Parallel Algorithms

Lecture 1: introduction

Ronald Veldema

Administrative

- 1^{st} or 2^{nd} week of November
  - No lecture, I'm in boston
  - Last lecture: invited speaker?
- I also do the exercises
  - Send me an email if you want to participate
    - My email = veldema@cs.fau.de
    - Mail your:
      - Name, Email address, matricel number
  - Slides
    - http://www2.informatik.uni-erlangen.de/~veldema/parallel_algorithms/index.html

---

Administrative

- Pruefungen etc
  - Mundliche pruefung
    - Email prof. philippsen at the end of the lecture series for an apointment
    - Me = veldema@cs.fau.de
  - 2+2 SWS, 4 leistungs punkte
  - Prof. Philippsen (philippsen@cs.fau.de)
    - For Special circumstances….
  - Website informatik 2 (www2.cs.fau.de) -> Lehre -> Studienplan &Hauptdiplomspruefungen
    - If computational engineering, wirtschafts inf. Etc
    - Scheine (benoted, unbenoted, etc)

Books

<available in my room: 05.155>

- Algorithms Sequential & Parallel
  - Russ miller, Laurence Boxer
- Designing and Building Parallel Programs
  - Ian Foster
- An Introduction to Parallel Algorithms
  - Joseph Jájá

Why should I care ?

- how long would SETI@home take without parallel computing ?
  - Already spent 1625048 years…
  - how does seti@home work ?
- My algorithm is too slow, but I've tried everything else already !
  - Real time…
  - Years to complete…
- Most programming environments have *some* support for parallel programming already…

What is parallel programming ?

- What is a parallel algorithm ?
  - program that runs on a parallel machine
  - ….and processors cooperate to solve a problem
- Shared or distributed memory ?
  - This course: both

What shall we talk about in this course ?

- Parallel programming techniques
- Algorithms used in parallel programming
- Parallel computing environment
  - Software (languages, libraries)
  - Hardware (networks, machines)
- Lightly touch on theory
- NOT Prof. Schneiders course
- Follow up on “cluster computing” course

Non-deterministic Turing Machines (1)

- The ultimate parallel machine ?
- Deterministic turing machine
  - State + transition rule → next state
- Non-deterministic turing machine
  - State + transition rules → alternative next states
  - Try all alternative next states in parallel
    - If one of these accepts/halts, the NDTM halts/accepts

Non-deterministic Turing Machines (2)

- NDTM
  - Set of states Q,
    - startstate s in Q
    - F in Q are accepting states (input accepted by program)
  - Tape alphabet G
    - B = blank / no-op
  - Input alphabet S
  - D: (Q,G) → (Q, {L,R,S})
- Every NDTM can be simulated by a DTM
  - A DTM will take more time than the NDTM
    - Question: how much more ?

Non-deterministic Turing Machines (3)

- NDTM
  - Set of states Q,
    - startstate s in Q
    - F in Q are accepting states (input accepted by program)
  - Tape alphabet G
    - B = blank / no-op
  - Input alphabet S
  - D: (Q,G) → (Q, {L,R,S})
- Every NDTM can be simulated by a DTM
  - A DTM will take more time than the NDTM
    - Question: how much more ?
      - This is exactly the question of P=NP ($1M prize money)

Non-deterministic Turing Machines (4)

- NDTM = quantum computer ?

Non-deterministic Turing Machines (5)

- NDTM = quantum computer ?
  - No: NP complete problems can not be solved in P time by a quantum computer
  - (there are also problems a quantum computer can solve in P but a NDTM not)

Quantum Computing

- Pairs of atoms/light ‘particles’ can be brought into an ‘entangled’ state
  - One particle magically ‘knows’ what happens to its brother/sister particle
- Qubits = 0, 1 or in between. Each state has a probability attached
- Generic operation of searching on a quantum computer
  - Try all values at the same time, one is ofcourse ‘special’
  - When reading the output, we get the selected answer only
    - Why ? The selected answer is entangled with the ‘wrong’ answers
    - More later…

In practice...

- take a problem
- divide in to small pieces (“tasks”)
  - called “task decomposition”
- connect pieces
  - called “task dependency graph creation”
- combine pieces
  - “coarsening”
- distribute pieces over processors
- combine results

why are some problems harder to parallelize than others ?

- hard to subdivide problem
- one piece is dependent on results of other pieces
- is in parallel but requires extreme amounts of communication between tasks
- complex transformation of sequential algorithm needed
- Many tradeoffs, creating good parallel programs is as much art, skill as technique…

limits to parallelism ?

- given problem with 3 components A,B,C:
  - seq(A).par(B).seq(C) then time(A)+time(longest component B)+time(C)
  - spawn time, sequential parts, reconcile
  - scalability: will resources scale ?
    - network, routers, mean time between failures (MTBF)
    - will it run with 1000 cpus, a million ?

Parallel vs Sequential ?

- Theory:
  - Parallel Computation Thesis:
    - Time on any parallel machine model is equivalent to sequential log(space)
    - Sequential space is a polynomial of parallel time.
  - This is a ‘thesis’ (unproven claim) by Turing&assoc.
  - Idea: every processor needs a little private scratch space
    - Using Turing machine: each machine needs ‘scratch’ space to hold compute state (temporary variables).
    - One processor:
      - need a lot of scratch to handle all input
      - Each scratch variable is used a lot
    - Many processors: scratch space is made smaller per processor

Concurrency Control (1)

- programmer implied program invariants:
  - stack: after push, size = old_size + 1
  - list: tail has no next, head has no previous
- invariants broken in intermediate program states
  - intermediate program states are observable when running in parallel..
- consistency control: restrict observability of intermediate states

Concurrency Control (2)

- Ways to perform concurrency control
  - Atomic actions using
    - Monitors
    - Locks
    - Semaphores
    - (all these will be handled in lecture on multi-threading)
  - Detection & recovery
  - (Temporary) Privatization

Concurrency Control (3)

- Detection & recovery
  - The hacker's way of concurrency control
  - Perform your operation as usual but
    - Record state before operation
    - afterwards check if the operation was indeed successful
    - If not entirely successful, try to patch using the saved state

Concurrency Control (4)

- Example: add node to linked list
  - Before adding, record what the old list-tail was
  - Afterwards, atomically test if you are the tail and if not, atomically restore list state

Concurrency Control (5)

- Try and avoid thread synchronization if possible (expensive operation)
  - Example: list manipulation using 'dijkstra's observation'
    - When using a producer consumer pattern, we can avoid synchronization when "length(list) > number_of_tasks"
      - When #consumer tasks > 1 then atomic test-and-set required to mark list elements as 'taken'

Concurrency Control (6)

- Privatization
  - Instead of maintaining a single shared resource for all tasks, give each task his own private resource
  - Example:
    - Maintain a list of best solutions for a search algorithm
      - Shared list of best solutions
        - protect list integrity each 'add'
      - Each search task has own list of best solutions thusfar
        - Lists of each task merged after everyone is done

Bad concurrency control (1)

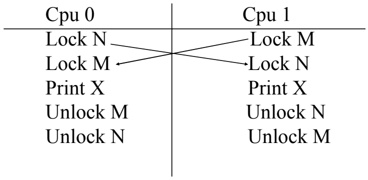
- Race conditions
  - Two cpus concurrently try to read or write some data where atleast one is a write
  - No concurency control over that data item

Cpu 0	Cpu 1
X = 5	print X
Y = 3	Y=6

Prints old or new X ?
Is the resulting Y from cpu 0 or 1 ?

Dead lock (1)

- A number of cpus concurently try and ackuire some resource which won’t be available
- All cpus wait for the resource to become available



Cpu 0	Cpu 1
Lock N	Lock M
Lock M	Lock N
Print X	Print X
Unlock M	Unlock N
Unlock N	Unlock M

Dead lock avoidance: Java ?

- Can the previous example be written in Java ?
- Can you write a dead-lock in Java using synchronized blocks ?

Live lock

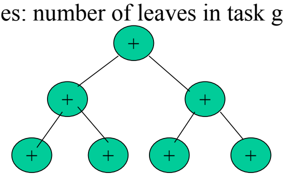
- Cpu is inside a set of states S
  - S has no transition out of S
  - All states in S perform no useful action
  - in each state in S there is a transition to another state in S

degrees of parallelism (1)

- fine-grained parallelism
  - tasks communicate often (per millisecond)
- coarse-grained parallelism
  - tasks communicate often (every second)
- embarrassingly parallel
  - tasks communicate every hour / at startup & shutdown
  - “independent tasks”

degrees of parallelism (2)

- Maximum
  - trees: number of leaves in task graph
- average
  - trees: number of leaves in task graph / 2



Deterministic algorithms

- deliver the same answer each run
- sometimes not most efficient as non-det.
- some reasons for non-determinism
  - no influence over thread scheduler
  - don’t know when a message over a network arrives

‘easy’ parallel problems

- parallel sorting
- parallel search for a data item
- parameter sweep
  - given a function F, find interesting spots in F(x) by permuting over all ‘x’

‘hard’ problems

- molecular dynamics
  - each molecule has some influence on other molecules requiring lots of communication
  - more in later lecture in this series.
- fluid dynamics
  - pressure in one spot has influence on neighbors, which in turn has some influence on their neighbors

Parallel programming machines

- shared memory machines
  - architecture/software simulates a shared global memory for all cpus: each cpu ‘sees’ the same memory.
- distributed memory machines
  - communication between processors is by explicitly sending messages
  - can simulate a shared memory machine on top of a distributed memory machine !

Shared memory machines: threads

- Threads share everything inside a process except the call stack and their own registers
- Memory and file descriptors are shared
- More on threads in later lecture

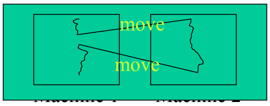
```
Void foo() {
 // compute
}

Void main() {
    Create_thread(foo);
    Create_thread(foo);
}
```

Distributed Memory Machines

- Message passing
  - Location independence
    - We don’t need to know the address of the machine to send a message to. Instead use logical machine id’s
  - Network independence
    - Don’t need to know about underlying network layer (TCP/IP, myrinet, ATM, infiniband, etc)

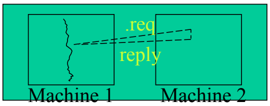
Distributed Memory Machines: function shipping

- Function shipping
  - (temporarily) let the thread (of control) move to the machine that has the data
  - Remote Procedure Call (RPC)



Distributed Memory Machines: Data shipping

- Data shipping
  - move the data that the thread currently needs to the machine running the thread
  - access object fault -> send_data_request-> reply_with_data -> map object



What kind of machines do we have ?

- Irrespective of processor interconnect:

	1 cpu	N cpus
1 data flow	SPSD	MPSP
M data flows	SPMD	MPMD

(sp/mp = single/multiple program,
sd/md = single/multiple data streams

Parallel programs in the real world

- In-order execution
- Out-of-order execution
- Super scalar machines
- Predication
- Symmetric multi processor (SMP) machines

- VLIW (Intel's IA64)
- Cluster computing
- Grid computing

Measuring Parallel Performance

- Speedup
  - (Time with N cpus) / (time with 1 cpu)
    - Time with 1 cpu = with fastest sequential algorithm
  - Super linear speedup
    - Happens when bottleneck is resolved when using multiple processors
      - Example: problem requiring 500MBytes of memory swaps on a machine with 256MBytes of memory but will fit in memory when using two machines.
- Efficiency
  - (Speedup with N cpus) / N

Why can't every algorithm be parallelized ? (1)

- When task X depends on the results of all tasks 0 … (X-1)
- Data dependencies
  - True data dependencies:
    - Y = X + 1
    - Z = Y + 1
  - False data dependencies:
    - X = X + 1
    - X = 3
- Note: you can still *use a different algorithm* that is parallelizable !

Why can't every algorithm be parallelized ? (2)

- Resource dependencies
  - X = X * 10
  - Y = Y * 3
  - Can't run in parallel if machine only has one multiplication unit
- Procedural dependencies
  - Void foo() { X = X + 1; }
  - Void zoo() { Y = Y + 1; foo(); }

Reasons for badly performing parallel algorithms

- Bad load balancing
  - One cpu does all the work while others do nothing
- Bad choice of granularity
  - Too many messages sent to achieve high performance
- Parallel algorithm is far worse than sequential algorithm on 1 cpu
- Work performed multiple times