Parallel Programming Assignment 3

Nils Bäckström, Jonathan Löfgren, Anton Sundin

March 5, 2015

1 Problem description

When sorting a uniformly distributed list with n elements $e \in [0, 1]$, one way of sorting the list in parallel is to divide the elements into k buckets b_i and sort the buckets separately on different threads with any fast sorting algorithm. For example, bucket b_0 contains elements $e \in [0, h)$, b_1 contains elements $e \in [h, 2h)$ and bucket b_{k-2} contains elements $e \in [1 - h, 1]$ where h is interval size of the unsorted elements divided by the number of buckets. Sorting the buckets is a perfectly parallel task and after they are sorted, putting them together to get the full sorted list is easy. Assuming a uniform distribution of elements, this works well in parallel because the work load is evenly distributed between the buckets, since on average the buckets will contain the same number of elements.

If, however, the list is not uniformly distributed, the buckets will not hold the same amount of elements and the work load will thus not be the same for every bucket. To make the sorting as effective as possible in parallel, the challenge is to balance the work load instead of number of buckets between the threads. We need to somehow measure how much time it will take to sort each bucket and then distribute all the buckets among the threads so that the total time needed for each thread is as equal as possible.

In this paper we investigate a number of ways to distribute the work load when using OpenMP for the parallel implementation.

2 Implementation

When using a list of normally distributed numbers (with mean 0 and standard deviation 1) we choose a main interval $[-l, l]$ with buckets $b_i, i = 1, 2, ..., k-2$ containing equally large sub-intervals. To fit the entire normal distribution we also have bucket b_0 for $(-\infty, -l)$ and b_{k-1} for (l, ∞). A fixed value of $l = 3$ was used for all the following experiments, although this should ideally be related to the number of buckets and the length of the list. After dividing all the elements in the list into their appropriate bucket, the next step is to sort all the buckets in parallel as efficiently as possible. Following are the different methods investigated to do this.

2.1 OMP schedule

For this method we simply sorted all the buckets in a for loop which was parallelised with OMP by a *parallel for* using dynamic scheduling with chunk size 1. Static and guided scheduling was also investigated but dynamic was faster in our experiments. Using a small chunk size increases the overhead but not enough to cancel out the benefits of the improved balancing in our case. The implementation was very easy to carry out since it just needs one compiler directive.

2.2 OMP tasks

What this does is that one thread goes through the for-loop where the buckets get sorted and creates a queue of *tasks* which all the other threads (and also itself when its done creating the

tasks) can start working on. This is quite similar to schedule(dynamic, 1) since every time a thread is done with a task, it will start up with a new one. This implementation was also quite easy to implement since OMP makes it very easy to create tasks, and the scheduling of the tasks to the threads are then carried out automatically by OMP.

2.3 Bin-pack

In this solution, each thread will be seen as a bin and the task is to manually balance the work load between the bins by distributing the buckets in a smarter way. For approximating the work load of the buckets we use $n \log n$ since each bucket is sorted using quicksort since this algorithm uses time complexity $n \log n$ where n is the number of elements in the list. The buckets are then moreover sorted by descending work load and are assigned to the bins in this order. The buckets in sorted order are always assigned to the bin with least total work load. This way the work load needed tends to balance out among the bins since the smallest buckets are always assigned absolutely last. There are more exact ways to solve the bin-pack problem but they can be tedious to perform. This implementation of solving the bin-pack problem approximately takes imperceptible time in comparison with the actual sorting and with enough number of buckets we confirmed that the balance turns out really good.

3 Results

All the times reported are only the time needed to sort the buckets in parallel, since the other parts of the program were the same regardless of the load balancing method used.

Load balancing	8 Threads	16 Threads	32 Threads
OMP schedule	2.0973 s	1.0721 s	0.6956 s
OMP tasks	2.0771 s	1.0682 s	0.6911 s
Bin-pack	2.0943 s	1.0920 s	0.7391 s

Table 1: Timings for sorting 100 million elements with 1000 buckets.

Load balancing	8 Threads	16 Threads	32 Threads
OMP schedule	1.7037 s	0.8651 s	0.5064 s
OMP tasks	1.7020 s	0.8711 s	0.5882 s
Bin-pack	1.6898 s	0.8537 s	0.4959 s

Table 2: Timings for sorting 100 million elements with 10000 buckets.

3.1 Run-time system

The results of the run-time system algorithm are not comparable to the other algorithms since the number of threads used vary with the number of assigned buckets. With 1000 buckets, the sorting time reached 1.225 seconds.

3.2 Bin-pack balancing

The timings of the individual threads is the best measure we have for the performance of the bin pack algorithm. When using 100 million elements in the list and 32 threads, the time difference between the fastest and the slowest thread were 1.5% for 10000 buckets and 10% for 1000 buckets.

4 Conclusion

The actual sorting time is very similar between the algorithms presented in tables 1 and 2. This indicates that they all manage to balance the loads equally well.

4.1 Schedule / Task

We think that the overhead from creating tasks or schedule dynamically is negligible in most cases since the work done in each task/dynamically scheduled part is quite large (quick-sorting 10^4 to 10^5 elements). However, when having a very large number of buckets, this overhead may start to cost time.

4.2 Bin-pack

With "enough" amount of buckets (related to number of threads and list length) we can achieve very good load balance with the $n \log n$ approximation of estimated load per bucket. When using 1000 buckets we only have slightly worse performance than the OMP dynamic scheduling but for 10000 buckets we seem to have better performance. This could be because when the number of buckets increase the overhead of assigning them dynamically to threads become a factor. Our algorithm gets rid of this overhead with the cost of doing some balancing before all the sorting.

4.3 Run time system

This parallelisation performs worse than all other algorithms using 16 or more threads. This is a lazy and inefficient way to utilize the buckets. Since the need for more buckets increases with the number of elements, so does the need of more threads with this algorithm and this becomes a problem as the maximum number of available threads is approached and severely cripples the potential of this sorting algorithm.

5 Appendix

The files needed to run the sorting algorithm are the following.

1. quicksort.c
2. bucketsort.c
3. bucketsort.h
4. Makefile