Case Study: Implementing Enumeration Sort in OpenMP

Jarmo Rantakokko
Senior lecturer, IT UU

**Purpose:** To study and identify different parallel overheads in OpenMP (we are not interested in how to parallelize enumsort in the best way)

**Algorithm:** Enumeration Sort

```c
for (j=0; j<len; j++)
{
    rank=0;
    for (i=0; i<len; i++)
        if (indata[i]<indata[j]) rank++;
    outdata[rank]=indata[j];
}
```

For each element (j) check how many other elements (i) are smaller than it => rank

Perfectly parallel tasks for each element (j)
Alternative 1: Parallelize j-loop

```c
#pragma omp parallel for private(rank,i)
for (j=0;j<len;j++)
{
    rank=0;
    for (i=0;i<len;i++)
        if (indata[i]<indata[j]) rank++;
    outdata[rank]=indata[j];
}
```

**Note 1:** If rank equal on two threads
⇒ Race condition (but write same data)

**Note 2:** All threads reading all data for each element
⇒ Mem BW limited performance (if data does not fit in cache, especially bad on NUMA)

Alternative 2: Parallelize i-loop

```c
for (j=0;j<len;j++)
{
    rank=0;
    #pragma omp parallel for reduction (+:rank)
    for (i=0;i<len;i++)
        if (indata[i]<indata[j]) rank++;
    outdata[rank]=indata[j];
}
```

**Note 1:** Frequent creation/termination of threads and small tasks per thread (high parallel overhead)

**Note 2:** Each thread works only on a part of the data in all iterations, good for cache performance (if the whole array does not fit in cache).
Also no race condition, only master updates
Alternative 3: Use one parallel region

```c
#pragma omp parallel private(j)
{
    for (j=0; j<len; j++)
    {
        #pragma omp single
        { rank=0; }
    
        #pragma omp for reduction (+:rank)
        for (i=0; i<len; i++)
            if (indata[i]<indata[j]) rank++;
    
        #pragma omp single
        { outdata[rank]=indata[j]; }
    }
}
```

**Note:** 3 barriers per iteration, how can we decrease the number of synchronization points?

Alternative 4: Interleave two iterations

```c
j1=0; j2=1; rank1=0; rank2=0;
#pragma omp parallel
{
    while (j1<len)
    {
        #pragma omp for reduction (+:rank1) // Barrier
        for (i=0; i<len; i++)
            if (indata[i]<indata[j1]) rank1++;
        
        #pragma omp single nowait
        { outdata[rank1]=indata[j1];
          rank1=0; j1+=2; }
        
        if (j2>=len) break;
        
        #pragma omp for reduction (+:rank2) // Barrier
        for (i=0; i<len; i++)
            if (indata[i]<indata[j2]) rank2++;
        
        #pragma omp single nowait
        { outdata[rank2]=indata[j2];
          rank2=0; j2+=2; }
    }
}
Results (runtime):

<table>
<thead>
<tr>
<th>Nthr</th>
<th>Enum1</th>
<th>Enum2</th>
<th>Enum3</th>
<th>Enum4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.5</td>
<td>13.0</td>
<td>11.7</td>
<td>12.0</td>
</tr>
<tr>
<td>2</td>
<td>5.80</td>
<td>7.90</td>
<td>9.23</td>
<td>7.80</td>
</tr>
<tr>
<td>4</td>
<td>2.96</td>
<td>5.73</td>
<td>9.05</td>
<td>5.57</td>
</tr>
<tr>
<td>8</td>
<td>1.50</td>
<td>7.90</td>
<td>38.1</td>
<td>6.13</td>
</tr>
</tbody>
</table>

What overheads do we have?

**Enum1**: All threads read all data in all iterations, mem BW limited performance (if small cache).

**Enum2**: Create/terminate threads in each iteration

**Enum3**: Three barriers per iteration

**Enum 4**: One barrier per iteration => Memory flush
We have update of invalid cache-lines, outdata is updated irregularly (randomly) and we get "communication" due to (false) sharing in outdata

**Alternative 5**: Let each thread be responsible for a fixed section of outdata and only that thread writes in the corresponding locations

```
0       outdata       len

Thread 0    Thread 1    Thread 2    Thread 3
```
Alternative 5: Owner writes

```c
j1=0; j2=1; rank1=0; rank2=0;
#pragma omp parallel
{
    while (j1<len)
    {
        #pragma omp for reduction (+:rank1)
        for (i=0;i<len;i++)
            if (indata[i]<indata[j1]) rank1++;
        if (rank1/(len/nthr)==thrid)
            { outdata[rank1]=indata[j1];
                rank1=0; j1+=2; }
        if (j2=len) break;
        #pragma omp for reduction (+:rank2)
        for (i=0;i<len;i++)
            if (indata[i]<indata[j2]) rank2++;
        if (rank2/(len/nthr)==thrid)
            { outdata[rank2]=indata[j2];
                rank2=0; j2+=2; }
    }
}
```

Results (runtime):

<table>
<thead>
<tr>
<th>Nthr</th>
<th>Enum1</th>
<th>Enum2</th>
<th>Enum3</th>
<th>Enum4</th>
<th>Enum5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.5</td>
<td>13.0</td>
<td>11.7</td>
<td>12.0</td>
<td>11.4</td>
</tr>
<tr>
<td>2</td>
<td>5.80</td>
<td>7.90</td>
<td>9.23</td>
<td>7.80</td>
<td>6.90</td>
</tr>
<tr>
<td>4</td>
<td>2.96</td>
<td>5.73</td>
<td>9.05</td>
<td>5.57</td>
<td>5.20</td>
</tr>
<tr>
<td>8</td>
<td>1.50</td>
<td>7.90</td>
<td>38.1</td>
<td>6.13</td>
<td>5.75</td>
</tr>
</tbody>
</table>

**Enum1**: Good if all data fits in cache (small arrays), perfectly parallel, no synchronization of threads.

**Enum5**: Still one barrier per iteration and small work load per thread between synchronizations but good cache locality for large arrays.
Alternative 6, Reformulate algorithm:

Reformulate algorithm. Save ranks in a new array and compute outdata in the end using the rank-array, i.e., outdata[rank[i]]=indata[i];

In parallel, split indata and rank-arrays and compute partial ranks in parallel with compute-and-shift. Use an extra copy of the indata-array which is shifted p-times. In each step compute the partial ranks for each element and add to the rank-array.

Shift data cyclicly and compute partial ranks adding to rank-array
Alternative 6: Reformulate algorithm

Use a rank-array and compare with MPI-version, no need to shift data as we have shared memory.

```c
#pragma omp parallel private(k, i1, i2, thrid)
{
    thrid = omp_get_thread_num();
    slice = len / nthr;
    for (k=0; k<nthr; k++) // The "shift" loop
    {
        i1 = slice * ((thrid+k) % nthr); i2 = i1 + slice;
        #pragma omp for private(i) nowait
        for (j=0; j<len; j++)
        {
            if (indata[i] < indata[j]) rank[j]++;
        }
    }
    for (j=0; j<len; j++) outdata[rank[j]] = indata[j];
}
```

Note: We can not write this only with directives!

Results (runtime):

<table>
<thead>
<tr>
<th>Nthr</th>
<th>Enum1</th>
<th>Enum2</th>
<th>Enum3</th>
<th>Enum4</th>
<th>Enum5</th>
<th>Enum6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.5</td>
<td>13.0</td>
<td>11.7</td>
<td>12.0</td>
<td>11.4</td>
<td>11.9</td>
</tr>
<tr>
<td>2</td>
<td>5.80</td>
<td>7.90</td>
<td>9.23</td>
<td>7.80</td>
<td>6.90</td>
<td>5.53</td>
</tr>
<tr>
<td>4</td>
<td>2.96</td>
<td>5.73</td>
<td>9.05</td>
<td>5.57</td>
<td>5.20</td>
<td>2.76</td>
</tr>
<tr>
<td>8</td>
<td>1.50</td>
<td>7.90</td>
<td>38.1</td>
<td>6.13</td>
<td>5.75</td>
<td>1.51</td>
</tr>
</tbody>
</table>

**Enum6**: Good cache locality (even for large arrays)
No synchronization of threads but a small serial section run on the master thread.

The difference between *Enum1* and *Enum6* will increase with problem size in favor for *Enum6.*
Alternative 7: Nested parallelism

```c
omp_set_nested(1);
#pragma omp parallel for private(rank) num_threads(4)
for (j=0; j<len; j++)
{
    rank=0;
    #pragma omp parallel for reduction (+:rank) num_threads(2)
    for (i=0; i<len; i++)
        if (indata[i]<indata[j]) rank++;
    outdata[rank]=indata[j];
}
```

**Note:** Increase the parallel overhead compared to Enum1 (create/terminate threads in each iteration j)
Decrease the parallel overhead compared to Enum2 (synchronize a smaller team of threads, larger task/thread)

---

Quick Sort

**Algorithm:**
1. Select pivot element
2. Divide data into two sets (smaller or larger)
3. Sort each set with Quick Sort

**Parallelism:**
In each split acquire a new processor/thread and proceed with the two lists in parallel.

⇒ Limited parallelism (in the first step only one processor have something to do, waste if resources) and poor load balance due to pivot selection.

Reformulate the algorithm!
Parallel Quick Sort

Algorithm:
1. Divide the data into p equal parts
2. Sort the data locally in each processor
3. Perform global sort
   3.1 Select pivot in each processor set
   3.2 In each processor, divide the data into two sets (smaller or larger)
   3.3 Split the processors into two groups and exchange data pair-wise
   3.4 Merge data into a sorted list in each processor
4. Repeat 3.1-3.4 recursively for each processor group
Step 1, Divide data into p equal parts

Step 2, Sort locally in each processor
Step 3.1 Select pivot

Step 3.2, 3.3 Divide and exchange
Step 3.4, Merge into a sorted list

Step 3.1 Select pivot
Step 3.2, 3.3 Divide and exchange

Step 3.4, Merge into a sorted list
Step 3.1 Select pivot

Step 3.2, 3.3 Divide and exchange
Step 3.4, Merge into a sorted list

Final state
Pivot selection strategies:

**Strategy 1**: Select median in processor 0 in each processor set (communicator) and step. (OK if data equally rand, bad if almost sorted)

**Strategy 2**: Select the mean of all medians in respective processor set and step. (Can give too much weight to extreme medians)

**Strategy 3**: Sort the medians and select the mean value of the two middlemost medians in each processor set and step. (Independent of dist but more costly strategy)

Bucket Sort

- Filter the elements into buckets
- Assign buckets to processors
- Sort the buckets, concurrently
Filtering

Filtering is a linear operation for equal sized buckets in the interval \([\text{min max}]\).

Bucket \(b = \text{ceil}((a[i] - \text{min})/(\text{max} - \text{min}) \times \text{nbuck}) - 1\)

with special case \(a[i] = \text{min}\).

On distributed memory, scatter indata and shift in a ring collecting data into respective bucket.

On shared memory let each thread go through the whole array and pick elements to its own bucket.

---

Performance obstacles

Not perfectly parallel, why?
- Communication (on distributed memory)
- Filtering is not parallel, elements are checked several times (of different processors) if they belong to the bucket, on shared memory all threads check all elements (duplicated work)
- Load imbalance in the different buckets, can be severe depending on the distribution

Load balancing strategy:
Create many small buckets and threads (>cores) and let the operating system load balance the computations by time-sharing of threads on the cores