Analysis of Multithreaded Algorithms

Marc Moreno Maza
University of Western Ontario, London, Ontario (Canada)

CS4402-9535

Plan

1. Matrix Multiplication
2. Merge Sort
3. Tableau Construction

Matrix Multiplication

We will study three approaches:

- a naive and iterative one
- a divide-and-conquer one
- a divide-and-conquer one with memory management consideration
Matrix Multiplication

### Naive iterative matrix multiplication

cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<n; ++j) {
        for (int k=0; k<n; ++k {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

- **Work:** $\Theta(n^3)$
- **Span:** $\Theta(n)$
- **Parallelism:** $\Theta(n^2)$

Matrix multiplication based on block decomposition

\[
\begin{pmatrix}
    C_{11} & C_{12} \\
    C_{21} & C_{22}
\end{pmatrix} = \begin{pmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{pmatrix} \cdot \begin{pmatrix}
    B_{11} & B_{12} \\
    B_{21} & B_{22}
\end{pmatrix}
= \begin{pmatrix}
    A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
    A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
\]

The divide-and-conquer approach is simply the one based on blocking, presented in the first lecture.

\[
// C <- C + A * B
void MMult(T *C, T *A, T *B, int n, int size) {
    T *D = new T[n*n];
    cilk_spawn MMult(C11, A11, B11, n/2, size);
    cilk_spawn MMult(C12, A11, B12, n/2, size);
    cilk_spawn MMult(C22, A21, B12, n/2, size);
    cilk_spawn MMult(C21, A21, B11, n/2, size);
    cilk_spawn MMult(D11, A12, B21, n/2, size);
    cilk_spawn MMult(D12, A12, B22, n/2, size);
    cilk_spawn MMult(D22, A22, B22, n/2, size);
    cilk_sync;
    MAdd(C, D, n, size); // C += D;
    delete[] D;
}
\]
Divide-and-conquer matrix multiplication

```cpp
void MMult(T *C, T *A, T *B, int n, int size) {
    T *D = new T[n*n];
    //base case & partition matrices
    cilk_spawn MMult(C11, A11, B11, n/2, size);
    cilk_spawn MMult(C12, A11, B12, n/2, size);
    cilk_spawn MMult(C21, A21, B12, n/2, size);
    cilk_spawn MMult(C22, A21, B22, n/2, size);
    cilk_spawn MMult(D11, A12, B21, n/2, size);
    cilk_spawn MMult(D12, A12, B22, n/2, size);
    cilk_spawn MMult(D22, A22, B22, n/2, size);
    MMult2(C21, A22, B21, n/2, size);
    cilk_sync; MAdd(C, D, n, size); // C += D;
    delete[] D; }
```

\[ A_p(n) \text{ and } M_p(n): \text{ times on } p \text{ proc. for } n \times n \text{ ADD and MULT.} \]

\[ A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2) \]

\[ A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\log n) \]

\[ M_1(n) = 8M_1(n/2) + A_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3) \]

\[ M_\infty(n) = M_\infty(n/2) + \Theta(\log n) = \Theta(\log^2 n) \]

\[ M_1(n)/M_\infty(n) = \Theta(n^3/\log^2 n) \]

**Work** ? **Span** ? **Parallelism** ?

Divide-and-conquer matrix multiplication: No temporaries!

Plan

1. **Matrix Multiplication**
2. **Merge Sort**
3. **Tableau Construction**

Besides, saving space often saves time due to hierarchical memory.
Merging two sorted arrays

```c
void Merge(T *C, T *A, T *B, int na, int nb) {
    while (na>0 && nb>0) {
        if (*A <= *B) {
            *C++ = *A++; na--;
        } else {
            *C++ = *B++; nb--;
        }
    }
    while (na>0) {
        *C++ = *A++; na--;
    }
    while (nb>0) {
        *C++ = *B++; nb--;
    }
}
```

Time for merging $n$ elements is $\Theta(n)$.

Parallel merge sort with serial merge

```c
template <typename T>
void MergeSort(T *B, T *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        T* C[n];
        cilk_spawn MergeSort(C, A, n/2);
        MergeSort(C+n/2, A+n/2, n-n/2);
        cilk_sync;
        Merge(B, C, C+n/2, n/2, n-n/2);
    }
}
```

- $T_1(n) = 2T_1(n/2) + \Theta(n)$ thus $T_1(n) = \Theta(n \log n)$.
- $T_\infty(n) = T_\infty(n/2) + \Theta(n)$ thus $T_\infty(n) = \Theta(n)$.
- $T_1(n)/T_\infty(n) = \Theta(\log n)$. **Puny parallelism!**
- We need to parallelize the merge!

● **Work?**
● **Span?**
Idea: if the total number of elements to be sorted in $n = n_a + n_b$ then the maximum number of elements in any of the two merges is at most $3n/4$.

![Diagram of parallel merge]

template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
    if (na < nb) {
        P_Merge(C, B, A, nb, na);
    } else if (na==0) {
        return;
    } else {
        int ma = na/2;
        int mb = BinarySearch(A[ma], B, nb);
        C[ma+mb] = A[ma];
        cilk_spawn P_Merge(C, A, B, ma, mb);
        P_Merge(C+ma+mb+1, A+ma+1, B+mb, na-ma-1, nb-mb);
        cilk_sync;
    }
}

Let $PM_p(n)$ be the $p$-processor running time of P-Merge.

In the worst case, the span of P-Merge is

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

The worst-case work of P-Merge satisfies the recurrence

$$PM_1(n) \leq PM_1(\alpha n) + PM_1((1-\alpha)n) + \Theta(\lg n)$$

Recall $PM_1(n) \leq PM_1(\alpha n) + PM_1((1-\alpha)n) + \Theta(\lg n)$ for some $1/4 \leq \alpha \leq 3/4$.

To solve this hairy equation we use the substitution method.

We assume there exist some constants $a, b > 0$ such that $PM_1(n) \leq an - b\lg n$ holds for all $1/4 \leq \alpha \leq 3/4$.

After substitution, this hypothesis implies:

$PM_1(n) \leq an - b\lg n - b\lg n + \Theta(\lg n)$.

We can pick $b$ large enough such that we have $PM_1(n) \leq an - b\lg n$ for all $1/4 \leq \alpha \leq 3/4$ and all $n > 1$.

Then pick $a$ large enough to satisfy the base conditions.

Finally we have $PM_1(n) = \Theta(n)$.
Parallel merge sort with parallel merge

```cpp
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        T C[n];
        cilk_spawn P_MergeSort(C, A, n/2);
        P_MergeSort(C+n/2, A+n/2, n-n/2);
        cilk_sync;
        P_Merge(B, C, C+n/2, n/2, n-n/2);
    }
}
```

- The work satisfies $T_1(n) = 2T_1(n/2) + \Theta(n)$ (as usual) and we have $T_1(n) = \Theta(n \log n)$.
- The worst case critical-path length of the Merge-Sort now satisfies $T_\infty(n) = T_\infty(n/2) + \Theta(\log^2 n) = \Theta(\log^3 n)$.
- The parallelism is now $\Theta(n \log n)/\Theta(\log^3 n) = \Theta(n/\log^2 n)$.

Constructing a tableau $A$ satisfying a relation of the form:

$$A[i,j] = R(A[i-1,j], A[i-1,j-1], A[i,j-1]).$$

(1)

The work is $\Theta(n^2)$. 

(Moreno Maza) Analysis of Multithreaded Algorithms CS4402-9535 21 / 27
Recursive construction

Parallel code

- $T_1(n) = 4T_1(n/2) + \Theta(1)$, thus $T_1(n) = \Theta(n^2)$.
- $T_\infty(n) = 3T_\infty(n/2) + \Theta(1)$, thus $T_\infty(n) = \Theta(n\log_2^3)$.
- **Parallelism**: $\Theta(n^{2-\log_2^3}) = \Omega(n^{0.41})$.

A more parallel construction

- $T_1(n) = 9T_1(n/3) + \Theta(1)$, thus $T_1(n) = \Theta(n^2)$.
- $T_\infty(n) = 5T_\infty(n/3) + \Theta(1)$, thus $T_\infty(n) = \Theta(n^{\log_3^5})$.
- **Parallelism**: $\Theta(n^{2-\log_3^5}) = \Omega(n^{0.53})$.
- This nine-way d-n-c has more parallelism than the four way but exhibits more cache complexity (more on this later).

Acknowledgements

- Charles E. Leiserson (MIT) for providing me with the sources of its lecture notes.
- Matteo Frigo (Intel) for supporting the work of my team with Cilk++ and offering us the next lecture.
- Yuzhen Xie (UWO) for helping me with the images used in these slides.
- Liyun Li (UWO) for generating the experimental data.