CS4460 Advanced Algorithms
Batch 08, L4S2

Lecture 11
Multithreaded Algorithms – Part 1

N. H. N. D. de Silva
Dept. of Computer Science & Eng
University of Moratuwa
Announcements

• Last topic discussed is Multithreading
  – Will have two sessions
Outline: Multithreaded Algorithms

• Part 1: today
  – Introduction, Dynamic Multithreading
  – Model for Multithreaded Execution
  – Performance Measures
  – Analyzing Multithreaded Algorithms

• Part 2: next session
  • Parallel Loops
  • Race Conditions
  • Examples
Introduction

• Aim: extend models to parallel algorithms
  – Can run on multiprocessors that permit multiple instructions to execute concurrently
    • E.g.: chip multiprocessors (multicore), clusters, supercomputers

• Multiprocessor models
  – Shared memory vs. distributed memory
  – Shared memory seem to be the trend
  – We use shared memory model for this study
Introduction

• Threading models
  – Static threading
    • Software abstraction of “virtual processors” with shared common memory
    • Programmer can manage (create/destroy) threads with OS support but difficult, error prone, etc
    • Work-load balancing, scheduling by programmer
    • Has lead to concurrency platforms that provide software layers to coordinate, schedule, manage
Introduction

• Threading models
  – Dynamic multithreading
    • Allows programmers to specify parallelism without worrying about load-balancing, managing etc
    • Concurrency platform has a scheduler
    • Simplifies programming
    • Supports nested parallelism and parallel loops
    • We use this model
Introduction

• **Nested Parallelism**
  – Allows a subroutine (procedure) to be *spawned*
    • This allows the *caller* to proceed while the *called* subroutine is computing

• **Parallel Loop**
  – Like a normal (sequential) loop, but the iterations of the loop can execute concurrently
Dynamic Multithreading

• Advantages of the model
  – Programmer specifies only the logical parallelism within a computation
    • Threads within the concurrency platform schedule and load balance within themselves
  – Simply extends the serial programming model
    • Only need 3 keywords: parallel, spawn, sync
    • When these are deleted, results in serial code for the same problem (⇒ serialization of algorithm)
Dynamic Multithreading

• Advantages of the model
  – Clean way to quantify parallelism
    • Based on concepts of “work” and “span”
  – Many multithreaded algorithms involving nested parallelism follow naturally from the divide-and-conquer paradigm
    • Can analyze by solving recurrences also
  – Close to how parallel computing is evolving
Dynamic Multithreading: Basics

• Example: Fibonacci sequence
• Recall: \( F_0 = 0 \) and \( F_1 = 1 \)
  \[ F_i = F_{i-1} + F_{i-2} \quad \text{for} \ i \geq 2 \]
• Simple recursive serial algorithm

\[
\text{FIB}(n) \\
\quad \text{if } n \leq 1 \ \text{return } n \\
\quad \text{else} \quad x \leftarrow \text{FIB}(n-1) \\
\quad \quad \quad y \leftarrow \text{FIB}(n-2) \\
\quad \quad \text{return } x + y
\]
Figure 27.1  The tree of recursive procedure instances when computing \texttt{FIB}(6). Each instance of \texttt{FIB} with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.
Dynamic Multithreading: Basics

• Example: Fibonacci sequence
  – What about the running time of FIB?
  – Running time $T(n) = T(n-1) + T(n-2) + \Theta(1)$
  – Solution to the recurrence is $T(n) = \Theta(F_n)$
  – This can be bound as $T(n) = \Theta(\phi^n)$
    • Where $\phi = (1 + \sqrt{5})/2$ is the golden ratio
    • $T(n)$ grows exponentially in $n$

  – Let us consider a multithreaded algorithm
Dynamic Multithreading: Basics

• Example: Fibonacci sequence

• FIB using dynamic multithreading

\[
P-FIB(n) =
\begin{cases}
  n & \text{if } n \leq 1 \\
  \text{spawn } P-FIB(n-1) + P-FIB(n-2) & \text{else}
\end{cases}
\]

*Nested parallelism* occurs when keyword *spawn* precedes a procedure call.
Dynamic Multithreading: Basics

• Nested parallelism
  – The parent procedure instance may continue instead of waiting for the child to continue
    • E.g., while the child is computing P-FIB(n-1), the parent may go on to compute P-FIB(n-2)
  – The two calls themselves and their children create nested parallelism (they are recursive)
  – But, spawn does not say parent must execute concurrently with spawned children
    • Shows logical parallelism; scheduler will decide
Dynamic Multithreading: Basics

• Nested parallelism
  – A procedure cannot safely use the values returned by a child until it executes a `sync`
    • sync says parent must wait there for all spawned children to complete before proceeding
  
  – Every procedure executes a `sync` implicitly before it returns
  • Ensures all children terminate before it does
Model for Multithreaded Execution

• We can think of a multithreaded computation as a computation dag
  – A vertex: A chain of instructions with no parallel control (a strand)
  – An edge \((u,v)\): strand \(u\) must execute before \(v\)
  – If a strand has 2 successors?
    • One of them must have been spawned
  – A strand with multiple predecessors?
    • Predecessors joined because of a sync
Model for Multithreaded Execution

If a computation dag has a directed path from strand $u$ to strand $v$, then $u$ and $v$ are (logically) in series; else they are (logically) in parallel.

**Figure 27.2** A directed acyclic graph representing the computation of P-Fib(4). Each circle represents one strand, with black circles representing either base cases or the part of the procedure (instance) up to the spawn of P-Fib($n - 1$) in line 3, shaded circles representing the part of the procedure that calls P-Fib($n - 2$) in line 4 up to the `sync` in line 5, where it suspends until the spawn of P-Fib($n - 1$) returns, and white circles representing the part of the procedure after the `sync` where it sums $x$ and $y$ up to the point where it returns the result. Each group of strands belonging to the same procedure is surrounded by a rounded rectangle, lightly shaded for spawned procedures and heavily shaded for called procedures. Spawn edges and call edges point downward, continuation edges point horizontally to the right, and return edges point upward. Assuming that each strand takes unit time, the work equals 17 time units, since there are 17 strands, and the span is 8 time units, since the critical path—shown with shaded edges—contains 8 strands.
Model for Multithreaded Execution

• Multithreaded algorithms are executed on an ideal parallel computer with
  – a set of processors of equal power
  – sequentially consistent shared memory
  • Due to scheduling, instruction ordering may differ from one program run to next, but can assume executed in some order consistent with the computation dag

• We ignore cost of scheduling
Performance Measures

• Two measures: work and span

• Work
  – Total time to execute on one processor
  – i.e., sum of times taken by each strand = the number of vertices in the dag

• Span
  – The longest time to execute strands along any path in the dag
    • If each strand takes unit time, span = # vertices along the critical path
Performance Measures

• E.g., in Fig 27.2
  – Total 17 vertices, 8 vertices in the critical path
  – $\rightarrow$ work = 17 time units, span = 8 time units

• Actual running time also depends on
  – number of processors available
  – how the scheduler allocates strands to proc’s
Performance Measures

• Notations
  – $T_p$ = running time on $P$ processors
  – $T_1$ = the work = running time on one proc.
  – $T_\infty$ = the span = running time if each strand can run on its processor (have infinite proc’s)

– Work, span: lower bounds on running time $T_p$ of a multithreaded computation on $P$ proc’s


Performance Measures

• **Work law**
  – In a step, ideal computer with P proc’s can do at most P units of work. In $T_P$ time, can do at most $P \times T_p$ work. So, $T_p \geq T_1/P$

• **Span law**
  – A P-processor ideal parallel computer cannot run any faster than one with unlimited proc’s. So $T_p \geq T_\infty$
Performance Measures

- **Speedup** = \( T_1 / T_P \)
  - Of a computation on \( P \) processors
  - How many times faster the computation is on \( P \) processors than on 1 processor
  - At most \( P \), i.e., \( T_1 / T_P \leq P \)
  - When \( T_1 / T_P = \Theta(P) \) \( \Rightarrow \) **linear speedup**
  - When \( T_1 / T_P = P \) \( \Rightarrow \) **perfect linear speedup**
Performance Measures

• Parallelism = \( T_1 / T_\infty \)
  – As a ratio: denotes the average amount of work that can be performed in parallel for each step along the critical path
  – As an upper bound: gives the maximum possible speedup on any # of processors
  – Also: Provides a limit on the possibility of attaining perfect linear speedup
    • If # of proc’s exceeds parallelism, cannot possibly achieve perfect linear speedup
Performance Measures

- (Parallel) Slackness = \( \frac{T_1}{T_\infty} \) / P  
  – = \( T_1 / (PT_\infty) \)  
  – The factor by which the parallelism of the computation exceeds the number of processors in the machine  
  • E.g., if slackness < 1, cannot hope to achieve perfect linear speedup  
  – As the slackness decreases from 1 toward 0, speedup of the computation diverges further and further from perfect linear speedup
Scheduling

- Rely on concurrency platform’s scheduler
- A greedy scheduling is good enough

*Theorem 27.1*
On an ideal parallel computer with $P$ processors, a greedy scheduler executes a multithreaded computation with work $T_1$ and span $T_\infty$ in time

$$T_P \leq T_1 / P + T_\infty.$$  \hfill (27.4)
Scheduling

The following corollary to Theorem 27.1 shows that a greedy scheduler always performs well.

**Corollary 27.2**
The running time $T_P$ of any multithreaded computation scheduled by a greedy scheduler on an ideal parallel computer with $P$ processors is within a factor of 2 of optimal.

The next corollary shows that, in fact, a greedy scheduler achieves near-perfect linear speedup on any multithreaded computation as the slackness grows.

**Corollary 27.3**
Let $T_P$ be the running time of a multithreaded computation produced by a greedy scheduler on an ideal parallel computer with $P$ processors, and let $T_1$ and $T_∞$ be the work and span of the computation, respectively. Then, if $P \ll T_1/T_∞$, we have $T_P \approx T_1/P$, or equivalently, a speedup of approximately $P$. 

Nov 2012  
N. H. N. D. de Silva
The \(<\ll\) symbol denotes “much less,” but how much is “much less”? As a rule of thumb, a slackness of at least 10—that is, 10 times more parallelism than processors—generally suffices to achieve good speedup. Then, the span term in the greedy bound, inequality 27.4, is less than 10% of the work-per-processor term, which is good enough for most engineering situations. For example, if a computation runs on only 10 or 100 processors, it doesn’t make sense to value parallelism of, say 1,000,000 over parallelism of 10,000, even with the factor of 100 difference. As Problem 27-2 shows, sometimes by reducing extreme parallelism, we can obtain algorithms that are better with respect to other concerns and which still scale up well on reasonable numbers of processors.
Analyzing Multithreaded Algorithms

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(a)

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

(b)

Figure 27.3  The work and span of composed subcomputations.  (a) When two subcomputations are joined in series, the work of the composition is the sum of their work, and the span of the composition is the sum of their spans.  (b) When two subcomputations are joined in parallel, the work of the composition remains the sum of their work, but the span of the composition is only the maximum of their spans.
Analysis of P-FIB(n)

• The span of P-FIB(n)

\[ T_\infty = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \]
\[ = T_\infty(n-1) + \Theta(1) = \Theta(n) \]

• The parallelism of P-FIB(n)

\[ T_1(n)/ T_\infty(n) = \Theta(\phi^n / n) \]
– Grows dramatically as n gets large
– even on largest parallel computers, modest values of n give near perfect linear speedup
Next Session – Part 2

• Topics
  – Parallel loops
  – Race conditions
  – Examples: Multithreaded matrix-vector and matrix-matrix multiplication

• Before next session, read if possible
  • pp. 785-796 (12 pages) from CLSR 3/e
  • Document “A Minicourse on Dynamic Multithreaded Algorithms” (focus on relevant topics)
References

• The lecture slides are based on the slides prepared by Prof. Sanath Jayasena for this class in previous years.

• Mainly: CLRS book, 3e
  – Part VII: Selected Topics
  – Chapter 27: Multithreaded Algorithms

• Other resources (on LMS)
  – Document “A Minicourse on Dynamic Multithreaded Algorithms”
  – Slide sets: Cilk and Design and Analysis of Dynamic Multithreaded Algorithms and Strassen’s Matrix Multiplication Algorithm
Conclusion

• We discussed multithreaded algorithms
  – Dynamic Multithreading
  – Model for Multithreaded Execution
  – Performance Measures
  – Analyzing Multithreaded Algorithms

• Next session (last session) – Part 2
  – Parallel Loops, Race Conditions, Examples