Multithreaded Algorithms

Bin Wang

School of Software
Tsinghua University

December 27, 2019

Outline

Overview

Parallel computers

- Highest-priced machines:
  Supercomputers
- Intermediate price:
  Clusters built from individual computers
- Inexpensive desktop/laptop:
  Chip multiprocessors

Overview

Parallel computing models

- **shared memory** vs **distributed memory**
- **static threading**
- **concurrency platforms**

Overview

Dynamic multithreaded programming

- **nested parallelism**
- **parallel loops**
- **parallel, spawn, sync and new**
- follow naturally from the divide-and-conquer paradigm
- faithful to how parallel-computing practice is evolving

Overview

Dynamic multithreaded programming

- **Cilk**
  - MiT Cilk, from 1994
  - Cilk++, from 2006
  - Intel Cilk Plus, from 2009, being deprecated in 2018
- **Open MP**, from 1997
- **Task Parallel Library**, from 2010 (.NET Framework 4.0)
- **Threading Building Blocks**, from 2006

Multithreaded Fibonacci Algorithm

The serial algorithm

$\textsc{Fib}(n)$

```
if n ≤ 1
    return n
else x = Fib(n − 1)
    y = Fib(n − 2)
    return x + y
```

Multithreaded Fibonacci Algorithm

The serial algorithm

$\textsc{Fib}(n)$

```
if n ≤ 1
        return n
else x = Fib(n − 1)
        y = Fib(n − 2)
        return x + y
```

$$T(n) = \Theta(\phi^n), \text{where } \phi = (1+\sqrt{5})/2.$$

Multithreaded Fibonacci Algorithm

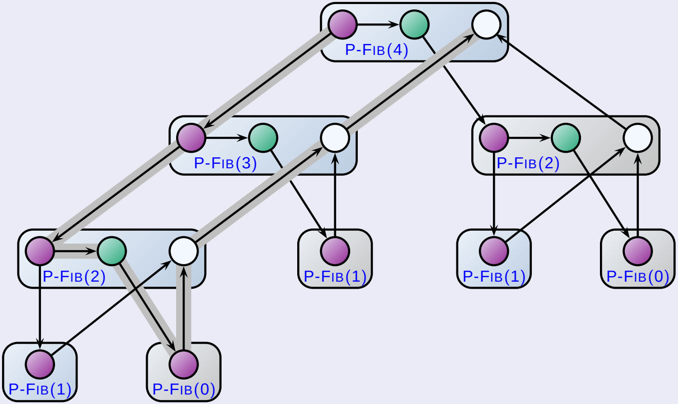
The multithreaded algorithm

P-FIB(n)

```
if n ≤ 1
    return n
else x = spawn P-FIB(n − 1)
    y = P-FIB(n − 2)
    sync
    return x + y
```

Multithreaded Fibonacci Algorithm

A model for multithreaded execution



Multithreaded Fibonacci Algorithm

A model for multithreaded execution

- **Computation dag**
- **Initial strand** and **final strand**
- **Continuation edge**
- **Spawn edge**
- **Call edge**
- **Ideal parallel computer**
- **Sequentially consistent** shared memory

Performance measures

Performance measures

- The ***work*** of a multithreaded computation is the total time to execute the entire computation on one processor. Denote as T_1.

- The ***span*** is the longest time to execute the strands along any path in the dag. Denote as T_∞.

Performance measures

Performance measures

- ***work law:***

$$T_p \geq T_1/P.$$

- ***span law:***

$$T_p \geq T_\infty.$$

- ***speedup :***

$$T_1/T_p.$$

Performance measures

Performance measures

- ***linear speedup :***

$$T_1/T_p = \Theta(P).$$

- ***perfect linear speedup :***

$$T_1/T_p = P.$$

- ***parallelism :***

$$T_1/T_\infty.$$

Performance measures

Performance measures

- ***(parallel) slackness :***

$$(T_1/T_\infty)/P = T_1/(PT_\infty).$$

- As the slackness increases from 1, a good scheduler can achieve closer and closer to perfect linear speedup.

Scheduling

Scheduling

- A multithreaded scheduler must schedule the computation ***on-line***.
- ***Greedy schedulers*** assign as many strands to processors as possible in each time step.
- If at least P strands are ready to execute during a time step, we say that the step is a ***complete step***.

Scheduling

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_∞ in time

$$T_p \leq T_1/P + T_\infty.$$

Scheduling

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_∞ in time

$$T_p \leq T_1/P + T_\infty.$$

Proof.

Suppose for the purpose of contradiction that the number of complete steps is stricly greater than $\lfloor T_1/P \rfloor$.

Scheduling

Proof.

Then, the total work of the complete steps is at least

$$\begin{aligned} P \cdot (\lfloor T_1/P \rfloor + 1) &= P\lfloor T_1/P \rfloor + P \\ &= T_1 - (T_1 \bmod P) + P \\ &> T_1 \end{aligned}$$

Contradiction! So we conclude that the number of complete steps is at most $\lfloor T_1/P \rfloor$.

Scheduling

Proof.

Now we consider an incomplete step. Let G be the dag representing the entire computation.

An incomplete step decreases the span of the unexecuted dag by 1. Hence, the number of incomplete steps is at most T_∞.

Since each step is either complete or incomplete, the theorem follows. □

Scheduling

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.

Proof.

Since the work and span laws give us $T_P^* \geq \max(T_1/P, T_\infty)$, Theorem 27.1 implies that

Scheduling

Proof.

$$\begin{aligned} T_P &\le T_1/P + T_\infty \\ &\le 2 \cdot \max(T_1/P, T_\infty) \\ &\le 2T_P^* \end{aligned}$$

$\square$

Scheduling

Corollary 27.3

If $P \ll T_1/T_\infty$. we have $T_p \approx T_1/P$, or equiva-lenty, a speedup of approximately P.

Proof.

If we suppose that $P \ll T_1/T_\infty$, then we also have $T_\infty \ll T_1/P$, and hence Theorem 27.1 gives us $T_p \leq T_1/P + T_\infty \approx T_1/P$.

Since $T_p \geq T_1/P$, we conclude $T_p \approx T_1/P$. $\square$

Analyzing multithreaded algorithms

P-FIB(n)

- $T_1(n) = T(n) = \Theta(\phi^n)$
-
$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2))) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) \\ &= \Theta(n) \end{aligned}$$
- The parallelism of P-FIB(n) is
$$T_1(n)/T_\infty(n) = \Theta(\phi^n/n).$$

Parallel loops

Multiplying matrix by and vector

$\text{MAT-VEC}(A, x)$

```
n = A.rows
let y be a new vector of length n
parallel for i = 1 to n
        y_i = 0
parallel for i = 1 to n
        for new j = 1 to n
                y_i = y_i + a_ij x_j
return y
```

Parallel loops

Multiplying matrix by and vector

MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')

if $i == i'$
    for $j = 1$ to n
        $y_i = y_i + a_{ij} x_j$
else $mid = \lfloor (i + i')/2 \rfloor$
    spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
    MAT-VEC-MAIN-LOOP$(A, x, y, n, mid + 1, i')$
    sync

Parallel loops

Multiplying matrix by and vector

- $T_1 = \Theta(n^2)$.
- $T_\infty(n) = \Theta(\lg n) + \max_{1 \le i \le n} iter_\infty(i)$.
- The parallel initialization loop in lines $3 - 4$ has span $\Theta(\lg n)$.
- The span of the doubly nested loops in lines $5 - 7$ is $\Theta(n)$.
- The parallelism is $\Theta(n^2)/\Theta(n) = \Theta(n)$.

Race conditions

Race conditions

- A multithreaded algotihm is ***deterministic*** if it always does the same thing on the same input.
- It is ***nondeterministic*** if its behavior might vary from run to run.
- A multithreaded algorithm intended to be deterministic **fails** to be, because it contains a ***"determinacy race"***.

Race conditions

Example

A ***determinacy race*** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

RACE-EXAMPLE()

$x = 0$
parallel for $i = 1$ to 2
    $x = x + 1$
print x

Race conditions

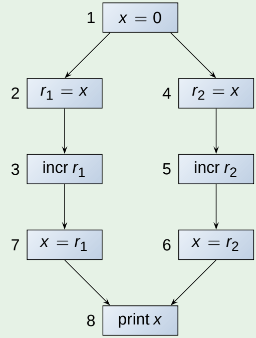
Example

When a processor increments x, the operations is composed of a sequence of instructions:

1. Read x from memory into one of the processor's registers.
2. Increment the value in the register.
3. Write the value in the register back into x in memory.

Race conditions

Example



step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

Race conditions

How to cope with races?

- Mutual-exclusion locks and other methods of synchronization.
- Ensure that strands that operate are independent.
    - In a **parallel for** construct, all the iterations should be independent. Sometimes using the **new** keyword to ensure that different iterations do not operate on the same variable.
    - Between a **spawn** and the corresponding **sync**, the code of the spawned child should be independent of the code of the parent.

Race conditions

Example

$\textsc{Mat-Vec-Wrong}(A, x)$

```
n = A.rows
let y be a new vector of length n
parallel for i = 1 to n
        y_i = 0
parallel for i = 1 to n
        parallel for new j = 1 to n
                y_i = y_i + a_ij x_j
return y
```

A chess lesson

A chess lesson

- The program was prototyped on a $32-$processor computer but was ultimately to run on a supercomputer with 512 processors.
- The original version of the program had work $T_1 = 2048$ seconds and span $T_\infty = 1$ second.

A chess lesson

A chess lesson

- With optimization, the work became $T'_1 = 1024$ seconds and span became $T'_\infty = 8$ seconds.
- $T_{32} = 2048/32 + 1 = 65$ and $T'_{32} = 1024/32 + 8 = 40$
- $T_{512} = 2048/512 + 1 = 5$ and $T'_{512} = 1024/512 + 8 = 10$

Multithreaded matrix multiplication

Example

P-SQUARE-MATRIX-MULTIPLY(A, B)

$n = A.rows$
let C be a new $n \times n$ matrix
parallel for $i = 1$ to n
    parallel for new $j = 1$ to n
        $c_{ij} = 0$
        for new $k = 1$ to n
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

Multithreaded matrix multiplication

Analyzing P-Square-Matrix-Multiply

- $T_1(n) = \Theta(n^3)$
- $T_\infty(n) = \Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n)$
- The parallelism is $\Theta(n^3)/\Theta(n) = \Theta(n^2)$

A divide-and-conquer algorithm

Example

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

$$\begin{aligned} \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{21}B_{21} & A_{22}B_{22} \end{pmatrix} \end{aligned}$$

A divide-and-conquer algorithm

Example

P-MATRIX-MULTIPLY-RE(C, A, B)

$n = A.rows$
if $n == 1$
    $c_{11} = a_{11}b_{11}$
else let T be a new $n \times n$ matrix
    partition A, B, C and T into $n/2 \times n/2$
        submatrices: $A_{11}, A_{12}, A_{21}, A_{22}$;
        $B_{11}, B_{12}, B_{21}, B_{22}$; $C_{11}, C_{12}, C_{21}, C_{22}$;
        and $T_{11}, T_{12}, T_{21}, T_{22}$; respectively
    spawn P-MATRIX-MULTIPLY-RE(C_{11}, A_{11}, B_{11})
    spawn P-MATRIX-MULTIPLY-RE(C_{12}, A_{11}, B_{12})

A divide-and-conquer algorithm

Example

spawn P-MATRIX-MULTIPLY-RE(C_{21}, A_{21}, B_{11})
spawn P-MATRIX-MULTIPLY-RE(C_{22}, A_{21}, B_{12})
spawn P-MATRIX-MULTIPLY-RE(T_{11}, A_{12}, B_{21})
spawn P-MATRIX-MULTIPLY-RE(T_{12}, A_{12}, B_{22})
spawn P-MATRIX-MULTIPLY-RE(T_{21}, A_{22}, B_{21})
P-MATRIX-MULTIPLY-RE(T_{22}, A_{22}, B_{22})
sync
parallel for $i = 1$ to n
    parallel new for $j = 1$ to n
        $c_{ij} = c_{ij} + t_{ij}$

A divide-and-conquer algorithm

Analyzing the running time

- The work

$$M_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$$

- The span

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$$

- Its parallelism is

$$M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$$

Multithreading Strassen's method

Multithreading Strassen's method

[1] Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices. This step takes $\Theta(1)$ work and span by index calculation.

[2] Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\lg n)$ span by using doubly nested **parallel for** loops.

Multithreading Strassen's method

Multithreading Strassen's method

[3] Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively spawn the computation of seven $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$.

[4] Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices, once again using doubly nested **parallel for** loops.

A naive multithreaded merge sort

Example

$\textsc{Merge-Sort}'(A, p, r)$

if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    spawn $\textsc{Merge-Sort}'(A, p, q)$
    $\textsc{Merge-Sort}'(A, q + 1, r)$
    sync
    $\textsc{Merge}(A, p, q, r)$

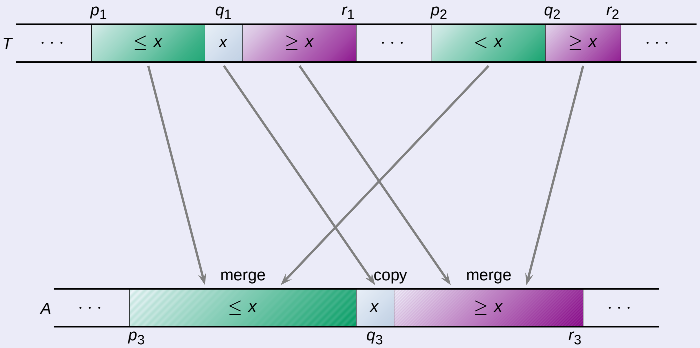
A naive multithreaded merge sort

Analyzing the running time

- The work
$$MS'_1(n) = 2MS'_1(n/2) + \Theta(n) = \Theta(n \lg n)$$
- The span
$$MS'_\infty(n) = MS'_\infty(n/2) + \Theta(n) = \Theta(n)$$
- Its parallelism is
$$MS'_1(n)/MS_\infty(n) = \Theta(\lg n)$$

Multithreaded merging

Parallel Merging



Multithreaded merging

Parallel Merging

P-MERGE$(T, p_1, r_1, p_2, r_2, A, p_3)$

$n_1 = r_1 - p_1 + 1$
$n_2 = r_2 - p_2 + 1$
if $n_1 < n_2$
    exchange p_1 with p_2
    exchange r_1 with r_2
    exchange n_1 with n_2
if $n_1 == 0$
    return

Multithreaded merging

Parallel Merging

else $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
    $q_2 = \textsc{Binary-Search}(T[q_1], T, p_2, r_2)$
    $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
    $A[q_3] = T[q_1]$
    spawn $\textsc{P-Merge}(T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3)$
    $\textsc{P-Merge}(T, q_1 + 1, r_1, q_2, r_2, A, q_3 + 1)$
    sync

Multithreaded merging

Parallel Merging

BINARY-SEARCH(x, T, p, r)

```
low = p
high = max(p, r + 1)
while low < high
    mid = ⌊(low + high)/2⌋
    if x ≤ T[mid]
        high = mid
    else low = mid + 1
return high
```

Multithreaded merging

Analysis of multithreaded merging

$$
\begin{aligned}
\lfloor n_1/2 \rfloor + n_2 &\le n_1/2 + n_2/2 + n_2/2 \\
&= (n_1 + n_2)/2 + n_2/2 \\
&\le n/2 + n/4 \\
&= 3n/4
\end{aligned}
$$

$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\lg n) = \Theta(\lg^2 n)$$

Multithreaded merging

Analysis of multithreaded merging

- $PM_1(n) = \Omega(n)$
- $PM_1(n) = PM_1(\alpha n) + PM_1((1-\alpha)n) + O(\lg n)$
- We prove that $PM_1 = O(n)$ via the substitution method.
- The parallelism of P-MERGE is $PM_1(n)/PM_\infty(n) = \Theta(n/\lg^2 n)$

Multithreaded merge sort

Multithreaded merge sort

```
P-MERGE-SORT(A, p, r, B, s)
n = r − p + 1
if n == 1
    B[s] = A[p]
else let T[1 . . . n] be a new array
    q = ⌊(p + r)/2⌋
    q′ = q − p + 1
    spawn P-MERGE-SORT(A, p, q, T, 1)
    P-MERGE-SORT(A, q + 1, r, T, q′ + 1)
    sync
    P-MERGE(T, 1, q′, q′ + 1, n, B, s)
```

Multithreaded merge sort

Analysis of multithreaded merge sort

- The work

$$PMS_1(n) = 2PMS_1(n/2) + \Theta(n) = \Theta(n \lg n)$$

- The span

$$\begin{aligned} PMS_\infty(n) &= PMS_\infty(n/2) + \Theta(\lg^2 n) \\ &= \Theta(\lg^3 n) \end{aligned}$$

- Its parallelism is

$$PMS_1(n)/PMS_\infty(n) = \Theta(n/\lg^2 n)$$