Analysis of Multithreaded Algorithms

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS4402-9535

Plan

Plan

Matrix multiplication

$$\underbrace{\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix}}_{C} = \underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}}_{B}$$

We will study three approaches:

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

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**: ?
- **Span**: ?
- **Parallelism**: ?

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

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

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

Divide-and-conquer matrix multiplication

```
// C <- C + A * B
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(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);
             MMult(D21, A22, B21, n/2, size);
  cilk_sync;
  MAdd(C, D, n, size); // C += D;
  delete[] D;
}
```

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

Divide-and-conquer matrix multiplication

```
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(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);
             MMult(D21, A22, B21, n/2, size);
  cilk_sync; MAdd(C, D, n, size); // C += D;
  delete[] D;  }
```

- $A_p(n)$ and $M_p(n)$: times on p proc. for $n \times n$ ADD and MULT.
- $A_1(n) = 4A_1(n/2) + \Theta(1) = \Theta(n^2)$
- $A_\infty(n) = A_\infty(n/2) + \Theta(1) = \Theta(\lg n)$
- $M_1(n) = 8M_1(n/2) + A_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$
- $M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$
- $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$

Divide-and-conquer matrix multiplication: No temporaries!

```
template <typename T>
void MMult2(T *C, T *A, T *B, int n, int size) {
  //base case & partition matrices
  cilk_spawn MMult2(C11, A11, B11, n/2, size);
  cilk_spawn MMult2(C12, A11, B12, n/2, size);
  cilk_spawn MMult2(C22, A21, B12, n/2, size);
             MMult2(C21, A21, B11, n/2, size);
  cilk_sync;
  cilk_spawn MMult2(C11, A12, B21, n/2, size);
  cilk_spawn MMult2(C12, A12, B22, n/2, size);
  cilk_spawn MMult2(C22, A22, B22, n/2, size);
             MMult2(C21, A22, B21, n/2, size);
  cilk_sync;  }
```

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

Divide-and-conquer matrix multiplication: No temporaries!

```
template <typename T>
void MMult2(T *C, T *A, T *B, int n, int size) {
  //base case & partition matrices
  cilk_spawn MMult2(C11, A11, B11, n/2, size);
  cilk_spawn MMult2(C12, A11, B12, n/2, size);
  cilk_spawn MMult2(C22, A21, B12, n/2, size);
             MMult2(C21, A21, B11, n/2, size);
  cilk_sync;
  cilk_spawn MMult2(C11, A12, B21, n/2, size);
  cilk_spawn MMult2(C12, A12, B22, n/2, size);
  cilk_spawn MMult2(C22, A22, B22, n/2, size);
             MMult2(C21, A22, B21, n/2, size);
  cilk_sync;  }
```

- $MA_p(n)$: time on p proc. for $n \times n$ MULT-ADD.
- $MA_1(n) = \Theta(n^3)$
- $MA_\infty(n) = 2MA_\infty(n/2) + \Theta(1) = \Theta(n)$
- $MA_1(n)/MA_\infty(n) = \Theta(n^2)$
- Besides, saving space often saves time due to hierarchical memory.

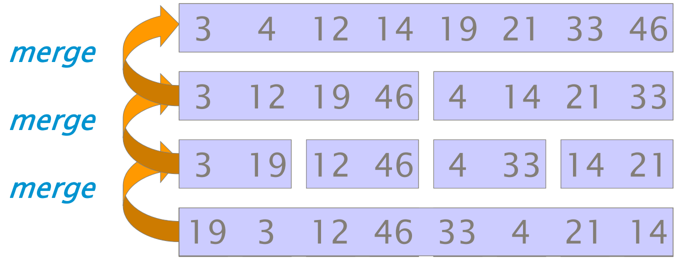
Plan

Merging two sorted arrays

```
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 $\Theta(n)$.

Merge sort



Parallel merge sort with serial merge

```
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);
  }
```

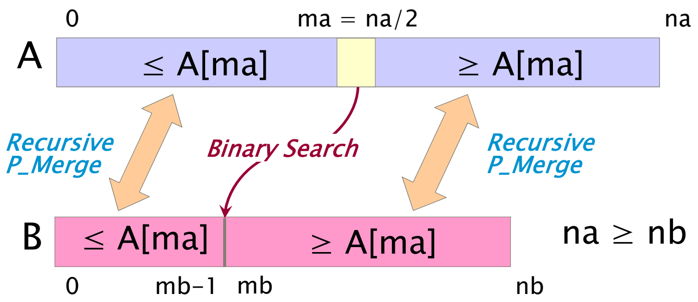
- **Work**?
- **Span**?

Parallel merge sort with serial merge

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

Parallel merge



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 $3n/4$.

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;
  }
}
```

- One should coarse the base case for efficiency.
- **Work**? **Span**?

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

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

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

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

Analyzing parallel merge

- 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

```
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);
  }
}
```

- **Work**?
- **Span**?

Parallel merge sort with parallel merge

```
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

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

.

- The parallelism is now $\Theta(n \lg n)/\Theta(\lg^3 n) = \Theta(n/\lg^2 n)$.

Plan

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

Tableau construction

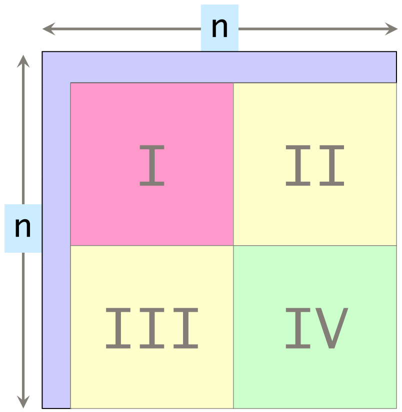
00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Constructing a tableau A satisfying a relation of the form:

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

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

Recursive construction

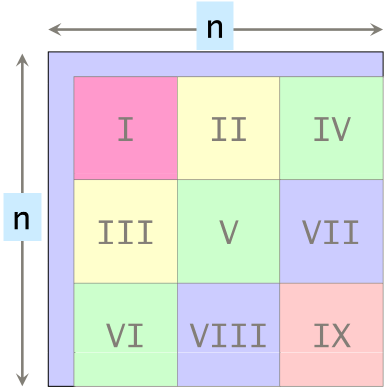


Parallel code

```
I;
cilk_spawn II;
III;
cilk_sync;
IV;
```

- $T_1(n) = 4T_1(n/2) + \Theta(1)$, 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




```
I;
cilk_spawn II;
III;
cilk_sync;
cilk_spawn IV;
cilk_spawn V;
VI;
cilk_sync;
cilk_spawn VII;
VIII;
cilk_sync;
IX;
```

- $T_1(n) = 9T_1(n/3) + \Theta(1)$, 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.