Parallelization of sorting

1) Write parallel program, which reads and sorts file **sort.txt** increasingly according to 1st column (and 2nd if the values in 1st column are equal) to output file **sorted.txt**. Number of items (lines) is determined by reading (until EOF or error)

Input - sort.txt

=======

0.071109	66.49892864
0.528986045	38.32743474
0.348693158	72.68342121
0.450123953	87.86723222

Output - sorted.txt

=======

0.071109	66.49892864
0.348693158	72.68342121
0.450123953	87.86723222
0.528986045	38.32743474

Input files 10.txt, 1k.txt, 10k.txt,100k.txt are in directory ~student/pp/sort

Copy the one you want to sort to **sort.txt**

Parallelization of sorting

2) Serial version of sorting program is provided - **sort.c**

It does not change the location of data in arrrays **b** and **c** (1st and 2nd column, resp.), but creates a pointer-like array **ind**, which yields the information about sorted order

Example: If the first sorted items are e.g. lines with indices 105, 7, and 34, then ind[0]=105, ind[105]=7, ind[7]=34. The final order is given by this recursive calling of cur=ind[cur], where cur indicates current value (line)

If you don't like the provided **sort.c**, use its initial (reading) and final (writing – will be affected by your method of sorting) parts and **plug in your own sorting engine.**

The algorithm does not need to be the most efficient, in fact the used bubble sort is far from being the fastest algorithm.

Parallelization of sorting

Example, how the sorting with pointer-wise index works – animated in Powerpoint

Input - sort.txt

=======

index	value
1	0.071109
2	0.528986045
3	0.348693158
4	0.450123953

ind[0] = 1
ind[1] = 2
ind[2] = undef.
ind[3] = 4
ind[4] = 3

Order during sorting

=======

ordered index	original index	value
1	1	0.071109
2	3	0.348693158
3	2	0.528986045
4	4	0.450123953

Parallelization of sorting

3) In the directory ~student/pp/sort, there is also a code **check.c** verifying if **sorted.txt** is sorted correctly in increasing order

You can use it to verify that your sorting program works correctly

Best verification that your parallel sorting works is to compare the output with that of serial sorting. You can use the linux command **diff**, e.g.

diff sorted.txt sorted-100k-serial.txt

4) Measure the time of sorting of 100k.txt when executed on 1, 4, 8, and 12 threads (or even other number) within one node

It can happen that calculation on more threads will be faster than you'd anticipate ☺

To do the same for 10k.txt is welcome

You can parallelize your code with either MPI or OpenMP

If using OpenMP, you might copy the data from a shared array to your private arrays which are then sorted independently.

5) As a solution of this task, deliver:

Source code of parallel sorting

Table of measured times

Parallelization of sorting

Possible strategies: (more difficult)

A: 1) Assign to each thread a block of lines to be sorted

2) Sort each block independently on each thread (1st stage)

3) Combine the sorted blocks serially by one thread (2nd stage)

Advice: If the number of lines to sort is not a multiple of the number of threads, you can add few huge numbers to the list. These will remain at the end of the list when sorted and will not be printed.

Of course you can use a condition to handle sublists, which become empty.

Parallelization of sorting

Possible strategies: A (more difficult)

- example of 2nd stage final sorting on a single thread
- animated in Powerpoint

thread		0	1	2	3
	index				
values	0	-990	-994	-995	-999
	1	-985	-975	-993	-996
	2	-970	-960	-980	-940

red – values which are currently being compared

blue – selected and printed values

values of variable **ind[thread]** indicating, from which **nthread** values the minimum number is selected for printing as next smallest number

thread	0	1	2	3
ind[thread]	0	0	0	0
	1	1	1	1
	2	2	2	2
	3	3	3	3

Parallelization of sorting

MPI

Possible strategies: (easier, recommended)

B: Read the data by a single thread and determine the lowest (**low**) and highest (**high**) values. Split this range equally to all threads and let each thread sort one of the disjunct intervals – the resulting list is then given by ordered sequence of sorted blocks

Example: If the data are in range 0..100, when parallelized on 4 threads threads would process values from intervals

me=0: 0<=a<=25

me=1: 25<a<=50

me=2: 50<a<=75

me=3: 75<a<=100

Resulting blocks (of unequal lengths!) can be communicated together using commands

MPI_Gatherv

MPI_Send a MPI_Recv, after sending the lenghs of blocks first

more difficult

or print to sorted.txt (with the help of MPI_Barrier, fopen "sorted.txt","a") directly by individual threads

easy

without the data being gathered to a single thread

C: Any other – originality and creativity is appreciated!!!

Parallelization of sorting

OpenMP

Possible strategies: B (easier, recommended)

```
while                                   ... reading of N values and finding minimum (low)
                                            and maximum (high) values
#pragma omp parallel private(i, me, Nme, b_me[...], c_me[...]) ... beginning
                                                                    of parallel region
{
for (i=1;i<=N;i++) {}; ... each thread searches through all values and
      choses only those, which fall into the appropriate interval. This for cycle is
      in parallel region but is not parallelized! Each thread goes through all
      values of i and selects only those which fall into their range of values.
 … sorting   …                          ... each thread sorts its Nme values
#pragma omp ordered   ... beginning of a region (within parallel
                          region) executed in ordered sequence of threads
{                                       ... other threads wait for their turn
  if(me==0) fp=fopen("sorted.txt",“w"); ... thread writes/appends its sorted
                                                                    values
      else  fp=fopen("sorted.txt","a");    ... to output file
  for (i=1;i<=Nme;i++)fprintf(...);     ... printing of sorted values
close(fp);
}                                       ... end of ordered region within parallel region
}                                       ... end of parallel region
```

Useful hints

Logical AND - &

```
if((a<b)&(c<d)) {...}
```

Logical OR - |

```
if((a<b)|(c<d)) {...}
```

Jump to a label

```
if(a<b)goto label1;
...
label1: ;
```

Opening a file in append mode

Standard writing from the beginning of a file (overwriting)

```
fp=fopen("sorted.txt","w");
```

Opening a file for writing to the end of file (appending)

```
fp=fopen("sorted.txt","a");
```

Useful hints

Sending of only part of an array

- Initial index lze can be specified by an argumentem, which is address of the desired first element

Examples

- Sending of first 5 elements of array b

```
MPI_Bcast(&b[0],5,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

- Sending of 5 elements of array b with indices 7-11

```
MPI_Bcast(&b[7],5,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

This ‘trick’ can be used in all commands

- e.g. MPI_Scatter, MPI_Gather, MPI_Send, MPI_Recv

Useful hints

The writing of individual threads in order of their rank can be done in MPI using the already detailed sequence (e.g. the lecture on input/output)/

In OpenMP, ordered execution of a part of parallel region can be enforced by #prama omp ordered

More synchronization constructs

Sun microsystems

The enclosed block of code is executed in the order in which iterations would be executed sequentially:

```
#pragma omp ordered
{<code-block>}
```

May introduce serialization (could be expensive)