Abstractions for algorithms and parallel machines

Keshav Pingali
University of Texas, Austin

High-level idea

- Difficult to work directly with textual programs
  - Where is the parallelism in the program?
  - Solution: use an abstraction of the program that highlights opportunities for exploiting parallelism
  - What program abstractions are useful?
- Difficult to work directly with a parallel machine
  - Solution: use an abstraction of the machine that exposes features that you want to exploit and hides features you cannot or do not want to exploit
  - What machine abstractions are useful?

Abstractions introduced in lecture

- Program abstraction: computation graph
  - nodes are computations
    - granularity of nodes can range from single operators (+,*), etc. to arbitrarily large computations
  - edges are precedence constraints of some kind
    - edge $a \rightarrow b$ may mean computation $a$ must be performed before computation $b$
  - many variations in the literature
    - imperative languages community: data-dependence graphs, program dependence graphs
    - functional languages community: dataflow graphs
- Machine abstraction: PRAM
  - parallel RAM model
  - exposes parallelism
  - hides synchronization and communication

Computation DAG’s

- DAG with START and END nodes
  - all nodes reachable from START
  - END reachable from all nodes
  - START and END are not essential
- Nodes are computations
  - each computation can be executed by a processor in some number of time-steps
  - computation may require reading/writing shared memory
  - node weight: time taken by a processor
  - $w_i$ is weight of node $i$
- Edges are precedence constraints
  - nodes other than START can be executed only after immediate predecessors in graph have been executed
  - known as dependences
- Very old model
  - PERT charts (late 50’s)
    - Program Evaluation and Review Technique
    - developed by US Navy to manage Polaris submarine contracts

START
\[ \cdots \]
END

1
\[ \cdots \]
P

Computation DAG
Computer model

- P identical processors
- Memory
  - processors have local memory
  - all shared-data is stored in global memory
- How does a processor know which nodes it must execute?
  - work assignment
- How does a processor know when it is safe to execute a node?
  - (eg) if P1 executes node a and P2 executes node b, how does P2 know when P1 is done?
  - synchronization
- For now, let us defer these questions
- In general, time to execute program depends on work assignment
  - for now, assume only that if there is an idle processor and a ready node, that node is assigned immediately to an idle processor
- \( T_P \) = best possible time to execute program on P processors

Work and critical path

- Work = \( \sum w_i \)
  - time required to execute program on one processor
  - \( T_1 \)
- Path weight
  - sum of weights of nodes on path
- Critical path
  - path from START to END that has maximal weight
  - this work must be done sequentially, so you need this much time regardless of how many processors you have
  - call this \( T_\infty \)

Terminology

- Instantaneous parallelism
  \( IP(t) \) = maximum number of processors that can be kept busy at each point in execution of algorithm
- Maximal parallelism
  \( MP \) = highest instantaneous parallelism
- Average parallelism
  \( AP = T_1/T_\infty \)

Computing critical path etc.

- Algorithm for computing earliest start times of nodes
  - Keep a value called minimum-start-time (mst) with each node, initialized to 0
  - Do a topological sort of the DAG
    - ignoring node weights
  - For each node \( n \) (= START) in topological order
    - for each node \( p \) in predecessors(n)
      - \( mst_n = \max(mst_n, mst_p + w_p) \)
  - Complexity = \( O(|V|+|E|) \)
- Critical path and instantaneous, maximal and average parallelism can easily be computed from this
Speed-up

- Speed-up(P) = T₁/Tₚ
  - intuitively, how much faster is it to execute program on P processors than on 1 processor?
- Bound on speed-up
  - regardless of how many processors you have, you need at least T₁/T∞ units of time
  - speed-up(P) ≤ T₁/T∞ = Σ_i w_i /CP = AP

Amdahl’s law

- Amdahl:
  - suppose a fraction p of a program can be done in parallel
  - suppose you have an unbounded number of parallel processors and they operate infinitely fast
  - speed-up will be at most 1/(1-p).
- Follows trivially from previous result.
- Plug in some numbers:
  - p = 90% ⇒ speed-up ≤ 10
  - p = 99% ⇒ speed-up ≤ 100
- To obtain significant speed-up, most of the program must be performed in parallel
  - serial bottlenecks can really hurt you

Scheduling

- Suppose P ≤ MP
- There will be times during the execution when only a subset of “ready” nodes can be executed.
- Time to execute DAG can depend on which subset of P nodes is chosen for execution.
- To understand this better, it is useful to have a more formal model of the machine

PRAM Model

- Parallel Random Access Machine (PRAM)
- Natural extension of RAM model
- Processors operate synchronously (in lock-step)
  - synchronization
- Each processor has private memory

What if we only had 2 processors?
Details

- A PRAM step has three phases
  - read: each processor can read a value from shared-memory
  - compute: each processor can perform a computation on local values
  - write: each processor can write a value to shared-memory
- Variations:
  - Exclusive read, exclusive write (EREW)
    - a location can be read or written by only one processor in each step
  - Concurrent read, exclusive write (CREW)
  - Concurrent read, concurrent write (CRCW)
    - some protocol for deciding result of concurrent writes
- We will use the CREW variation
  - assume that computation graph ensures exclusive writes

Schedules

Schedule 1:

- Function from node to (processor, start time)
- Also known as "space-time mapping"

Schedule 2:

Optimal schedules

- Optimal schedule
  - shortest possible schedule for a given DAG and the given number of processors
- Complexity of finding optimal schedules
  - one of the most studied problems in CS
- DAG is a tree:
  - level-by-level schedule is optimal (Aho, Hopcroft)
- General DAGs
  - variable number of processors (number of processors is input to problem): NP-complete
  - fixed number of processors
    - 2 processors: polynomial time algorithm
    - 3, 4, 5…: complexity is unknown
- Many heuristics available in the literature

Heuristic: list scheduling

- Maintain a list of nodes that are ready to execute
  - all predecessor nodes have completed execution
- Fill in the schedule cycle-by-cycle
  - in each cycle, choose nodes from ready list
  - use heuristics to choose "best" nodes in case you cannot schedule all the ready nodes
- One popular heuristic:
  - assign node priorities before scheduling
  - priority of node n:
    - weight of maximal weight path from n to END
    - intuitively, the "further" a node is from END, the higher its priority
List scheduling algorithm

cycle c = 0;
ready-list = {START};
inflight-list = {};
while (|ready-list|+|inflight-list| > 0) {
  for each node n in ready-list in priority order {
    if (a processor is free at this cycle) {
      remove n from ready-list and add to inflight-list;
      add node to schedule at time cycle;
    }
    else break;
  }
  c = c + 1; //increment time
  for each node n in inflight-list {
    if (n finishes at time cycle) {
      remove n from inflight-list;
      add every ready successor of n in DAG to ready-list
    }
  }
}

Example

```
<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td></td>
<td>a</td>
<td>c</td>
<td>END</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P0</td>
<td>START</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P1</td>
<td>b</td>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Heuristic picks the good schedule
Not always guaranteed to produce optimal schedule
(otherwise we would have a polynomial time algorithm)

Generating computation graphs

- How do we produce computation graphs in the first place?
- Two approaches
  - specify DAG explicitly
    - like parallel programming
    - easy to make mistakes
      - race conditions: two nodes that write to same location but are not ordered by dependence
  - by compiler analysis of sequential programs
- Let us study the second approach
  - called dependence analysis

Data dependence

- Basic blocks
  - straight-line code
- Nodes represent statements
- Edge $S_i \rightarrow S_j$
  - flow dependence (read-after-write (RAW))
    - $S_i$ is executed before $S_j$ in basic block
    - $S_i$ writes to a variable that is read by $S_j$
  - anti-dependence (write-after-read (WAR))
    - $S_i$ is executed before $S_j$ in basic block
    - $S_i$ reads from a variable that is written by $S_j$
  - output-dependence (write-after-write (WAW))
    - $S_i$ is executed before $S_j$ in basic block
    - $S_i$ and $S_j$ write to the same variable
  - input-dependence (read-after-read (RAR)) (usually not important)
    - $S_i$ is executed before $S_j$ in basic block
    - $S_i$ and $S_j$ read from the same variable
Conservative approximation

• In real programs, we often cannot determine precisely whether a dependence exists
  – in example,
    • $i = j$: dependence exists
    • $i \neq j$: dependence does not exist
  – dependence may exist for some invocations and not for others

• Conservative approximation
  – when in doubt, assume dependence exists
  – at the worst, this will prevent us from executing some statements in parallel even if this would be legal

• Aliasing: two program names for the same storage location
  – (e.g.) $X(i)$ and $X(j)$ are may-aliases
  – may-aliasing is the major source of imprecision in dependence analysis

Putting it all together

• Write sequential program.
• Compiler produces parallel code
  – generates control-flow graph
  – produces computation DAG for each basic block by performing dependence analysis
  – generates schedule for each basic block
    • use list scheduling or some other heuristic
    • branch at end of basic block is scheduled on all processors
• Problem:
  – average basic block is fairly small (~ 5 RISC instructions)
• One solution:
  – transform the program to produce bigger basic blocks

One transformation: loop unrolling

• Original program
  for $i = 1, 100$
  $X(i) = i$
• Unroll loop 4 times: not very useful!
  for $i = 1, 100, 4$
  $X(i) = i$
  $i = i + 1$
  $X(i) = i$
  $i = i + 1$
  $X(i) = i$
  $i = i + 1$
  $X(i) = i$

Smarter loop unrolling

• Use new name for loop iteration variable in each unrolled instance
  for $i = 1, 100, 4$
  $X(i) = i$
  $i1 = i + 1$
  $X(i1) = i1$
  $i2 = i + 2$
  $X(i2) = i2$
  $i3 = i + 3$
  $X(i3) = i3$
Array dependence analysis

• If compiler can also figure out that \(X(i), X(i+1), X(i+2),\) and \(X(i+3)\) are different locations, we get the following dependence graph for the loop body

\[
\begin{align*}
\text{for } i &= 1, 100, 4 \\
X(i) &= i \\
i1 &= i + 1 \\
X(i1) &= i1 \\
i2 &= i + 2 \\
X(i2) &= i2 \\
i3 &= i + 3 \\
X(i3) &= i3
\end{align*}
\]

Array dependence analysis (contd.)

• We will study techniques for array dependence analysis later in the course

• Problem can be formulated as an integer linear programming problem:
  – Is there an integer point within a certain polyhedron derived from the loop bounds and the array subscripts?

Limitations

• PRAM model abstracts away too many important details of real parallel machines
  – synchronous model of computing does not scale to large numbers of processors
  – global memory that can be read/written in every cycle by all processors is hard to implement

• DAG model of programs
  – for irregular algorithms, we may not be generate static computation DAG
  – even if we could generate a static computation DAG, latencies of some nodes may be variable on a real machine
  • what is the latency of a load?

• Given all these limitation, why study list scheduling on PRAM’s in so much detail?

Close connection to scheduling instructions for VLIW machines

• Processors ➔ functional units
• Local memories ➔ registers
• Global memory ➔ memory
• Time ➔ instruction
• Nodes in DAG are operations (load/store/add/mul/branch/…)
  – instruction-level parallelism
• List scheduling
  – useful for scheduling code for pipelined, superscalar and VLIW machines
  – used widely in commercial compilers
  – loop unrolling and array dependence analysis are also used widely
Historical note on VLIW processors

- Ideas originated in late 70's-early 80's
- Two key people:
  - Bob Rau (Stanford, UIUC, TRW, Cydrome, HP)
  - Josh Fisher (NYU, Yale, Multiflow, HP)
- Bob Rau's contributions:
  - Transformations for making basic blocks larger:
    - predication
    - Software pipelining
    - Hardware support for these techniques
    - Predicated execution
    - Rotating register files
    - Most of these ideas were later incorporated into the Intel Itanium processor
- Josh Fisher:
  - Transformations for making basic blocks larger:
    - Trace scheduling uses key idea of branch processing
    - Multiflow compiler used loop unrolling

Program dependence graph

- Program dependence graphs (PDGs) (Ferrante, Ottenstein, Warren)
- Data dependences + control dependences
- Intuition for control dependence:
  - Statement s is control-dependent on statement p if the execution of p determines whether n is executed
  - (Eg) statements in the two branches of a conditional are control-dependent on the predicate
- Control dependence is a subtle concept:
  - Formalizing the notion requires the concept of postdominance in control-flow graphs

Control dependence

- Intuitive idea:
  - Node w is control-dependent on a node u if node u determines whether w is executed
- Example:

We would say S1 and S2 are control-dependent on e
Examples (contd.)

We would say node S1 is control-dependent on e. It is also intuitive to say node e is control-dependent on itself: execution of node e determines whether or not e is executed again.

Example (contd.)

S1 and S3 are control-dependent on f.
Are they control-dependent on e?
Decision at e does not fully determine if S1 (or S3 is executed) since there is a later test that determines this.
So we will NOT say that S1 and S3 are control-dependent on e.
Intuition: control-dependence is about “last” decision point.
However, f is control-dependent on e, and S1 and S3 are transitively (iteratively) control-dependent on e.

Example (contd.)

Can a node be control-dependent on more than one node?
- yes, see example
- nested repeat-until loops
  - n is control-dependent on t1 and t2 (why?)
In general, control-dependence relation can be quadratic in size of program.

Example (contd.)

Formal definition of control dependence

Formalizing these intuitions is quite tricky.
Starting around 1980, lots of proposed definitions.
Commonly accepted definition due to Ferrane, Ottenstein, Warren (1987).
Uses idea of postdominance.
We will use a slightly modified definition due to Bilardi and Pingali which is easier to think about and work with.
Control dependence definition

- First cut: given a CFG G, a node \( w \) is control-dependent on an edge \((u \rightarrow v)\) if
  - \( w \) postdominates \( v \)
  - \( \ldots \ldots w \) does not postdominate \( u \)
- Intuitively,
  - first condition: if control flows from \( u \) to \( v \) it is guaranteed that \( w \) will be executed
  - second condition: but from \( u \) we can reach END without encountering \( w \)
  - so there is a decision being made at \( u \) that determines whether \( w \) is executed

Control dependence definition

- Small caveat: what if \( w = u \) in previous definition?
  - See picture: is \( u \) control-dependent on edge \((u \rightarrow v)\)?
  - Intuition says yes, but definition on previous slides says "\( u \) should not postdominate \( u \)" and our definition of postdominance is reflexive
- Fix: given a CFG G, a node \( w \) is control-dependent on an edge \((u \rightarrow v)\) if
  - \( w \) postdominates \( v \)
  - if \( w \) is not \( u \), \( w \) does not postdominate \( u \)

Strict postdominance

- A node \( w \) is said to strictly postdominate a node \( u \) if
  - \( w \neq u \)
  - \( w \) postdominates \( u \)
- That is, strict postdominance is the irreflexive version of the dominance relation
- Control dependence: given a CFG G, a node \( w \) is control-dependent on an edge \((u \rightarrow v)\) if
  - \( w \) postdominates \( v \)
  - \( w \) does not strictly postdominate \( u \)

Example
Computing control-dependence relation

• Nodes control dependent on edge \( (u \to v) \) are nodes on path up the postdominator tree from \( v \) to \( \text{ipdom}(u) \), excluding \( \text{ipdom}(u) \)
  – We will write this as \( [v,\text{ipdom}(u)) \)
  • half-open interval in tree

Computing control-dependence relation

• Compute the postdominator tree
• Overlay each edge \( u \to v \) on pdom tree and determine nodes in interval \( [v,\text{ipdom}(u)) \)
• Time and space complexity is \( O(EV) \).
• Faster solution: in practice, we do not want the full relation, we only make queries
  – \( \text{cd}(e) \): what are the nodes control-dependent on an edge \( e \)?
  – \( \text{conds}(w) \): what are the edges that \( w \) is control-dependent on?
  – \( \text{cdequiv}(w) \): what nodes have the same control-dependences as node \( w \)?
• It is possible to implement a simple data structure that takes \( O(E) \) time and space to build, and that answers these queries in time proportional to output of query (optimal) (Pingali and Bilardi 1997).

Effective abstractions

• Program abstraction is effective if you can write an interpreter for it
• Why is this interesting?
  – reasoning about programs becomes easier if you have an effective abstraction
  – (eg) give a formal Plotkin-style structured operational semantics for the abstraction, and use that to prove properties of execution sequences
• One problem with PDG
  – not clear how to write an interpreter for PDG

Dataflow graphs: an effective abstraction

• From functional languages community
• Functional languages:
  – values and functions from values to values
  – no notion of storage that can be overwritten successively with different values
• Dependence viewpoints:
  – only flow-dependences
  – no anti-dependences or output-dependences
• Dataflow graph:
  – shows how values are used to compute other values
  – no notion of control-flow
  – control-dependence is encoded as data-dependence
  – effective abstraction: interpreter can execute abstraction in parallel
• Major contributors:
  – Jack Dennis (MIT): static dataflow graphs
  – Arvind (MIT): dynamic dataflow graphs
Static Dataflow Graphs

Dataflow

- Execution of an operation is enabled by availability of the required operand values. The completion of one operation makes the resulting values available to the elements of the program whose execution depends on them.

  Dennis

- Execution of an operation must not cause side-effect to preserve determinacy. The effect of an operation must be local.

Dennis' Program Graphs

Operators connected by arcs

Firing Rules: Functional Operators
Firing Rules: T-Gate

The Switch Operator

Firing Rules: Merge

Firing Rules: Merge cont

not ready to fire
Some Conventions

\[ X_1 \times X_2 \equiv X_1 \times X_2 \]

Some Conventions Cont.

\[ X_1 \times X_2 \equiv X_1 \times X_2 \]

Rules To Form Dataflow Graphs: Juxtaposition

Given

\[ \begin{array}{c}
G_1 \\
\vdots \\
G_2 \\
\vdots
\end{array} \]

\[ \begin{array}{c}
\vdots \\
G_1 \\
\vdots \\
\vdots
\end{array} \]

Rules To Form Dataflow Graphs: Iteration

Given

\[ \begin{array}{c}
\vdots \\
G_1 \\
\vdots
\end{array} \]

\[ \begin{array}{c}
\vdots
\end{array} \]

\[ G \]
Example: The Stream Duplicator

1-to-2

The Gate Operator

\[ x \]

C

What happens if we don't use the gate in the Stream Duplicator?

The Stream Halver

2-to-1

The Stream Halver

Throws away every other token.

Translation to dataflow graphs

• \[ \text{fact}(n) = \begin{cases} 1 & \text{if } (n == 1) \text{ then } 1 \\ n \times \text{fact}(n-1) & \text{else} \end{cases} \]
Determinate Graphs

Graphs whose behavior is time independent, i.e., the values of output tokens are uniquely determined by the values of input tokens.

A dataflow graph formed by repeated juxtaposition and iteration of deterministic dataflow operators results in a deterministic graph.

Problem with functional model

- Data structures are values
- No notion of updating elements of data structures
- Think about our examples:
  - How would you do DMR?
  - Can you do event-driven simulation without speculation?

Effective parallel abstractions for imperative languages

- Beck et al: From Control Flow to Dataflow
- Approach:
  - extend dataflow model to include side-effects to memory
  - control dependences are encoded as data-dependences as in standard dataflow model
- Uses:
  - execute imperative languages on dataflow machines (which were being built back in 1990)
  - intermediate language for reasoning operationally about parallelism in imperative languages

Limitations of computation graphs

- For most irregular algorithms, we cannot generate a static computation graph
  - dependences are a function of runtime data values
- Therefore, much of the scheduling technology developed for computation graphs is not useful for irregular algorithms
- Even if we can generate a computation graph, latencies of operations are often unpredictable
- Bottom-line
  - useful to understand what is possible if perfect information about program is available
  - but need heuristics like list-scheduling even in this case!
Summary

• Computation graphs
  – nodes are computations
  – edges are dependences
• Static computation graphs: obtained by
  – studying the algorithm
  – analyzing the program
• Limits on speed-ups
  – critical path
  – Amdahl’s law
• PRAM model
• DAG scheduling for PRAM
  – similar to VLIW code generation problem
  – heuristic: list scheduling (many variations)
• Static computation graphs are useful for regular algorithms, but not very useful for irregular algorithms