opposed to instruction-level parallelism)
- To build applications that benefit from Multi-cores, we have to understand multi-cores, on how they differ from unicore machines.

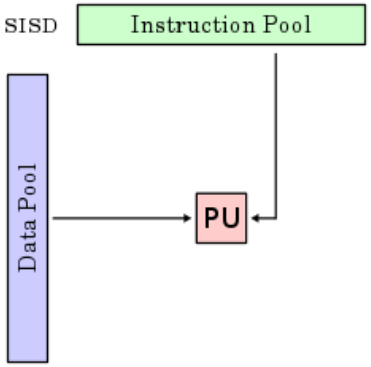
Flynn's Taxonomy.

Categorization of computers based on number of instruction and data streams[1].

- SISD: Single instruction Single Data - x86: sequential computer which exploits no parallelism in instruction or data streams.
- SIMD: Single instruction Multiple Data - Vector machines: A computer which exploits multiple data streams against a single instruction stream.
- MISD: Multiple instruction Single Data - Space Shuttle - Multiple instructions operate on a single data stream.
- MIMD: Multiple instruction Multiple Data - Bluegene, Cell - Multiple autonomous processors simultaneously executing different instructions on different data.

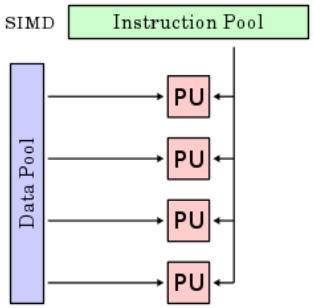
[1] Flynn, M. (1972). "Some Computer Organizations and Their Effectiveness". IEEE Trans. Comput. C-21: 948.

SISD



- Traditional Von Neumann Architecture, all traditional computations.
- a single processor, a uniprocessor, executes a single instruction stream, to operate on data stored in a single memory.
- Pipelined execution allowed.

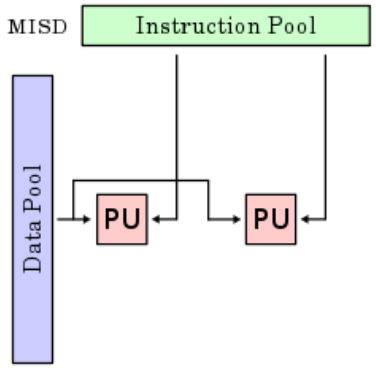
SIMD




```
for (int i=0;i<16;++i) A[i] = B[i] + C[i]
```

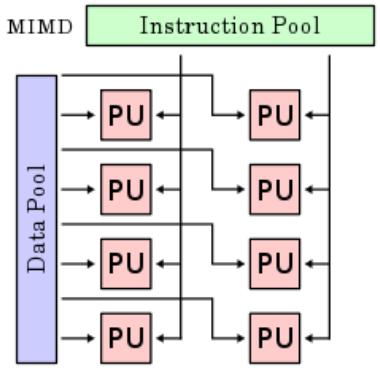
- Fetching / Write a bulk of data is efficient than single units of data.
- A compiler level optimization to generate SIMD instructions.
- Not all algorithm can be vectorized - for instance, parsing.
- increases power consumption and chip area.
- Detecting SIMD patterns is non-trivial.

MISD



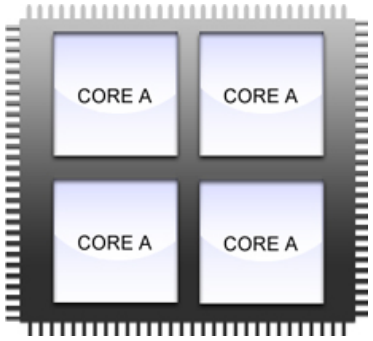
- Task replication for fault tolerance.
- Not used in practise. No known commercial system.

MIMD



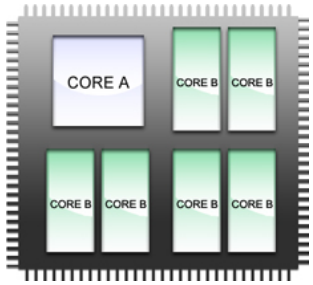
- Many processors that function asynchronously.
- Memory can be shared (less scalable) or distributed (memory consistency issues).
- Most of the modern parallel architectures fall into this category.

Different types of MIMD systems - homogeneous



- Homogeneous multi-core systems include only identical cores.
- Just as with single-processor systems, cores in multi-core systems may implement architectures like superscalar, VLIW, vector processing, SIMD, or multithreading.

Different types of MIMD systems - heterogeneous



- Mixture of different cores e.g.
  - a computational unit could be a general-purpose processor (GPP),
  - a special-purpose processor (i.e. digital signal processor (DSP)
  - a graphics processing unit (GPU)),
  - a co-processor, or custom acceleration logic
- Each core may be optimized for different roles.
- Clusters are often heterogeneous; future supercomputers mostly will be heterogeneous systems. Examples: Grids, lab clusters.

Pros and Cons

Homogeneous CPU multi-cores

Pros:

- Easier programming environment
- Easier migration of existing code

Cons:

- Lack of specialization of hardware to different tasks
- Fewer cores per server today (Typically less than 100)

Heterogeneous multi-cores

Pros:

- Massive parallelism today
- Specialization of hardware for different tasks.

Cons:

- Developer productivity - requires special training.
- Portability - e.g. software written for GPUs may not run on CPUs.
- Organization - multiple GPUs and CPUs in a grid need their work allocated and balanced and event-based systems need to be supported.

Challenges Involved (revisited)

- Harnessing parallelism
  - How to map parallel activities to different cores? How to distribute data?
- Locality: Data and threads. What is the challenge?
- Minimizing the communication overhead
- Exploring fine grain parallelism (SIMDization), coarse grain parallelism (SPMDization).
- Assist threads
- Dynamic code profiling and optimizations.
- Programmability issues.

Programmability issues

- With hardware becoming increasingly multi-core, software developed without attention to parallel processing capabilities of the hardware will typically under-utilize the hardware - Example?
- When software is designed to operate in a multi-threaded or multi-processed manner, how the threads are mapped to the cores becomes an important issue - Why?
- Software that is critically dependent on multi-threading is always based on assumptions regarding the thread-safety of the function calls - Why?
- Multi-threading of software is generally very important to applications that involve human interactivity.
- Understanding different levels of parallelism.
- Debugging parallel programs.

Starting model

- Multiple threads
- Single shared memory
- Objects live in memory
- Unpredictable asynchronous delays

Primality testing

- Challenge
  - Print first 1010 primes
    or
    Print primes from 1 to 10^{10}
- Given
  - Ten-processor/Ten-core multiprocessor
  - One thread per processor/core
- Goal
  - Get ten-fold speedup (or close)

See the slides from the ppt.

Speedups in Parallel Programs

- Say a serial Program P takes T units of time.
- Q: How much time will the best parallel version P' take (when run on N number of cores)? $\frac{T}{N}$ units?
- Linear speedups is almost unrealizable, especially for increasing number of compute elements.
- $T_{total} = T_{setup} + T_{compute} + T_{finalization}$
- T_{setup} and $T_{finalization}$ may not run concurrently - represent the execution time for the non-parallelizable parts of code.
- Best hope : $T_{compute}$ can be fully parallelized.
- $T_{total}(N) = T_{setup} + \frac{T_{compute}}{N} + T_{finalization}$ ..... (1)
- Speedup $S(N) = \frac{T_{total}(1)}{T_{total}(N)}$. In practice?
- Chief factor in performance improvement : Serial fraction of the code.

Amdahl's Law

- Serial fraction $\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)}$
- Fraction of time spent in parallelizable part = $(1-\gamma)$

$$T_{total}(N) = \underbrace{\gamma \times T_{total}(1)}_{\text{serial code}} + \underbrace{\frac{(1-\gamma) \times T_{total}(1)}{N}}_{\text{parallel code}}$$

$$= \left(\gamma + \frac{1-\gamma}{N}\right) \times T_{total}(1)$$

$$\text{Speedup } S(N) = \frac{T_{total}(1)}{\left(\gamma + \frac{1-\gamma}{N}\right) \times T_{total}(1)}$$

$$= \frac{1}{\left(\gamma + \frac{1-\gamma}{N}\right)}$$

$$\approx \frac{1}{\gamma} \quad \text{... Amdahl's Law}$$

- Max speedup is inversely proportional to the serial fraction of the code.

Implications of Amdahl's law

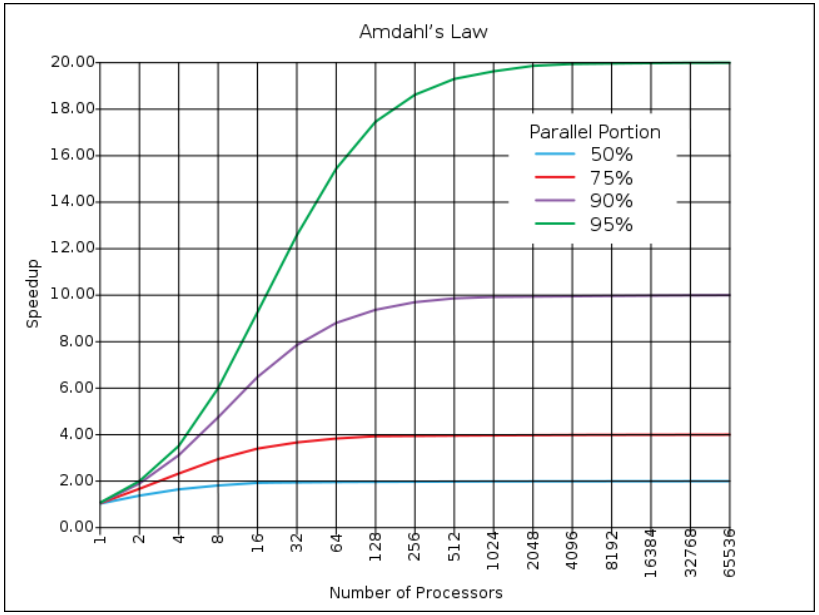
Assume: Ten processors. Goal: 10 fold speedup.

Serial fraction	Parallel fraction	Speedup = $\frac{1}{(\gamma+\frac{1-\gamma}{N})}$
40 %	60 %	2.17
20 %	80 %	3.57
10 %	90 %	5.26
99 %	01 %	9.17

Implications of Amdahl's law

- As we increase the number of parallel compute units, the speed up need not increase - an upper limit on the usefulness of adding more parallel execution units.
- For a given program maximum speedup nearly remains a constant.
- Say a parallel program spends only 10% of time in parallelizable code. If the code is fully parallelized, as we aggressively increase the number of cores, the speedup will be capped by ($\sim$) 1.11$\times$.
- Say a parallel program spends only 10% of time in parallelizable code. Q: How much time would you spend to parallelize it?
- Amdahl's law helps to set realistic expectations for performance gains from the parallelization exercise.
- Mythical Man-month - Essays on Software Engineering. Frederic Brooks.

Peaking via Amdahl's law



Limitations of Amdahl's law

- An over approximation : In reality many factors affect the parallelization and even fully parallelizable code does not result in linear speed ups.
- Overheads exist in parallel task creations/termination/synchronization.
- Does not say anything about the impact of cache - may result in much more or far less improvements.
- Dependence of the serial code on the parallelizable code - can the parallelization in result in faster execution of the serial code?
- Amdahl's law assumes that the problem size remains the same after parallelization: When we buy a more powerful machine, do we play only old games or new more powerful games?

Discussion: Amdahl's Law

- When we increase the number of cores - the problem size is also increased in practise.
- Also, naturally we use more and more complex algorithms, increased amount of details etc.
- Given a fixed problem, increasing the number of cores will hit the limits of Amdahl's law. However, if the problem grows along with the increase in the number of processors - Amdahl's law would be pessimistic
- Q: Say a program P has been improved to P' (increase the problem size) - how to keep the running time same? How many parallel compute elements do we need?

Gustafson's Law

- Invert the parameters in Eq(1):
  $T_{total}(1) = T_{setup} + N \times T_{compute}(N) + T_{finalization}$ ..... (2)
- Scaled serial fraction $\gamma_{scaled} = \frac{T_{setup}+T_{finalization}}{T_{total}(N)}$.
- $T_{total}(1) = \gamma_{scaled} \times T_{total}(N) + N \times (1 - \gamma_{scaled}) \times T_{total}(N)$
- $S(N) = N + (1 - N) \times \gamma_{scaled}$ ..... (Gustafson's Law)
- We are increasing the problem size. If we increase the number of parallel compute units - execution time may remain same (provided γ_{scaled} remains constant).
- It means that speedup is linear in N. Is it contradictory to Amdahl's law?

Comparison Amdhal's law and Gustafson's law

- Say we have program that takes 100s. The serial part takes 90s and the parallelizable part takes 10s.
- If we parallelize the parallel part (over 10 compute elements) the total time taken = 90 + $\frac{10}{10}$ = 91s.

Amdahl's law:	Gustafson's law:
$\gamma = 0.9$	$\gamma_{scaled} = \frac{90}{91} = 0.99$
Speedup $\approx \frac{1}{0.9} = 1.1$	Speedup(10) $= 10 + (1 - 10) \times 0.99 = 1.1$

- Speedups indicated by both Gustafson's Law and Amdahl's law are same.
- Gustafson's Law gives a better understanding for problems with varying sizes.

Bottlenecks in Parallel applications

- Serial part of the code (Amdahl's law).
- Traditional programs running on Von-Neumann Architectures - memory latency.
- The "memory wall" is the growing disparity of speed between CPU and memory outside the CPU chip.
- In the context of multi-core systems, the role of memory wall?
- Communication latency plays a far major role.
- Communication = remote task creation, sending data, synchronization etc.

Moral

Making good use of our multiple processors (cores) means

- Finding ways to effectively parallelize our code
- Minimize sequential parts
- Reduce idle time in which threads wait without compromising on correctness.

Means of Communication

Tasks/Threads/Processes need to communicate with each other for the program to make progress.

- Remote procedure calls.
- Shared memory.
- Message Passing.
- Synchronization.
- Examples: Files, Signals, Socket, Message queue, pipe, semaphore, shared memory, asynchronous message passing, memory mapped file.

Remote Procedure Calls

A subroutine or procedure to execute in another address space (core/processor), with no explicit coding.

- Typically, RPC is an synchronous event. While the server is processing the call the client is blocked.
- Easy to program, especially in reliable environments.
- Compared to local calls, a remote procedure may fail. Why?
- How to handle failure?
- By using RPC, programmers of distributed applications avoid the details of the interface with the network.
- The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.
- Examples: C, Java RMI, CORBA.
- Read yourself.

Shared memory

A large common RAM shared and simultaneously accessed by the multiple cores.
Note: Communication inside a task via memory is not generally referred to as 'shared memory'.

- Easy to visualize for the programmer.
- Communication can be fast.
- (Partitioned) Global Address Space.
- Scalable, especially for small number of cores.
- Not easily scalable for large number of cores.
- Cache coherence issues - Say a core updates its local cache - how to reflect the changes in the shared memory such that data access is not inconsistent.
- `#pragma omp flush [a, b, c]` : A synchronization point where memory consistency is enforced.
- `#pragma omp parallel private (a)`

Message passing

- Allows communication between processes (threads) using specific message-passing system calls.
- All shared data is communicated through messages
- Physical memory not necessarily shared
- Allows for asynchronous events
- Does not require programmer to write in terms of loop-level parallelism
- scalable to distributed systems
- A more general model of programming, extremely flexible
- Considered difficult to write
- Difficult to incrementally increase parallelism
- Traditionally - no implicitly shared data (allowed in MPI 2.0)