Parallel Algorithms

Lecture 2: Theory Overview

Ronald Veldema

Domain or Functional Decomposition ?

- Domain: partition data of the algorithm
  - Subdivide input
- Functional: partition algorithm, then deal with associated data
  - Pull apart code (loops, recursion) to allow parallel execution
- Can mix both !

Exploratory Partitioning

- Given a search space, everybody searches a partition of the search space
- Search states visited can be very different from states visited by sequential algorithm !
  - Example: find minimum value of set of numbers
- May cause non determinism !
- Question: is this functional or domain decomposition ?

Example: Exploratory Partitioning

- Find minima in function F(x) by trying all 'x'
  - Should work for all functions without requiring proofs
  - What is the speedup ?
- Game tree search: checkers for example
  - I have N possible moves, try all N in parallel
  - Try all decendent possible moves in parallel etc
  - Stop when win or search_depth > threshold
  - What is the speedup ?
    - Lucky guesses ?

Speculative Partitioning (1)

- Looking into the future, try all paths that will be taken
- When finally reaching a point in the program where a decision on taken path needs to be taken, choose precomputed data

Speculative Partitioning (2)

- Example:
  - A = foo()
  - Switch (A) {
    - Case 1: X = goo(); break;
    - Case 2: X = zoo(); break;
    - }
  - Print X;

Compute in foo, goo and zoo in parallel.
When foo finishes, take the result of the appropriate
Computed X (from goo or zoo)

Recursive partitioning

- Will be handled later
  – Sneak preview: handle all recursive invocations in parallel.

Input data partitioning (1)

- Different from exploratory partitioning because exploratory partitioning can cut off search space while data partitioning continues until search space is exhausted
- Each task handles part of input space as well as it can
- A.K.A. owner computes

Input data partitioning (2)

- Example 1: sort N numbers with M cpus
  – Each cpu handles N/M numbers
  – Each 2nd cpu merges with its neighbor
  – Each 4th cpu merges with its 2nd neighbor
  – etc

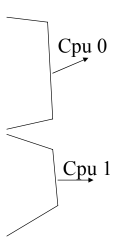
Input Data Partitioning (3)

- Example 2: data mining
  – Data mining = finding interresting correlations in data sets
  – Supermarket: which items are sold together most ?
    - P = set of transactions
    - Partition P in M sets, 1 set for each of M cpus
    - Each cpu: compute frequency table for all permutations of items
    - Afterwards, sum frequency tables from all cpus

Input Data Partitioning (4)

Transactions:

Peer Apple Banana
Peer Apple Bread
Peer Apple Sardines
Meat Sardine Bread
Meat Bratwurst Bread
…
…
…
…
…
…
…



Input Data Partitioning (5)

Cpu 0 makes frequency table:

Peer Apple	3x
Apple Banana:	1x
Meat Bread	2x
Apple Sardine	1x
…	
…	
…	
etc	

Cpu N makes frequency table:

Peer Apple	6x
Apple Banana:	2x
Meat Bread	8x
Apple Sardine	2x
…	
…	
…	
etc	

Output data partitioning (1)

- Each task computes part of the output value
- A.K.A. “owner computes”
- Can only be done if tasks independent

Output data partitioning (2)

- Example: matrix multiplication A*B = C
  - Matrix NxN, 4 cpus then each cpu computes submatrix (N/2)x(N/2)
  - Subdivide C in 4 parts
    - (A11 A12)   (B11 B12)   (C11 C12)
    - (A21 A22) * (B21 B22) = (C21 C22)
  - More parallelism required ?
    - Split C11, C12, C21, C22 in 4 parts, etc

Output Data Partitioning (3)

- Example: data mining a supermarket
  - Which items are sold together most ?
    - P = set of all permutations of items
    - Partition P over N cpus
    - Duplicate transaction database
    - Perform frequency analysis over all transactions
    - Merge frequency analysis data

Output Data Partitioning (4)

Items = { Apple, Banana, Meat, Bread, Sardine}
Permutions = {
Apple Banana,
Apple Meat,
Apple Bread,
Apple Sardine,
Banana Meat,
Banana Bread,
Banana Sardine
Meat Bread
Meat Sardine
…
…

Cpu 0 — Cpu 0 computes how many times Apple-Banana pairs sold, Apple-Meat Pairs sold, etc

Cpu 1

Intermediate partitioning

- Given a multistage problem, partition one of the intermediate stages
- If problem has only one stage, try and rewrite algorithm to a multistage algorithm
- Example: find most co-sold items
  - (1) find most sold items, (2) build freq. Table, (3) test co-sold property
  - Parallelize on step 2

Hybrid partitioning

- In a multistage algorithm, use different partitioning schemes for the different stages
  - Example: in the data-mining example, after finding ‘hot’ co-sold items using *output partitioning*, see if supplier has better/cheaper alternatives using *input partitioning*
    - 2 stage problem: data mining, finding better products
      - Contact all suppliers in parallel when finding a hot co-sold pair during data-mining process

Task generation

- Task generation is the process of creating work descriptors
  - Work descriptors can take many forms
    - Type of thread, object, array element, …
- Cpus accept work descriptors and perform the task
- Questions:
    - Generate all tasks at startup ?
    - While the program is running ?
    - Can sibling tasks influence each other ?
    - Do prior tasks influence the meaning of future tasks ?

Static task generation

- Generate all tasks at start of algorithm
- Example:
  - search for the minimum of a set of M numbers using N cpus
    - Generate N tasks, each searching M/N numbers
    - Afterwards, one processor searches the minimum of the N found minima
- Advantage: possible to statically map tasks to processors
  - Allows compile time knowledge to be applied

Dynamic task generation (1)

- While a task is running, it may generate additional tasks
- Advantage:
  - Adaptability to different network layouts/number of cpus
- Disadvantage:
  - Don't know apriori how large a task may be

Dynamic task creation (2)

- Example: game tree search
  - Find path from X to Y in labyrinth
    - Create all paths from X to neighbours
      - (creates new tasks to expand neighbours)
    - From neighbours expand paths to all neigbour's neighbours
    - Etc until we find an expansion of a node whose neighbour is Y
    - Parallel: cpus put/get 'expand' tasks in/from queue
  - Each expansion dynamically creates new "expand" tasks

Uniform/non uniform task sizes

- Are all tasks of the same size ?
  - Then it's a uniform task creation algorithm
    - When giving each cpu a partition of input data set its often uniform if there is no dynamic task creation
  - Do we know a-priori how long a task will take ?

Read-only/read-write task interactions

- Do tasks only read data from other tasks or do they also modify data owned by other tasks
- Example: find minimum value of a function
  - Partition input data amongst cpus
  - When finding a minimum broadcast it
  - When a partial evaluation is larger than current minimum quickly give up

Synchronous task interactions

- Task I and task J cooperate when sending messages
  - Task I performs a send
  - Task J performs a receive
- Message passing libraries often work this way

Asynchonous task interactions

- Task I informs task J of an event by directly writing into J's memory
  - Task J performs no explicit receive !
  - Threaded programs often work this way
    - One thread writes global minimum value while others concurrently read it

Static task interactions

- Always know that at some location cpu X will communicate with cpu Y
  - This is especially true in SPMD programs
- Example:=
  - For I=0;I<10;I++)
    - A[I*2] = A[I*2+1] + 14
  - Assume a[x] with x = even on cpu 0, with x uneven on cpu 1

Dynamic task interactions

- Do not a-priori know if communication will take place or to which cpu
  - Happens with MPMD programs
  - Example:
    - If (x > y) send_message(z) to cpu 5

Irregular task interactions

- We do not a-priori know with which processor we will communicate
  - Happens with MPMD machines
- Example:
  - Send message(Z) to cpu N, where N is the result of some computation
- Also known as "irregular problems)

Task mapping (1)

- Map task X to cpu Y
  - Lots of tunable parameters in this question to gain best performance
  - Generally: Tradeoff between load balancing and number of messages sent over network

Task mapping (2)

- Possible to statically load balance if
  - Static task creation
  - Uniform task sizes
  - (we know how long each task will take and how many tasks there are)
  - Allows all kinds of compiler optimization
    - Explicit send/receive
    - Loop/data transformations, etc

Task mapping (3)

- Can do even better if
  - Know data partitioning
  - Static task interactions
    - know when task X will communicate with task Y
  - Regular task interactions
  - Synchronous task interactions
  - Know network/cpu speed

Task mapping (4)

- Global scheduling
  - Use centralized knowledge to map tasks to processors
  - Example: M tasks with N cpus, M >> N and non-uniform task sizes
    - Cpu 0 tells cpus 1-N to run a task.
    - When cpu X is done, it asks cpu 0 for another
  - Typically, the centralized processor becomes a bottleneck

Task mapping (5)

- Local scheduling
  - Let each processor decide what to do on its own
  - Typically less globally optimal

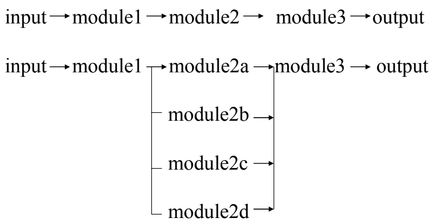
Data mapping

- Later lecture

Pipelining (1)

- Problem is splitable in chunks
- Each sub-problem is dependent on the previous chunk
- Performed in about all new processors
- Can be performed in software as well !
- speedup = length of pipeline / by communication speed for slowest module

Pipelining (2)



Pipelining (3)

- Pipelining to hide latencies
  - Example
    - Load reg1 = mem
    - Nop
    - Nop
    - Reg1 = reg1 + 1
    - Store mem = reg1

Pipelining (4)

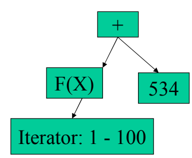
- Pipeline aborts
  - When inserting an jump into a pipeline, the already loaded insns after the jump need to be aborted

[prefetch] [decode] [load-operands] [execute] [store_results]

Here we know where we jump to and
Remove the insns that are in the stages before
The "execute" stage

Pipelining (5)

- Data flow languages use extensive pipelining to create parallel programs
- Data flow languages use mostly "visual" programming



Precise vs Imprecise parallel algorithms

- The parallel algorithm delivers a (slightly) different answer than the seq. algorithm
- Does it matter if the answer is off by 0.1% ?
  - Perform unsynchronized writes…
  - Remove fifoness of message queues…
  - Generally remove synchronization…

Randomized Algorithms (1)

- Partition data at random locations
- Replicate data & send work to random locations
- Note: random algoritms often provably optimal !
  - No administrative overheads !
- Note: random != random

Randomized Algorithms (2)

- Example: database with parallel processes accessing the database
  - Non randomized: use a directory (telephone book)
  - Data d = database.get_object("monkey");
    - String "monkey" sent to directory machine
    - Directory machine returns machine-id which holds the data
    - Requesting machine can now send request to correct machine

Randomized Algorithms (3)

- Example: database with parallel processes accessing the database
  - distribute data evenly over N cpus
  - When we need to access object X we need to know which object has it
    - Location(object X) = hash_value(X) % number_of_CPUs
    - No central directory needed to store object locations

Randomized Algorithms (4)

- Example:
  - Data d = database.get_object("monkey");
    - Get_object computes hash of "monkey"
    - Send message to that machine

Randomized Algorithms (5)

- Example 2: replicate input data on all machines
  - When using a dynamic master-slave model:
    - Master thinks that works needs to be done:
      - Creates a new slave, sends it somewhere with a description of the work it is supposed to do
    - Centralized:
      - Master keeps track of everything:
        - » Which CPU is busy / idle
        - » Requires messages to keep master up-to-date
    - Random:
      - Master sends slave to random CPU
        - » No overhead !

Randomized Algorithms (6)

- Imperfect hashing: load imbalance
- Does not work when time per work unit differs greatly