Parallel Sort and More Open MP

Chapter 5.6

Bryan Mills, PhD
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pairwise comparisons and swapping
"Bubbling Up" the Largest Element

• Traverse a collection of elements
  – Move from the front to the end
  – “Bubble” the largest value to the end using pairwise comparisons and swapping

```
1  2  3  5  6
42 35 77 12 101 5
```
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pairwise comparisons and swapping

```
1  2  3  4  5  6
42 35 12 77 101 5
```
"Bubbling Up" the Largest Element

• Traverse a collection of elements
  – Move from the front to the end
  – “Bubble” the largest value to the end using pairwise comparisons and swapping

No need to swap
"Bubbling Up" the Largest Element

• Traverse a collection of elements
  – Move from the front to the end
  – "Bubble" the largest value to the end using pairwise comparisons and swapping

```
1  2  3  4  5  6
42 35 12 77 5 101
```
"Bubbling Up" the Largest Element

• Traverse a collection of elements
  – Move from the front to the end
  – “Bubble” the largest value to the end using pairwise comparisons and swapping

Largest value correctly placed
Reducing the Number of Comparisons

1  2  3  4  5  6

77  42  35  12  101  5

1  2  3  4  5  6

42  35  12  77  5  101

1  2  3  4  5  6

35  12  42  5  77  101

1  2  3  4  5  6

12  35  5  42  77  101

1  2  3  4  5  6

12  5  35  42  77  101
Bubble Sort

for (list_length = n; list_length >= 2; list_length--) {
    for (i = 0; i < list_length; i++) {
        if (a[i] > a[i+1]) {
            tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
        }
    }
}

Serial Odd-Even Transposition Sort

```c
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) Swap(&a[i-1], &a[i]);
    else
        for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```
Serial Odd-Even Transposition Sort

<table>
<thead>
<tr>
<th>Phase</th>
<th>Subscript in Array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>9 ↔ 7</td>
</tr>
<tr>
<td></td>
<td>7 9</td>
</tr>
<tr>
<td>1</td>
<td>7 9</td>
</tr>
<tr>
<td></td>
<td>7 6</td>
</tr>
<tr>
<td>2</td>
<td>7 ↔ 6</td>
</tr>
<tr>
<td></td>
<td>6 7</td>
</tr>
<tr>
<td>3</td>
<td>6 7</td>
</tr>
<tr>
<td></td>
<td>6 7</td>
</tr>
</tbody>
</table>
First OpenMP Odd-Even Sort

for (phase = 0; phase < n; phase++) {
    if (phase % 2 == 0)
        # pragma omp parallel for num_threads(thread_count) \
        default(none) shared(a, n) private(i, tmp)
        for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
                tmp = a[i-1];
                a[i-1] = a[i];
                a[i] = tmp;
            }
        }
    else
        # pragma omp parallel for num_threads(thread_count) \
        default(none) shared(a, n) private(i, tmp)
        for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
                tmp = a[i+1];
                a[i+1] = a[i];
                a[i] = tmp;
            }
        }
}

Inc. All rights Reserved
Second OpenMP Odd-Even Sort

```c
#pragma omp parallel num_threads(thread_count) \
  default(none) shared(a, n) private(i, tmp, phase)
for (phase = 0; phase < n; phase++) {
  if (phase % 2 == 0)
    # pragma omp for
    for (i = 1; i < n; i += 2) {
      if (a[i-1] > a[i]) {
        tmp = a[i-1];
        a[i-1] = a[i];
        a[i] = tmp;
      }
    }
  else
    # pragma omp for
    for (i = 1; i < n-1; i += 2) {
      if (a[i] > a[i+1]) {
        tmp = a[i+1];
        a[i+1] = a[i];
        a[i] = tmp;
      }
    }
}
```

Copyright © 2010, Elsevier Inc. All rights Reserved
Odd-even sort with two parallel for directives and two for directives. (Times are in seconds.)

<table>
<thead>
<tr>
<th>thread_count</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two parallel for directives</td>
<td>0.770</td>
<td>0.453</td>
<td>0.358</td>
<td>0.305</td>
</tr>
<tr>
<td>Two for directives</td>
<td>0.732</td>
<td>0.376</td>
<td>0.294</td>
<td>0.239</td>
</tr>
</tbody>
</table>
Parallel Merge Sort

• How can we do this?
## Merge Sort Example

| 99 | 6  | 86 | 15 | 58 | 35 | 86 | 4  | 0  |
## Merge Sort Example

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|99 |6  |86 |15 |58 |35 |86 |4  |0  |

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|99 |6  |86 |15 |   |   |   |   |
|   |   |   |   |   |58 |35 |86 |4  |0  |
### Merge Sort Example

<table>
<thead>
<tr>
<th>99</th>
<th>6</th>
<th>86</th>
<th>15</th>
<th>58</th>
<th>35</th>
<th>86</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Merge Sort Example

<table>
<thead>
<tr>
<th>99</th>
<th>6</th>
<th>86</th>
<th>15</th>
<th>58</th>
<th>35</th>
<th>86</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Merge Sort Example

99  6  86  15  58  35  86
0   4

Merge
Merge Sort Example

Merge
Merge Sort Example

6 15 86 99
6 99 15 86
0 4 35 58 86
58 35 0 4 86

Merge
Merge Sort Example

```
0  4  6  15  35  58  86  86  99
```

```
6  15  86  99
```

```
0  4  35  58  86
```

Merge
Merge Sort Example

| 0 | 4 | 6 | 15 | 35 | 58 | 86 | 86 | 99 |
void MergeSort(int *A, int n) {
    int mid, i, *L, *R;
    if(n < 2) return;
    mid = n/2;  // find the mid index.
    L = (int*)malloc(mid*sizeof(int));
    R = (int*)malloc((n - mid)*sizeof(int));

    for(i = 0;i<mid;i++) L[i] = A[i];
    for(i = mid;i<n;i++) R[i-mid] = A[i];

    MergeSort(L,mid);  // sorting the left subarray
    MergeSort(R,n-mid);  // sorting the right subarray
    Merge(A,L,mid,R,n-mid);
}

This is looking good, lets make execute in parallel!
void Merge(int *A, int *L, int leftCount, int *R, int rightCount) {
    int i, j, k;
    // i - to mark the index of left subarray (L)
    // j - to mark the index of right subarray (R)
    // k - to mark the index of merged subarray (A)
    i = 0; j = 0; k = 0;

    while (i < leftCount && j < rightCount) {
        if (L[i] < R[j]) A[k++] = L[i++];
        else A[k++] = R[j++];
    }
    while (i < leftCount) A[k++] = L[i++];
    while (j < rightCount) A[k++] = R[j++];
}

Merge is the problem? How do we merge in parallel?
void MergeSort(int t, int p, int r, int *a) {
    int mid, i, n;
    n = r - p;
    if(n < 2) return;
    mid = n/2;  // find the mid index.
    MergeSort(t, p, mid, a);  // sorting the left subarray
    MergeSort(t, mid+1, r, a);  // sorting the right subarray
    Merge(t, p, mid, mid+1, r, a, p);
}

Note issues with inplace, ignore for now!
Parallel Merge
Merge two ranges of array $T[ p1 .. r1 ]$ and $T[ p2 .. r2 ]$ into destination array $A$ starting at index $p3$.

```c
Void MergePar(int t, int p1, int r1, int p2, int r2,
               int* a, int p3 )
{
    int length1 = r1 - p1 + 1;
    int length2 = r2 - p2 + 1;
    if ( length1 < length2 )
    {
        exchange(      p1,      p2 );
        exchange(      r1,      r2 );
        exchange( length1, length2 );
    }
    if ( length1 == 0 ) return;
    int q1 = ( p1 + r1 ) / 2;
    int q2 = BinarySearch( t[ q1 ], t, p2, r2 );
    int q3 = p3 + ( q1 - p1 ) + ( q2 - p2 );
    a[ q3 ] = t[ q1 ];
    MergePar( t, p1,     q1 - 1, p2, q2 - 1, a, p3     );
    MergePar( t, q1 + 1, r1,     q2, r2,     a, q3 + 1 );
}
```

How to Parallelize?

Sections

• Sections will be executed in parallel
• Can combine parallel and section like we did with for
• If more threads than sections then idle threads will exist
• If less threads than sections then some sections will execute in serial
Section

#define N 1000
main (){ 
    int i;float a[N], b[N], c[N];
    for (i=0; i < N; i++) a[i] = b[i] = ... ;

    # pragma omp parallel shared(a,b,c) private(i)
    {
    
    ...

    # pragma omp sections
    {
    # pragma omp section
    {
        for (i=0; i < N/2; i++) c[i] = a[i] + b[i];
    }
    # pragma omp section
    {
        for (i=N/2; i < N; i++) c[i] = a[i] + b[i];
    }
    } /* end of sections */

    ...

    } /* end of parallel */
}
# How to use in Parallel Merge

. . .

```c
#pragma omp parallel
{
    # pragma omp sections
    {
        # pragma omp section
        MergePar( t, p1, q1 - 1, p2, q2 - 1, a, p3 );
        # pragma omp section
        MergePar( t, q1 + 1, r1, q2, r2, a, q3 + 1 );
    } // END sections block
} // end parallel block
```

. . .

Note the recursion with parallel sections
Does this work? How?
We’ll spend sometime discussing next week
Single and Master

- Single indicates that only one thread in team will execute the code.
  
  #pragma omp single

- Master indicates that only the master thread will execute the code.
  
  #pragma omp master
Critical with Names

• Critical ensures that only one thread can execute code block at a time.

  # pragma omp critical
global_result += my_result ;

• You can name critical sections, then critical sections with different names can be executed in parallel

  # pragma omp critical(name)
global_result += my_result ;
Barrier

• When a thread executes a barrier it will wait until all other threads in the team also execute the barrier.

• Then threads continue working as usual.

  # pragma omp barrier
  ... continue on past...
Atomic

• If your critical section is of the form
  \[ x \ += \ y \ OR \ x \ = \ x \ <\text{operation}> \ y \]
• Compiler can use hardware atomic operations
• Think “mini-critical”

```c
#pragma omp parallel for
shared(sum)
for(i = 0; i < n; i++){
    value = f(a[i]);
    #pragma omp atomic
    sum = sum + value;
}
```
We want to parallelize this loop.

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

<table>
<thead>
<tr>
<th>Thread</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, n/t, 2n/t, ...</td>
</tr>
<tr>
<td>1</td>
<td>1, n/t + 1, 2n/t + 1, ...</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>t - 1</td>
<td>t - 1, n/t + t - 1, 2n/t + t - 1, ...</td>
</tr>
</tbody>
</table>

Assignment of work using cyclic partitioning.
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}

Our definition of function $f$. 
Results

• $f(i)$ calls the sin function $i$ times.
• Assume the time to execute $f(2i)$ requires approximately twice as much time as the time to execute $f(i)$.

• $n = 10,000$
  – one thread
  – run-time = 3.67 seconds.
Results

• \( n = 10,000 \)
  – two threads
  – default assignment
  – run-time = 2.76 seconds
  – speedup = 1.33

• \( n = 10,000 \)
  – two threads
  – cyclic assignment
  – run-time = 1.84 seconds
  – speedup = 1.99
The Schedule Clause

• **Default schedule:**

```c
sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \ 
  reduction(+:sum)
for (i = 0; i <= n; i++)
  sum += f(i);
```

• **Cyclic schedule:**

```c
sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \ 
  reduction(+:sum) schedule(static,1)
for (i = 0; i <= n; i++)
  sum += f(i);
```
schedule ( type, chunksize )

• Type can be:
  – static: the iterations can be assigned to the threads before the loop is executed.
  – dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  – auto: the compiler and/or the run-time system determine the schedule.
  – runtime: the schedule is determined at run-time.

• The chunksize is a positive integer.
The Static Schedule Type

twelve iterations, 0, 1, . . . , 11, and three threads

\texttt{schedule} (static, 1)

Thread 0 : 0, 3, 6, 9
Thread 1 : 1, 4, 7, 10
Thread 2 : 2, 5, 8, 11
The Static Schedule Type

twelve iterations, 0, 1, \ldots, 11, and three threads

\texttt{schedule(static, 2)}

- Thread 0: 0, 1, 6, 7
- Thread 1: 2, 3, 8, 9
- Thread 2: 4, 5, 10, 11
The Static Schedule Type

twelve iterations, 0, 1, \ldots, 11, and three threads

\texttt{schedule(static, 4)}

\begin{align*}
\text{Thread 0 : } & 0, 1, 2, 3 \\
\text{Thread 1 : } & 4, 5, 6, 7 \\
\text{Thread 2 : } & 8, 9, 10, 11
\end{align*}
The Dynamic Schedule Type

• The iterations are also broken up into chunks of \textit{chunksize} consecutive iterations.
• Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
• This continues until all the iterations are completed.
• The \textit{chunksize} can be omitted. When it is omitted, a \textit{chunksize} of 1 is used.
The Guided Schedule Type

• Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
• However, in a guided schedule, as chunks are completed the size of the new chunks decreases.
• If no `chunksize` is specified, the size of the chunks decreases down to 1.
• If `chunksize` is specified, it decreases down to `chunksize`, with the exception that the very last chunk can be smaller than `chunksize`.
Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

<table>
<thead>
<tr>
<th>Thread</th>
<th>Chunk</th>
<th>Size of Chunk</th>
<th>Remaining Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 – 5000</td>
<td>5000</td>
<td>4999</td>
</tr>
<tr>
<td>1</td>
<td>5001 – 7500</td>
<td>2500</td>
<td>2499</td>
</tr>
<tr>
<td>1</td>
<td>7501 – 8750</td>
<td>1250</td>
<td>1249</td>
</tr>
<tr>
<td>1</td>
<td>8751 – 9375</td>
<td>625</td>
<td>624</td>
</tr>
<tr>
<td>0</td>
<td>9376 – 9687</td>
<td>312</td>
<td>312</td>
</tr>
<tr>
<td>1</td>
<td>9688 – 9843</td>
<td>156</td>
<td>156</td>
</tr>
<tr>
<td>0</td>
<td>9844 – 9921</td>
<td>78</td>
<td>78</td>
</tr>
<tr>
<td>1</td>
<td>9922 – 9960</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>1</td>
<td>9961 – 9980</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>1</td>
<td>9981 – 9990</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>9991 – 9995</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>9996 – 9997</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>9998 – 9998</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>9999 – 9999</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The Runtime Schedule Type

• The system uses the environment variable `OMP_SCHEDULE` to determine at runtime how to schedule the loop.

• The `OMP_SCHEDULE` environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.
Graveyard