CS515 Parallel Programming: Assignment #3

Due on May 18th, 2016 at 11:59pm

Jingke Li Spring 2016

Konstantin Macarenco

Implementation details

Main logic of the Assignment was implemented according to the given specifications. There are few language specific details,
First memory management and buckets implementation

```
struct Bucket {
    int size;
    int maxSize;
    int *elements;
    int pivot;
};
```

Where size, is the current number of elements added into the bucket, and maxSize is the maximum size of the elements array. Since size of each bucket is unknown all buckets start is size = 2, it is expanded as bucket overflows.

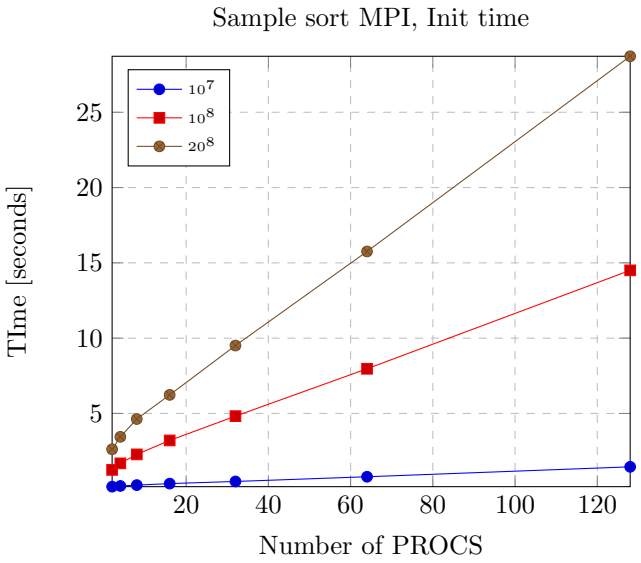
Performance Analysis

The assignment was testes on the linuxlab cluster, with the provided set of hosts, with one exception: machines "kororaa, scanner and snares", were not available at the time of testing. It total 56 machines were used. The program was executed against three randomly generated (by provided datagen) sets: $10^7, 10^8,$ *and* 20^8. The size of data sets is limited by "CAT disk quota", so I could't generate any larger data size. Each set wast tested with $2, 4, 8, 16, 32, 64, 128$ procs, distributed among the host by "mpirun" routine. Wall clock was measured by "rank 0" and set of check points, where all procs were synchronized by MPI_Barrier(), and validated by running the program in linux "time" utility without any additional synchronization. Use of additional sync routines added in average $\%2 - \%3$ delay.

The following time intervals are measured:

1. Total run time.
2. Data set read time.
3. Sort time.
4. Initialization time (division of data among buckets).
5. Time to send all gathered info to participating threads.
6. And total IO time.

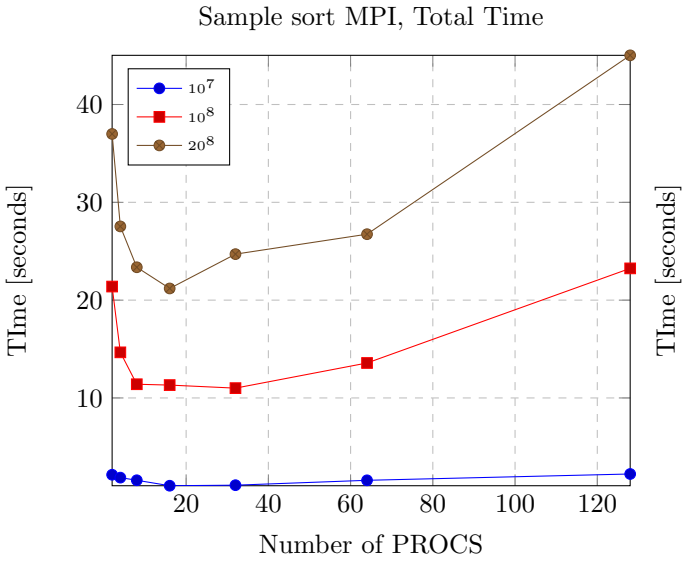
Test results showed similar to OMP quicksort picture, with sweet spot procs number = 16 for all data sets. Main bottleneck is data initialization, since data distribution was previously unknown, variable size buckets were implemented, with use of realloc(). According to realloc documentation, if it is unable to find appropriate chunk of memory, new memory is allocated with followed data move.

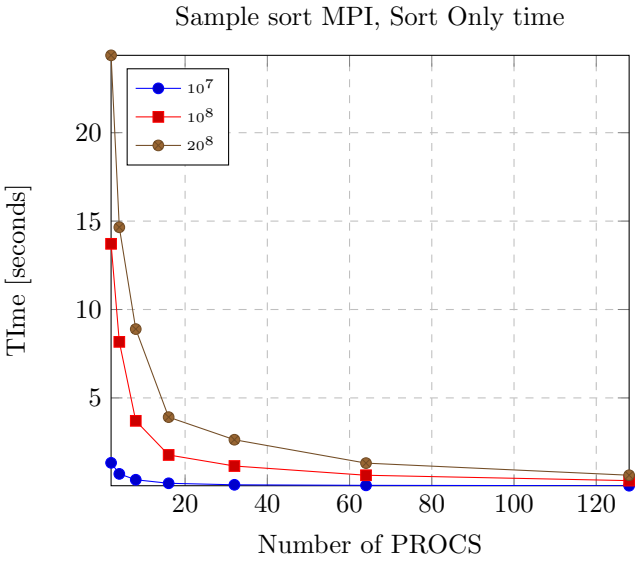


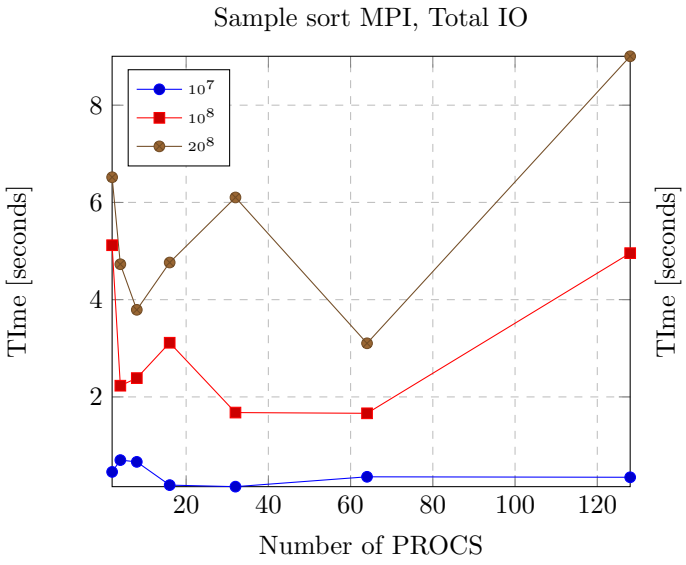
Although it is not clear for me why this operation was the slowest performer, on larger number of procs, since it is always executed only by the host machine I expected init time to be constant. Initialization time was linearly increasing with number of participating procs, i.e. only number of procs affected init performance but not the data set size.

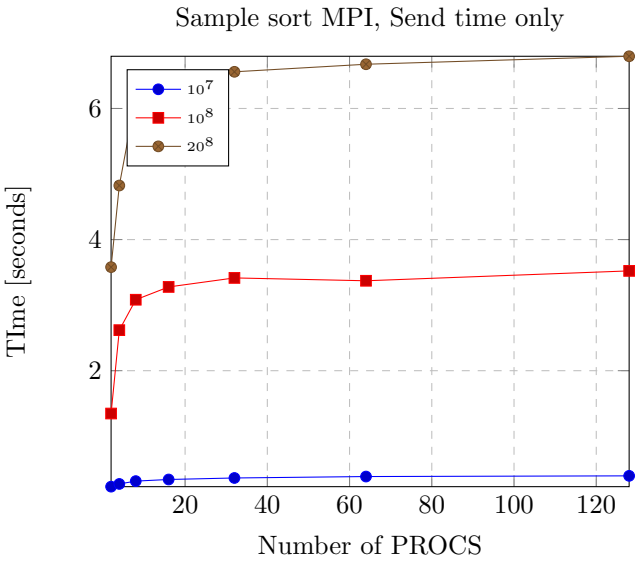
Other intervals behaved as expected:

- Sort stage performance increase is log(n), where n is number of procs.
- Data send time decrease is log(n), where n is number of procs (more messages however each message has smaller size).
- File read time is minuscule compared to all other operations (about %0.7 in average for all cases).
- Total IO (read data set + save results), is mostly affected by the save stage. Save stage pattern is similar to the patter of Total Time, with the sweet spot being nporcs = 64. This was expected since larger number of procs will have more data movements across network.

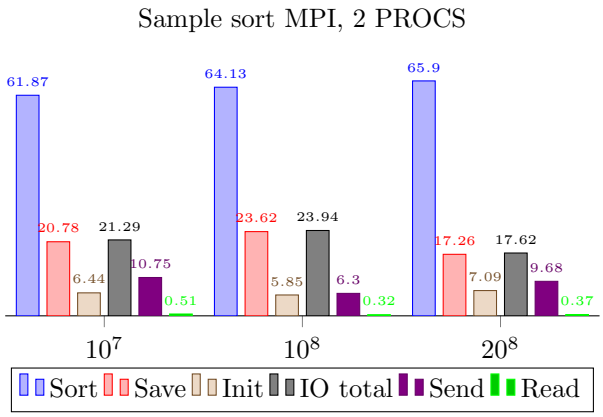


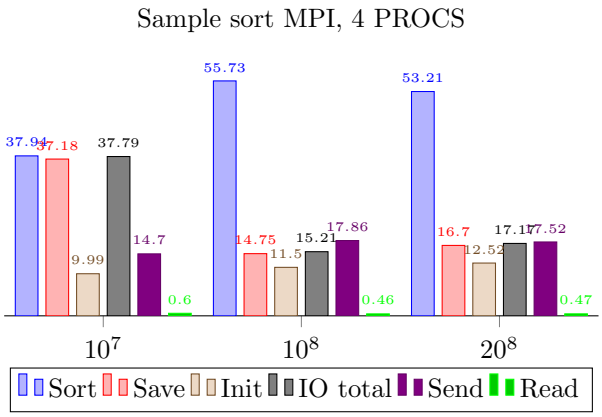




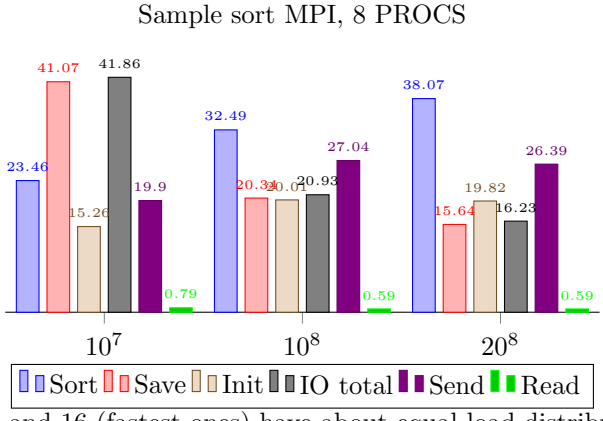


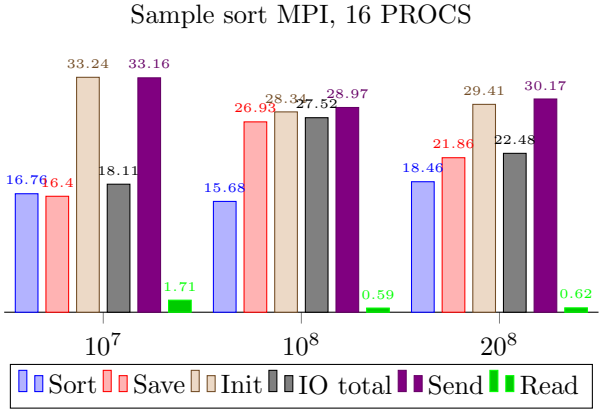
Following bar charts show normalized (to total time = %100) distribution of different program intervals (in percent), each chart represents different number of procs, in the order of $2, 4, 8, 16, 32, 64, 128$





Two and four spend most of the time $\approx$ %60 in sorting.

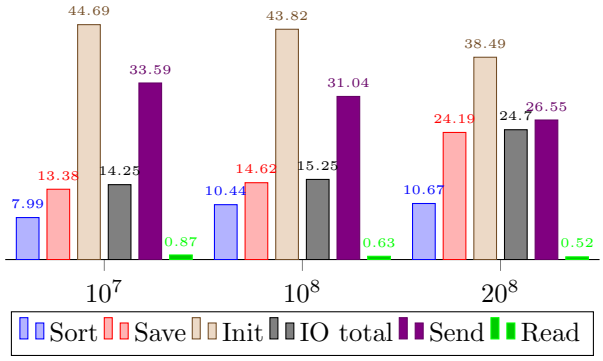




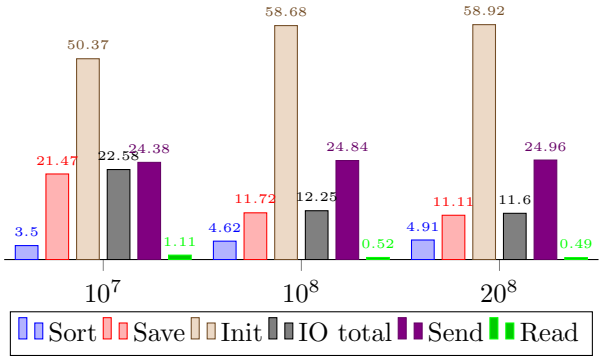
8 and 16 (fastest ones) have about equal load distribution.

All the rest spend most of the time in initialization stage

Sample sort MPI, 32 PROCS



Sample sort MPI, 64 PROCS



Sample sort MPI, 128 PROCS