Case Study: Implementing Enumeration Sort in OpenMP

Jarmo Rantakokko
Senior lecturer, IT UU

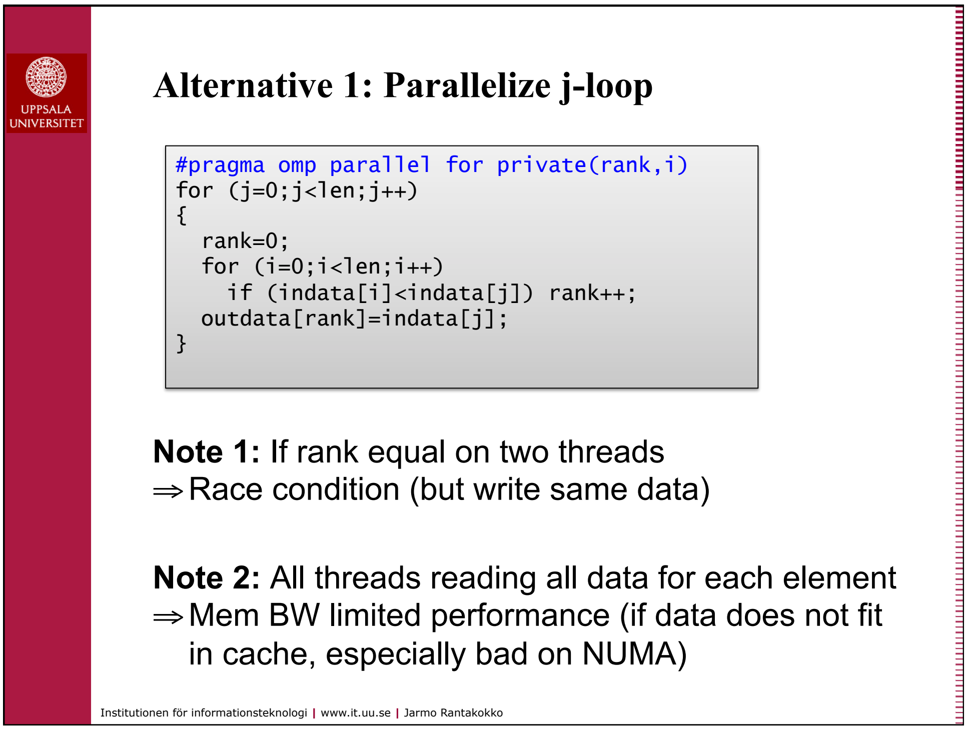
UPPSALA UNIVERSITET

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

```
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)

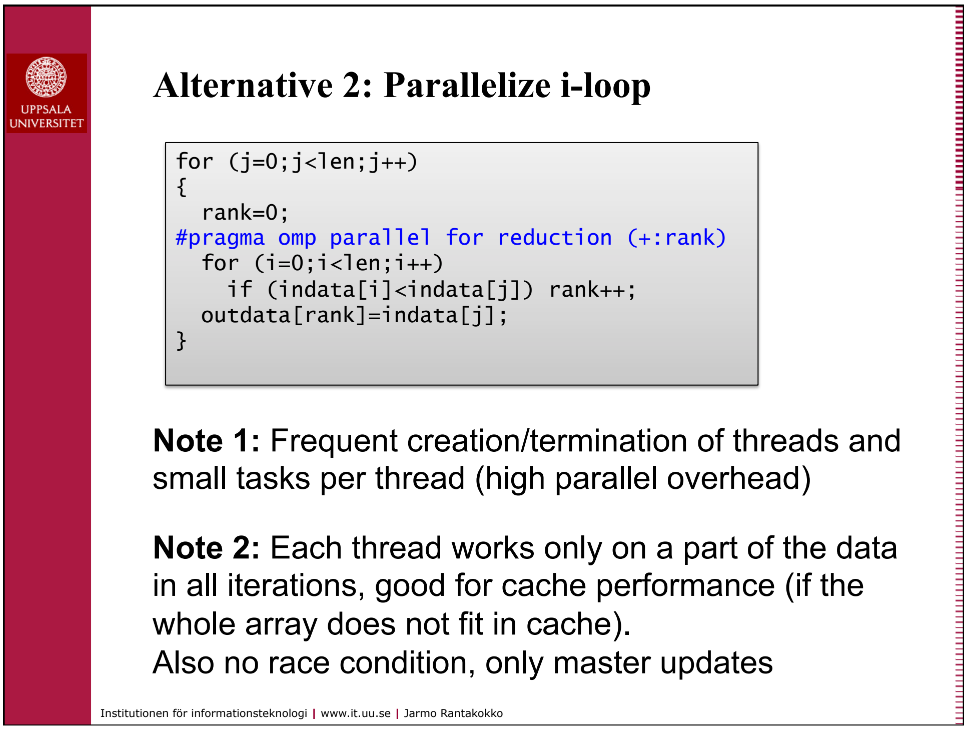


Alternative 1: Parallelize j-loop

```
#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)

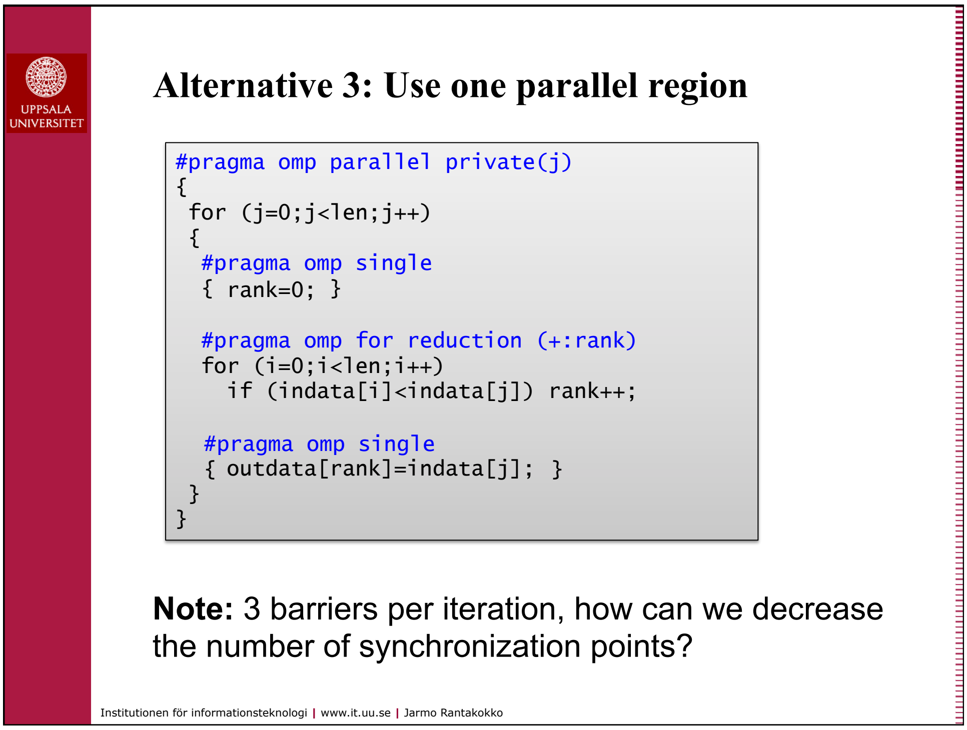


Alternative 2: Parallelize i-loop

```
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)

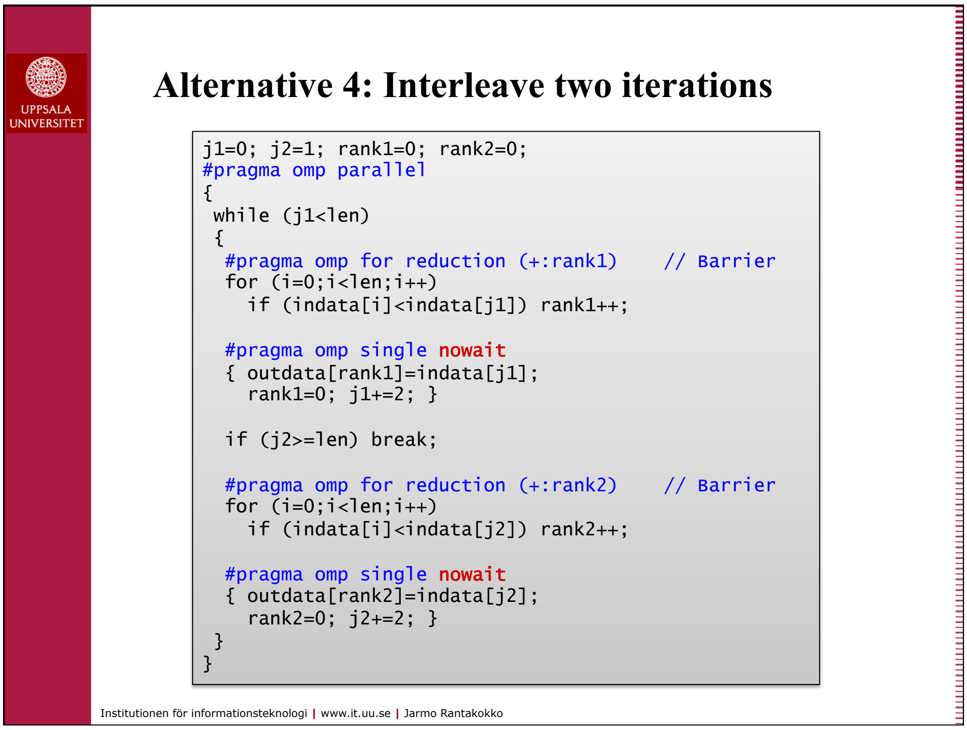
Note 2: Each thread works only on a part of the data in all iterations, good for cache performance (if the whole array does not fit in cache).
Also no race condition, only master updates



Alternative 3: Use one parallel region

```
#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

```
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):

Nthr	Enum1	Enum2	Enum3	Enum4
1	11.5	13.0	11.7	12.0
2	5.80	7.90	9.23	7.80
4	2.96	5.73	9.05	5.57
8	1.50	7.90	38.1	6.13
16	0.75	NA	NA	NA

What overheads do we have?

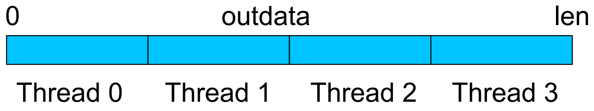
Enum1: All threads read all data in all iterations, mem BW limited performance (if small cache).

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



Alternative 5: Owner writes

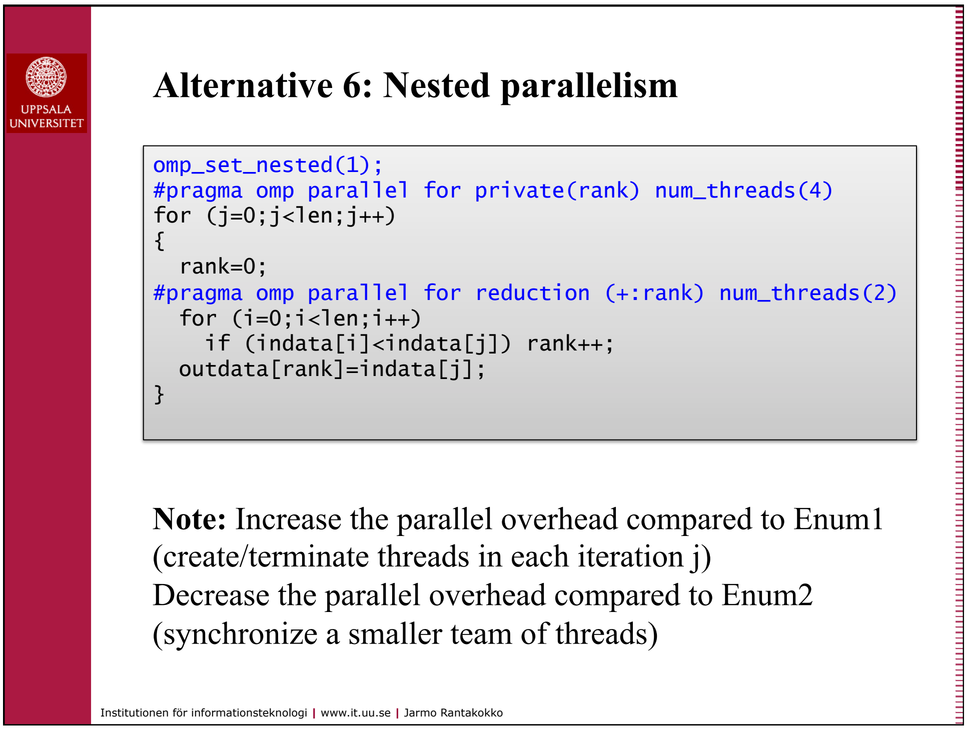
```
j1=0; j2=1; rank1=0; rank2=0;
#pragma omp parallel
{
 while (j1<len)
 {
  #pragma omp for reduction (+:rank1)
  for (i=0;i<len;i++)
    if (indata[i]<indata[j1]) rank1++;

  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):

Nthr	Enum1	Enum2	Enum3	Enum4	Enum5
1	11.5	13.0	11.7	12.0	11.4
2	5.80	7.90	9.23	7.80	6.90
4	2.96	5.73	9.05	5.57	5.20
8	1.50	7.90	38.1	6.13	5.75
16	0.75	NA	NA	NA	NA

Enum1: All data fits in cache, perfectly parallel
Maximal work per thread, minimal parallel overhead

Enum5: Still one barrier per iteration and small work load per thread between synchronizations (we need to go up in problem size to pay off)



Alternative 6: Nested parallelism

```
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)