CS4961 Parallel Programming

Lecture 5: Data and Task Parallelism, cont.

Mary Hall
September 8, 2009

Administrative

- Homework 2 posted, due September 10 before class
  - Use the "handin" program on the CADE machines
  - Use the following command:

    "handin cs4961 hw2 <prob1file>"
- Mailing list set up: cs4961@list.eng.utah.edu
- Sriram office hours:
  - MEB 3115, Mondays and Wednesdays, 2-3 PM

Homework 2

Problem 1 (#10 in text on p. 111):

The Red/Blue computation simulates two interactive flows. An n x n board is initialized so cells have one of three colors: red, white, and blue, where white is empty, red moves right, and blue moves down. Colors wrap around on the opposite side when reaching the edge.

In the first half step of an iteration, any red color can move right one cell if the cell to the right is unoccupied (white). On the second half step, any blue color can move down one cell if the cell below it is unoccupied. The case where red vacates a cell (first half) and blue moves into it (second half) is okay.

Viewing the board as overlaid with t x t tiles (where t divides n evenly), the computation terminates if any tile's colored squares are more than c% one color. Use Peril-L to write a solution to the Red/Blue computation.

Homework 2, cont.

Problem 2:

For the following task graphs, determine the following:

(1) Maximum degree of concurrency.

(2) Critical path length.

(3) Maximum achievable speedup over one process assuming an arbitrarily large number of processes is available.

(4) The minimum number of processes needed to obtain the maximum possible speedup.

(5) The maximum achievable speedup if the number of processes is limited to (a) 2 and (b) 8.

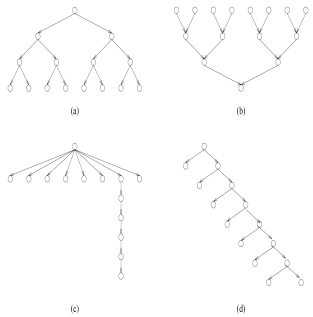


Figure 3.42 Task-dependency graphs for Problem 3.2.

Today's Lecture

- Parallel Scan and Peril-L
- Task Dependence Graphs
- Task Parallel Algorithm Models
- Introduction to SIMD for multimedia extensions (SSE-3 and Altivec)
- Sources for this lecture:
  - Larry Snyder, http://www.cs.washington.edu/education/courses/524/08wi/
  - Grama et al., Introduction to Parallel Computing, http://www.cs.umn.edu/~karypis/parbook
  - JaewookShin http://www-unix.mcs.anl.gov/~jaewook/slides/vectorization-uchicago.ppt
  - Some of the above from "Exploiting Superword Level Parallelism with Multimedia Instruction Sets", Larsen and Amarasinghe (PLDI 2000).

Definitions of Data and Task Parallelism

- Data parallel computation:
  - Perform the same operation to different items of data at the same time; the parallelism grows with the size of the data.
- Task parallel computation:
  - Perform distinct computations -- or tasks -- at the same time; with the number of tasks fixed, the parallelism is not scalable.
- Summary
  - Mostly we will study data parallelism in this class
  - Data parallelism facilitates very high speedups; and scaling to supercomputers.
  - Hybrid (mixing of the two) is increasingly common

Connecting Global and Local Memory

- CTA model does not have a "global memory". Instead, global data is distributed across local memories
- But #1 thing to learn is importance of locality. Want to be able to place data current thread will use in local memory
- Construct for data partitioning/placement

```
localize();
```

- Meaning
  - Return a reference to portion of global data structure allocated locally (not a copy)
  - Thereafter, modifications to local portion using local name do not incur λ penalty

Reductions (and Scans) in Peril-L

- Aggregate operations use APL syntax
  - Reduce: <op>/<operand> for <op> in {+, *, &&, ||, max, min}; as in +/priv_sum
  - Scan: <op>\<operand> for <op> in {+, *, &&, ||, max, min}; as in +\local_finds
- To be portable, use reduce & scan rather than programming them

```
exclusive {count += priv_count; }
```
WRONG

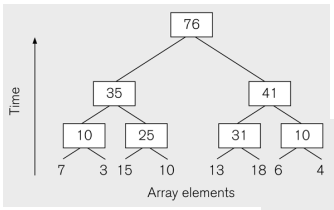
```
count = +/priv_count;
```
RIGHT

Reduce/Scan Imply Synchronization

Scalable Parallel Solution using Localize

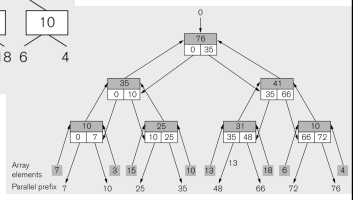
```
1  int array[length];                          The data is global
2  int t;                                      Number of desired threads
3  int total;                                  Result of computation, grand total
4  forall(j in(0..t-1))
5  {
6    int size=mySize(array,0);                 Figure size of local part of global data
7    int myData[size]=localize(array[]);
                                               Associate my part of global data with
                                               local variable
8    int i, priv_count=0;                      Local accumulation
9    for(i=0; i<size; i++)
10   {
11     if(myData[i]==3)
12     {
13       priv_count++;
14     }
15   }
16   total =+/priv_count;                      ompute grand total
17 }
```


Completeness: Scan Operation



Prefix Sum

Compute: $Y_i = \Sigma_{j=0,i} X_j$



Two phases:

Compute sum reduction at intermediate nodes in tree

Propagate prefix downward

Task Dependence Graph (or Task Graph)

- We need an abstraction for understanding the parallelism in a computation.
- Construct a directed graph
  - nodes correspond to tasks
  - edges indicating that the result of one task is required for processing the next.
- With this graph, we can reason about the available parallelism in the computation.
  - For example, ensure that work is equally spread across all processes at any point (minimum idling and optimal load balance).
- Properties of tasks
  - Usually weighted, as tasks may not have the same execution time.
  - Execution time may be unknown.
  - In full generality, may be created dynamically at run time.

Example: Database Query

- Consider processing the query

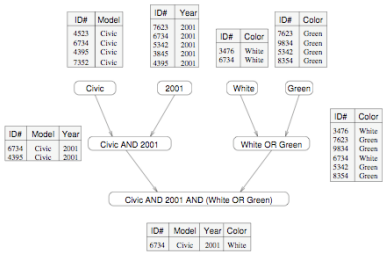
  MODEL = "CIVIC" AND YEAR = 2001 AND (COLOR = "GREEN" OR COLOR = "WHITE)

on the following database:

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

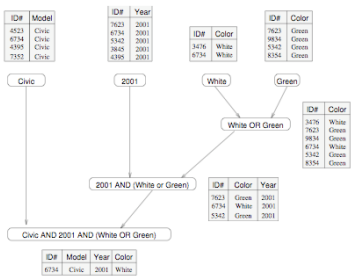
Database Query Task Dependence Graph

- Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.
- Edges in this graph: output of one task is needed to accomplish the next task.



Database Query Task Dependence Graph 2

- Another decomposition of tasks
  - Many different decompositions are possible with different performance properties.



Task Granularity

- Granularity is the amount of work associated with parallel tasks between synchronization/ communication points.
- From Lecture 1, finding the appropriate granularity is one of the key challenges in efficient parallel code
  - The appropriate level of granularity varies by architecture.
  - Too coarse grained: load imbalance, high memory latency, idle processors
  - Too fine grained: overhead dominates execution time

Degree of Concurrency (of a Decomposition)

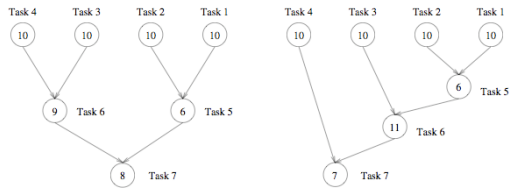
- Definition: The number of tasks that can be executed in parallel.
- Maximum degree of concurrency: The maximum number of tasks that can be executed in parallel at any point during execution.
  - Maximum degree of concurrency of the database query examples?
- Average degree of concurrency: The average number of tasks that can be processed in parallel over the execution of the program.
  - Average degree of concurrency of database queries?
- The degree of concurrency varies with the granularity of the decomposition.

Critical Path Length

- A directed path in the task dependence graph represents a sequence of tasks that must be processed in order to preserve meaning.
- The length of the longest path in a task dependency graph is called the *critical path length*.
- The longest such path determines the *minimum execution time* given sufficient available processes.
- The *average concurrency* is the ratio of total work to critical path length (in units of work). This corresponds to the maximum speedup (ignoring locality effects).

Database Query Examples

- Calculate:
  - Maximum concurrency (4)
  - Critical path length (27&34)
  - Average concurrency (2.33 & 1.88)
  - Minimum execution time (cpl)
  - Maximum speedup (ac)
  - # processors for max speedup (mc)
  - Speedup with 2 processors (1.7, 1.68)



Task Parallel Example Algorithm Models

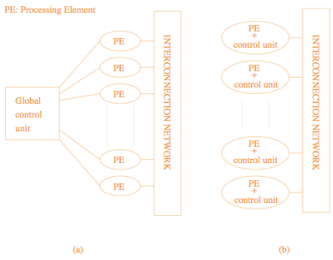
Structuring a parallel algorithm by selecting a decomposition and mapping

- Task Graph Model:
  - Partition a task dependence graph, usually statically.
- Master-Worker Model:
  - One or more processes generate work and allocate it to worker processes. This allocation may be static or dynamic.
- Pipeline / Producer-Consumer Model:
  - A stream of data is passed through a succession of processes, each of which perform some task on it.

Review: Predominant Parallel Control Mechanisms

Name	Meaning	Examples
Single Instruction, Multiple Data (SIMD)	A single thread of control, same computation applied across "vector" elts	Array notation as in Fortran 90: `A[1:n] = A[1:n] + B[1:n]`
Multiple Instruction, Multiple Data (MIMD)	Multiple threads of control, processors periodically synch	Parallel loop: `forall (i=0; i<n; i++)`
Single Program, Multiple Data (SPMD)	Multiple threads of control, but each processor executes same code	Processor-specific code: `if ($myid == 0) { }`

SIMD and MIMD Architectures: What's the Difference?



A typical SIMD architecture (a) and a typical MIMD architecture (b).

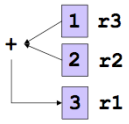
Slide source: Grama et al., Introduction to Parallel Computing, http://www.users.cs.umn.edu/~karypis/parbook

Overview of SIMD Programming

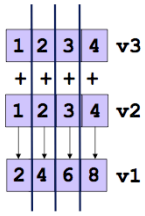
- Vector architectures
- Early examples of SIMD supercomputers
- TODAY Mostly
  - Multimedia extensions such as SSE and AltiVec
  - Graphics and games processors
  - Accelerators (e.g., ClearSpeed)
- Is there a dominant SIMD programming model
  - Unfortunately, NO!!!
- Why not?
  - Vector architectures were programmed by scientists
  - Multimedia extension architectures are programmed by systems programmers (almost assembly language!)
  - GPUs are programmed by games developers (domain-specific libraries)

Scalar vs. SIMD in Multimedia Extensions

```
Scalar: add r1,r2,r3
```

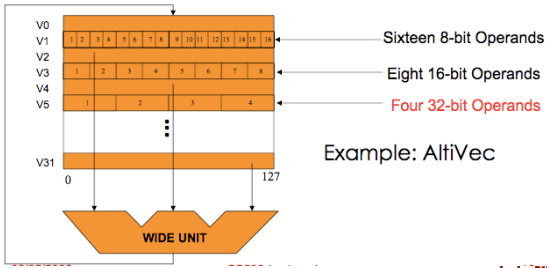



```
SIMD: vadd<sws> v1,v2,v3
```



Multimedia Extension Architectures

- At the core of multimedia extensions
  - SIMD parallelism
  - Variable-sized data fields:
  - Vector length = register width / type size



Example: AltiVec

Why SIMD

+More parallelism

+When parallelism is abundant

+SIMD in addition to ILP

+Simple design

+Replicated functional units

+Small die area

+No heavily ported register files

+Die area: +MAX-2(HP): 0.1% +VIS(Sun): 3.0%

-Must be explicitly exposed to the hardware

-By the compiler or by the programmer

Programming Multimedia Extensions

- Language extension
  - Programming interface similar to function call
  - C: built-in functions, Fortran: intrinsics
  - Most native compilers support their own multimedia extensions
    - GCC: -faltivec, -msse2
    - AltiVec: dst= vec_add(src1, src2);
    - SSE2: dst= _mm_add_ps(src1, src2);
    - BG/L: dst= __fpadd(src1, src2);
    - No Standard !
- Need automatic compilation

Programming Complexity Issues

- High level: Use compiler
  - may not always be successful
- Low level: Use intrinsics or inline assembly tedious and error prone
- Data must be aligned,and adjacent in memory
  - Unaligned data may produce incorrect results
  - May need to copy to get adjacency (overhead)
- Control flow introduces complexity and inefficiency
- Exceptions may be masked

Summary of Lecture

- Peril-L and parallel scan
- Task Parallelism
  - Task dependence graph, critical path, etc.
  - Task-Parallel Algorithm Models
- Introduction to SIMD for multimedia extensions

- Next Time:
  - SIMD for Multimedia Extensions