HUST-USYD Summer School on Parallel Programming Practice – Lecture 5

Bing Bing Zhou (bing.zhou@sydney.edu.au)

School of Computer Science, University of Sydney

THE UNIVERSITY OF SYDNEY

Outline

- Synchronization
  - Race condition and mutual exclusion
- OpenMP synchronization constructs
  - Critical, atomic and barrier
- Data dependency
  - Instruction-level data dependency
  - Loop-carried data dependency
- Reduction operation
- OpenMP reduction clause
- Exercise: reduction operation
- Scan (or prefix) operation
- Homework 3

Synchronization

- In shared-memory machines global data are shared by multiple threads
- Synchronization must be used to
  - Protect access to shared data and so to prevent race conditions
  - Impose order constraints
- OpenMP supported synchronization constructs
  - Critical, atomic and barrier
  - The full OpenMP specification has several more

Synchronization Constructs

- OpenMP critical directive:
- #pragma omp critical
- Threads wait their turn – only one at a time can execute in the critical construct
- Critical directive must be inside a parallel region
- E.g., threads to increment count

```
#pragma omp parallel shared(count)
{
   ...
  #pragma omp critical
  count++;
   ...
}
```

Synchronization Constructs

- OpenMP atomic directive:
- #pragma omp atomic
- Also provides mutual exclusion, but only applies to the update of one memory location
- atomic directive applies only to the statement immediately following it
- E.g., accumulate partial results done by multiple threads

```
double sum = 0.0;
#pragma omp parallel shared(sum)
{
   double lsum;
   …
  #pragma omp atomic
  sum += lsum
   …
}
```

Synchronization Constructs

- OpenMP barrier directive:
- #pragma omp barrier
- barrier is a point in a program all threads must reach before any threads are allowed to proceed
- In OpenMP most constructs have an implied barrier at the end of the constructs
  - To remove that barrier, use onwait clause

```
#pragma omp parallel shared(sum)
{
   //all threads do something
  #pragma omp barrier //Threads wait until all threads hit the barrier
   //all threads do some other thing
}
```

Instruction-Level Data Dependency

- Data dependency at the instruction level – the dependency in a sequence of instructions:
  - Flow dependency (Read After Write or RAW)
    - E.g., a = b + c (Write b+c to a)
      d = a * e (Read a out and assign a*e to d)
  - Anti-dependency (WAR)
    - E.g., b = a + c (Read a out and assign a+c to b)
      a = d * e (Write d+e to a)
  - Output dependency (WAW)
    - E.g., a = b + c
      a = d * e

Instruction-Level Data Dependency

- Data dependency at the instruction level – the dependency in a sequence of instructions:
  - Flow dependency is true dependency
  - Anti-dependency and output dependency are not real (so called artificial) dependency and can be eliminated
    - E.g., output dependency in the previous example can be removed by writing d*e to another variable and then use the new variable (instead of a) in the future

Loop-Carried Data Dependency

- Can we parallelize the following for loop?

```
for (i=0; i<n; i++)
{
    tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
}
```

- Clearly there are data dependencies in the three instructions
- However, there are no data dependencies to be carried out to the next iteration
- Therefore, the loop can be safely parallelized
- In OpenMP we really need to check loop-carried dependency!

Loop-Carried Data Dependency

– Can we parallelize the following for loop?

```
for (i=0; i<n; i++)
{
    a[i] = d[i] +e[i];
    d[i] = e * a[i+1];
}
```

WAR dependency

```
a[0] = d[0]+e[0]
d[0] = e * a[1]
a[1] = d[1]+e[1]
d[1] = e * a[2]
a[2] = d[2]+e[2]
d[2] = 3 * a[3]
a[3] = d[3]+e[3]
d[3] = e * a[4]
   ⋮
```

– Not very clear?

– Let's expand the for loop starting from i=0

  – a[i] in red means assigned new value

  – a[i] in red means old value

Loop-Carried Data Dependency

- Now consider parallelize the for loop:

Thread 0:	Thread 1:	Thread 2:
a[0] = d[0]+e[0]	a[2] = d[2]+e[2]	a[4] = d[0]+e[0]
d[0] = e * a[1]	d[2] = 3 * a[3]	d[0] = e * a[5]
a[1] = d[1]+e[1]	a[3] = d[3]+e[3]	a[5] = d[1]+e[1]
d[1] = e * a[2]	d[3] = e * a[4]	d[1] = e * a[6]

Wrong values!

- Now it becomes clear – there are indeed loop-carried data dependencies
- Also potential data race
  - i.e., one thread write and another read on the same data
- How to deal with this problem?

Loop-Carried Data Dependency

```
for (i=0; i<n; i++)
{
    a[i] = d[i] +e[i];
    d[i] = e * a[i+1];
}
```

```
for (i=0; i<n; i++)
{
    d[i] = e * a[i+1];
    a[i] = d[i] +e[i];
}
```

```
for (i=0; i<n; i++)
   d[i] = e * a[i+1];
for (i=0; i<n; i++)
   a[i] = d[i] +e[i];
```

- In the same i iteration, the order of two instructions can be exchanged because there is no data dependency
- Also separate the instructions into two for loops

Loop-Carried Data Dependency

– Now parallelize two for loops one by one:

```
#pragma omp parallel for
for (i=0; i<n; i++)
    d[i] = e * a[i+1];
#pragma omp parallel for
for (i=0; i<n; i++)
    a[i] = d[i] +e[i];
```

Implied barrier here

Thread 0:	Thread 1:	Thread 3:
d[0] = e * a[1]	d[2] = 3 * a[3]	d[4] = e * a[5]
d[1] = e * a[2]	d[3] = e * a[4]	d[5] = e * a[6]
a[0] = d[0]+e[0]	a[2] = d[2]+e[2]	a[4] = d[0]+e[0]
a[1] = d[1]+e[1]	a[3] = d[3]+e[3]	a[5] = d[1]+e[1]

– We now obtain the correct results!

Loop-Carried Data Dependency

- Can we parallelize the following for loop?

```
for (i=0; i<n; i++)
{
    a[i+1] = d[i] +e[i];
    d[i] = e * a[i];
}
```

- A simple problem for you to work out at home

Loop-Carried Data Dependency

– In the previous lecture we also discussed how to deal with another type of loop-carried data dependency:

```
int i, j, A[MAX];
 j = 5;
 for (i=0;i< MAX; i++){
     j +=2;
     A[i] = big(j);
}
```

Note: loop index "i" is private by default

```
int i,  A[MAX];
#pragma omp parallel for
 for (i=0;i< MAX; i++) {
     int j = 5 + 2*(i+1);
     A[i] = big(j);
}
```

Remove loop carried dependence

– In general we need to pay attention on the following loop-carried data dependency:
  – index of an array data is not equal to current loop index
  – The value of a variable changes with the iterations

Reduction Operation

- Given associative operator $\bigoplus$
- Examples
  - Add (+)
  - Multiply (*)
  - And, Or (&&, ||)
  - max, min
- and also an array of elements $[a_0, a_1, a_2, \ldots, a_{n-1}]$
- Calculate
- S = $a_0 \bigoplus a_1 \bigoplus a_2 \bigoplus \ldots \bigoplus a_{n-1}$

Reduction Operation

- E.g., let $\oplus = +$, that is, summation of n numbers:
  - $S = a_0 + a_1 + a_2 + \ldots + a_{n-1} = \sum_{i=0}^{n-1} a_i$
- Use recursion:
  - $S_0 = a_0$
  - $S_i = S_{i-1} + a_i$ for $1 \le i < n$
- A sequential routine:

```
S = a[0];
for (i=1; i<n; i++)
   S = S + a[i];
```

- Can we simply use OpenMP for directive to parallelize this sequential routine?
  - No, it is recursive and the value of S changes with iterations
    - There are loop-carried data dependencies!

Reduction Operation

- E.g., let $\bigoplus = +$, that is, summation of n numbers:
  - $S = a_0 + a_1 + a_2 + \ldots + a_{n-1} = \sum_{i=0}^{n-1} a_i$

- However, addition is associative and commutative
  - We can add the numbers in any order
  - E.g., $S = \sum_{i=0}^{k-1} a_i + \sum_{i=k}^{2k-1} a_i + \sum_{i=2k}^{n-1} a_i$
    $= S_0 + S_1 + S_2$
- Then we can use the OpenMP directive to parallelize the for loop for each thread to compute a partial sum and then accumulate them to obtain the final result

Reduction Operation

- E.g., let $\oplus = +$, that is, summation of n numbers:
  - $S = a_0 + a_1 + a_2 + \ldots + a_{n-1} = \sum_{i=0}^{n-1} a_i$

```
#pragma omp parallel shared(a,S) private(i)
{
  double lsum = 0.0; //each thread has a local lsum variable
  #pragma omp for nowait
  for (i=0; i<n; i++) //the number of iterations distributed
      lsum +=  a[i];    //each thread calculate a partial sum
  ··· //accumulate partial sums
}
```

- How to accumulate the partial sums?

OpenMP Reduction Clause

- OpenMP supports a reduction clause:
  - reduction (op : list)
- E.g.,

```
double S=0;
#pragma omp parallel for shared(S, a), reduction(+:S)
for (int i=0; i<n; ++)
   S += a[i];
```

- When reduction clause is added
  - A local copy of each list variable is made and initialized, depending on the “op” (e.g. 0 for “+”)
  - Updates occur on the local copy
  - Local copies are then reduced into a single value
  - The variables in “list” must be shared in the enclosing parallel region

OpenMP Reduction Clause

– Many different associative operands can be used
– Initial values are the ones that make sense mathematically

Operator	Initial value	Operator	Initial value
+	0	&&	1
*	1	||	0
-	0	&	~0
min	Largest pos. number	|	0
max	Most neg. number	^	0

Lab Exercise: Reduction Operation

- Write an OpenMP program to compute summation of n numbers
- You need to write two routines:
  - One uses OpenMP reduction clause
  - The other is to let each thread compute a partial sum and then accumulate the partial sums
- Run your program to compare the performance of these two routines

Scan Operation

- Let $A = [a_0, a_1, a_2, \ldots, a_{n-1}]$ be an array of elements and $\oplus$ an associative operator
- Scan (or prefix) operation produces another array C:

  $C = [a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \cdots, (a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1})]$

- E.g., $\mathrm{A} = [1,2,3,4,5,6]$, and after add scan of A we obtain $C = [1,3,6,10,15,21]$
- It can be seen that there are many redundancy in calculating c_i
  - Use recursion to remove redundant operations
- A sequential routine:

```
c[0] = a[0];
for (i=1; i<n; i++)
    c[i] = c[i-1] ⊕ a[i];
```

- It takes only $n - 1$ $\oplus$ operations

Parallel Algorithm for Scan Operation

- The sequential routine is very simple, but strictly sequential in nature
  - True data dependency and cannot directly use OpenMP for directive to parallelize the for loop
- Need to design new parallel algorithms
- Many parallel algorithms for scan operation have been developed based on the fact that $\oplus$ is an associative operator
  - Key is to minimize the amount of operations
- In the following we discuss one parallel algorithm which is efficient for parallel computation of scan operation on shared-memory machines

Parallel Algorithm for Scan Operation

– Rewrite the c_i:

$$
\begin{aligned}
c_0 &= a_0 \\
c_1 &= a_0 \oplus a_1 \\
c_2 &= a_0 \oplus a_1 \oplus a_2 \\
c_3 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \\
c_4 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \\
c_5 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \\
c_6 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \\
c_7 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \\
c_8 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus a_8 \\
c_9 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus a_8 \oplus a_9 \\
c_{10} &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus a_8 \oplus a_9 \oplus a_{10} \\
c_{11} &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus a_8 \oplus a_9 \oplus a_{10} \oplus a_{11}
\end{aligned}
$$

Parallel Algorithm for Scan Operation

– Reduce redundant calculation:

$$
\begin{aligned}
c_0 &= a_0 \\
c_1 &= a_0 \oplus a_1 \\
c_2 &= a_0 \oplus a_1 \oplus a_2 \\
c_3 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \\
c_4 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \\
c_5 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \\
c_6 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \\
c_7 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \\
c_8 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus a_8 \\
c_9 &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus a_8 \oplus a_9 \\
c_{10} &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus a_8 \oplus a_9 \oplus a_{10} \\
c_{11} &= a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus a_8 \oplus a_9 \oplus a_{10} \oplus a_{11}
\end{aligned}
$$

Divide the computation into blocks

Parallel Algorithm for Scan Operation

- Reduce redundant calculation:

$$c_{i-j} = a_i \oplus a_{i+1} \cdots \oplus a_j$$

e.g., $c_{3-5} = a_3 \oplus a_4 \oplus a_5$

$$c_{i-k} \oplus c_{(k+1)-j} = c_{i-j}$$

e.g., $c_{0-2} \oplus c_{3-5} = c_{0-5}$

$$
\begin{aligned}
c_0 &= a_0 \\
c_1 &= a_0 \oplus a_1 \\
c_2 &= a_0 \oplus a_1 \oplus a_2 \\
c_3 &= c_{0-2} \oplus a_3 \\
c_4 &= c_{0-2} \oplus a_3 \oplus a_4 \\
c_5 &= c_{0-2} \oplus a_3 \oplus a_4 \oplus a_5 \\
c_6 &= c_{0-2} \oplus c_{3-5} \oplus a_6 \\
c_7 &= c_{0-2} \oplus c_{3-5} \oplus a_6 \oplus a_7 \\
c_8 &= c_{0-2} \oplus c_{3-5} \oplus a_6 \oplus a_7 \oplus a_8 \\
c_9 &= c_{0-2} \oplus c_{3-5} \oplus c_{6-8} \oplus a_9 \\
c_{10} &= c_{0-2} \oplus c_{3-5} \oplus c_{6-8} \oplus a_9 \oplus a_{10} \\
c_{11} &= c_{0-2} \oplus c_{3-5} \oplus c_{6-8} \oplus a_9 \oplus a_{10} \oplus a_{11}
\end{aligned}
$$

Parallel Algorithm for Scan Operation

- Parallel algorithm:
- Stage 1: Each thread gets a subset of n/t elements (block partitioning), performs local scan operation on assigned elements, and then thread i (except the last one) stores its last local c element to a working array at $w[i+1]$

$[a_0, a_1, a_2]$	$[a_3, a_4, a_5]$	$[a_6, a_7, a_8]$
$c_0 = a_0$	$c_3 = a_3$	$c_6 = a_6$
$c_{0-1} = c_0 \oplus a_1$	$c_{3-4} = c_3 \oplus a_4$	$c_{6-7} = c_6 \oplus a_7$
$c_{0-2} = c_{0-1} \oplus a_2$	$c_{3-5} = c_{3-4} \oplus a_5$	$c_{6-8} = c_{6-7} \oplus a_8$

$$W: [0, c_{0-2}, c_{3-5}]$$

Sync must be applied (implicitly or explicitly) btw stages!

- Stage 2: A single thread performs scan operation on W

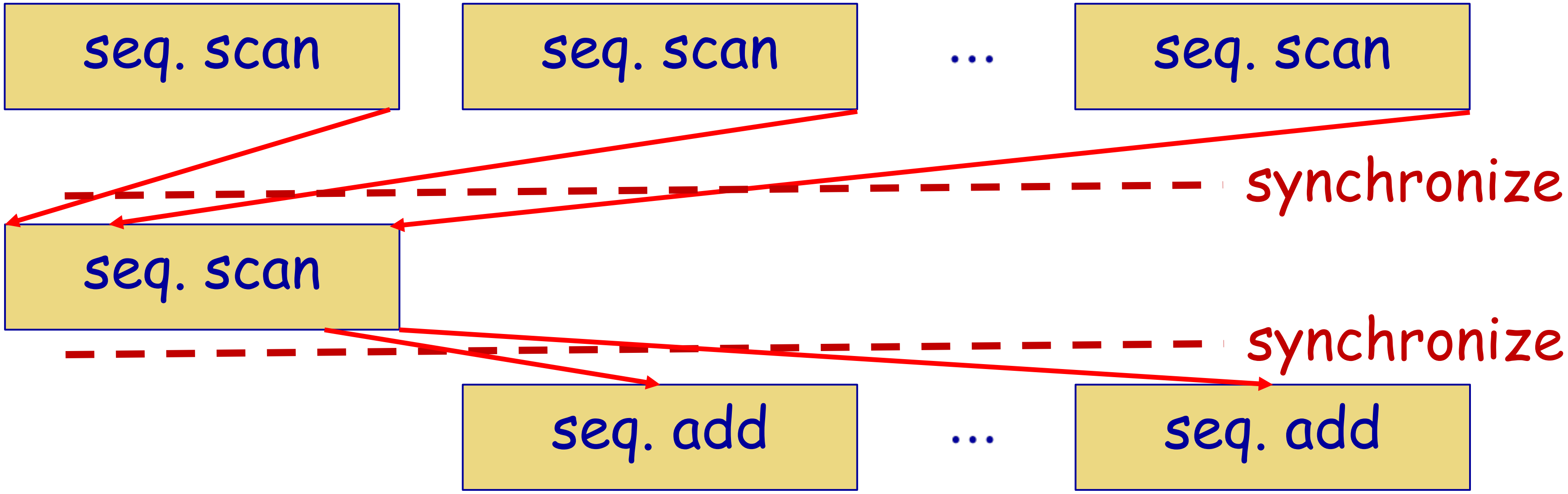
$$W: [0, c_{0-2}, c_{0-5}]$$

- Stage 3: Each thread $\oplus\, w[i]$ to every partial scan element

$c_0 = c_0 \oplus 0$	$c_3 = c_3 \oplus c_{0-2}$	$c_6 = c_6 \oplus c_{0-5}$
$c_1 = c_{0-1} \oplus 0$	$c_4 = c_{3-4} \oplus c_{0-2}$	$c_7 = c_{6-7} \oplus c_{0-5}$
$c_2 = c_{0-2} \oplus 0$	$c_5 = c_{3-5} \oplus c_{0-2}$	$c_8 = c_{6-8} \oplus c_{0-5}$

Parallel Algorithm for Scan Operation

- To summarize, this parallel algorithm consists of three parallel stages
  - Synchronization between stages
  - In 2nd stage only one thread is active
    - Purely sequential, but overhead not heavy as SM machines are usually small



Parallel Algorithm for Scan Operation

- Total amount of operations:
  - Stage 1: each thread performs local scan on n/t elements
    - $(n/t - 1) * t = n - t$
  - Stage 2: a single thread performs scan on W
    - $t - 1$ (the size of W is t)
  - Stage 3: each thread add w_i to the partially scanned elements
    - $n/t * t = n$
- Thus the total amount is
  - $n - t + t - 1 + n = 2n - 1$
- The total amount of operations in sequential computation is $n - 1$
- Thus the parallel algorithm takes less amount of time to complete the computation if the number of threads > 2

Homework 3: Parallel Scan Operation

- Implement the parallel algorithm for scan operation using OpenMP
- Compare the performance with the sequential algorithm

- In a parallel region, if we want part of code to be done by a single (any) thread, we can use single directive

#pragma omp single

- All other threads will skip the single region and stop at the barrier at the end of the single construct until all threads have reached the barrier

THE UNIVERSITY OF SYDNEY