High-performance Computing, Autumn 2024

Robert Buj

October 15, 2024

Question 1

Steps:

1. compile the source code

   ```
   make
   ```

2. queue the jobs using the bash script s2.sh

   ```
   for p in app app2; do ./s2.sh -n $p; done | sh
   ```

3. get gnuplot data

   ```
   for p in app app2; do python3 s4.py -n $p > $p.dat; done
   ```

4. get gnuplot figure

   ```
   gnuplot executiontime.gnu
   ```

```
qsub -N app_10_1 -v name=app -v size=10 template.sge
qsub -N app_10_2 -v name=app -v size=10 template.sge
...
qsub -N app_1500_10 -v name=app -v size=1500 template.sge
qsub -hold_jid "app_*" -N app_end -v name=app -cwd ./s3.sge
```

Figure 1: ./s2.sh -n app

```
#!/bin/bash
while getopts n: flag
do
    case "${flag}" in
        n) name=${OPTARG};;
    esac
done
for NUM in 10 50 100 500 1000 1500
do
    for SAMPLE in {1..10}
    do
        echo qsub -N ${name}_${NUM}_${SAMPLE} -v name=${name} \
-v size=${NUM} template.sge
    done
done
echo qsub -hold_jid \""${name}_*"\" -N "${name}_end" \
-v name=${name} -cwd ./s3.sge
```

Figure 2: file s2.sh

```
#!/bin/bash
#£ -cwd
#£ -S /bin/bash
#£ -o £JOB_NAME.£JOB_ID.out
#£ -e £JOB_NAME.£JOB_ID.err

echo host `hostname`
echo timestamp `date +%s`
time ./${name} ${size}
```

Figure 3: file template.sge

The scheduler will launch the job s3.sge after running all jobs. The script s3.sge reads the output files from ten executions and it generates a csv file for each application argument. The csv file has the following fields:

- host name that received and ran the job

- execution start of the job (UNIX timestamp)
- execution time of the job (seconds)

```
#!/bin/bash
#£ -cwd
#£ -S /bin/bash
#£ -o £JOB_NAME.£JOB_ID.out
#£ -e £JOB_NAME.£JOB_ID.err

for NUM in 10 50 100 500 1000 1500
do
    FILE=${name}_${NUM}.csv
    echo "hostname,timestamp,real" > $FILE
    for SAMPLE in {1..10}
    do
        OUT_FILE=`ls ${name}_${NUM}_${SAMPLE}.*.out`
        ERR_FILE=`ls ${name}_${NUM}_${SAMPLE}.*.err`
        HOST=`grep host ${OUT_FILE} | cut -d' ' -f2`
        TIME=`grep timestamp ${OUT_FILE} | cut -d' ' -f2`
        REAL=`grep real ${ERR_FILE} | cut -f2`
        REAL_TMP=(${REAL//m/ })
        REAL_MIN=${REAL_TMP[0]}
        REAL_TMP=(${REAL_TMP[1]//./ })
        REAL_SEC=${REAL_TMP[0]}
        REAL_TMP=(${REAL_TMP[1]//s/ })
        REAL_MIL=${REAL_TMP[0]}
        TOTAL=`echo "(${REAL_MIN} * 60) * 1000" | bc`
        TOTAL=`echo "${TOTAL} + (${REAL_SEC} * 1000)" | bc`
        TOTAL=`echo "${TOTAL} + ${REAL_MIL}" | bc`
        echo $HOST,$TIME,`echo "scale=3; ${TOTAL} / 1000" | bc` >> $FILE
    done
done
```

Figure 4: file s3.sge

The following script gets the gnuplot data from the csv obtained in the previous steep, and it calculates the mean execution time and its standard deviation for a certain argument (10, 50, 100, 1000, 1500).

```
# Import Pandas library
import pandas as pd
import getopt, sys

argumentList = sys.argv[1:]

options = "n:"
long_options = ["Name"]

try:
    arguments, values = getopt.getopt(argumentList, options, long_options)
    for currentArgument, currentValue in arguments:
        if currentArgument in ("-n", "--Name"):
            name = currentValue

except getopt.error as err:
    print (str(err))

dim_list=[10, 50, 100, 500, 1000, 1500]
for dim in dim_list:
    url=str(name)+'_'+str(dim)+'.csv'
    df=pd.read_csv(url)
    mean_value=df['real'].mean()
    std_value=df['real'].std()
    print(str(dim)+' '+f"{mean_value:.3f}"+' '+f"{std_value:.3f}")
```

Figure 5: file s4.py

REF: Use Pandas to Calculate Stats from an Imported CSV file

Question 2

The results are summarized in Figure 12. The plot below displays the execution time with -O3 optimization and without. The optimized binary got better results.

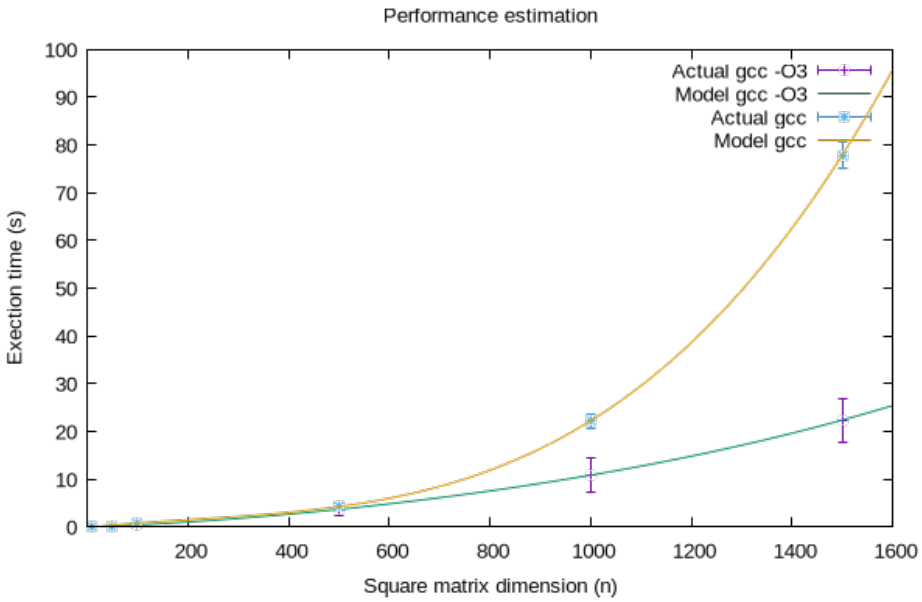

Figure 6: Time needed to run the jobs.

Table 1 summarize the results obtained. At the beginning the execution time grows linearly and later exponentially.

Parameter	Avg(s)	Std. deviation(s)	Avg(s) -O3	Std. deviation(s) -O3
10	0.040	0.031	0.055	0.028
50	0.253	0.158	0.157	0.175
100	0.860	0.097	0.473	0.258
500	4.243	0.854	3.692	1.320
1000	22.225	1.485	10.839	3.606
1500	77.853	2.800	22.375	4.534

Table 1: Summary of results.

Loop reordering (L.O.) improves execution time over all executions. The loop reordering binary got better results for large matrices: $N \geq 500$ with -O3 optimization flag and $N \geq 100$ without -O3 optimization flag. The results obtained using the -O3 optimization are summarized in Figure 7.

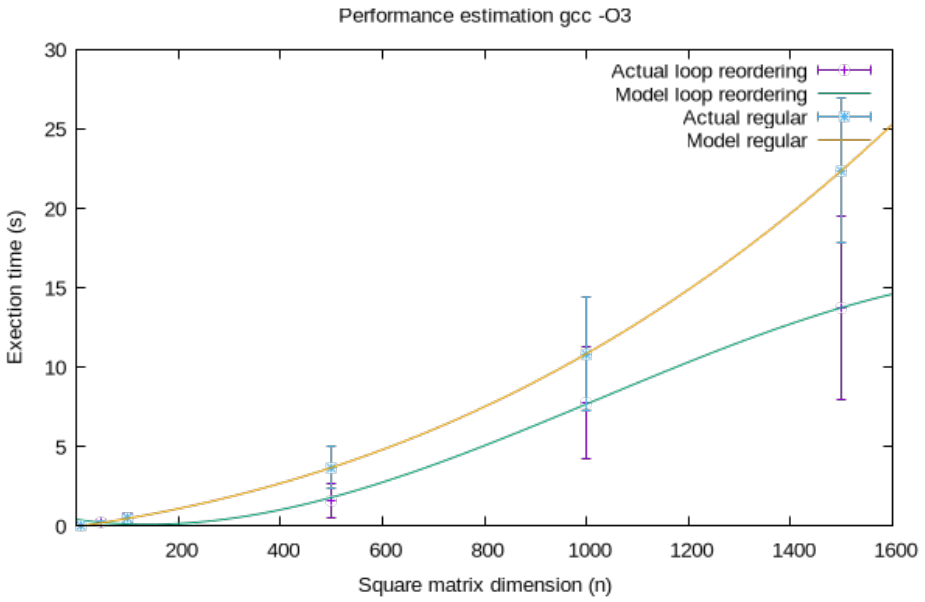

Figure 7: Execution time loop reordering vs regular (gcc -O3).

Parameter	Avg(s)	Std. deviation(s)	L.O. Avg(s)	L.O. Std. deviation(s)
10	0.055	0.028	0.059	0.032
50	0.157	0.175	0.255	0.176
100	0.473	0.258	0.530	0.310
500	3.692	1.320	1.605	1.064
1000	10.839	3.606	7.761	3.481
1500	22.375	4.534	13.723	5.794

Table 2: Summary of results for loop reordering vs regular (gcc -O3).

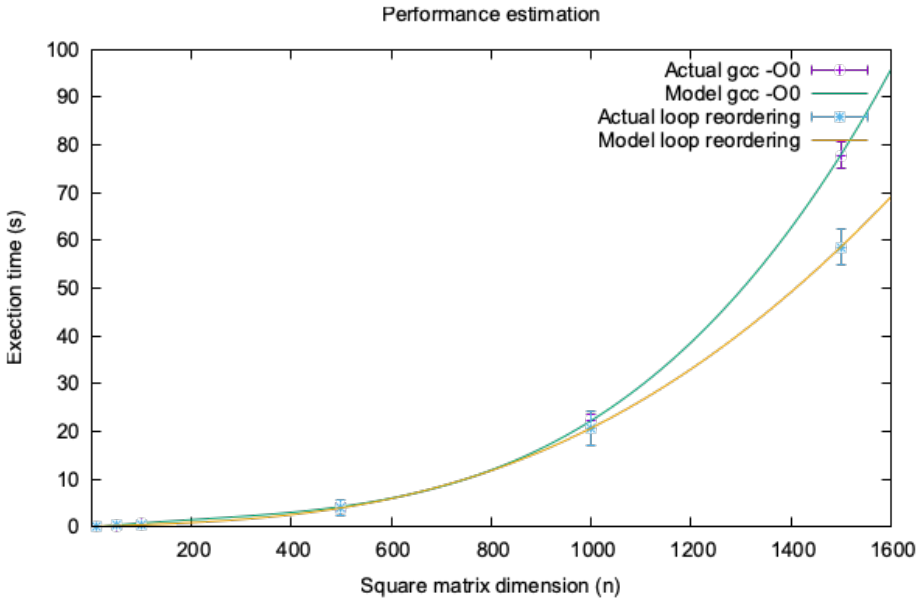

Figure 8: Execution time loop reordering vs regular.

Parameter	Avg(s)	Std. deviation(s)	L.O. Avg(s)	L.O. Std. deviation(s)
10	0.040	0.031	0.065	0.029
50	0.253	0.158	0.363	0.145
100	0.860	0.097	0.395	0.286
500	4.243	0.854	3.991	1.578
1000	22.225	1.485	20.678	3.661
1500	77.853	2.800	58.600	3.798

Table 3: Summary of results for loop reordering vs regular

```
for (i=0; i<N; i++) {
      for(j=0; j<N; j++) {
              for (k=0; k<N; k++) {
                      c[i][j] += a[i][k] * b[k][j];
              }
      }
}
```

Figure 9: regular matrix multiplication

```
for (i=0; i<N; i++) {
      for(j=0; j<N; j++) {
              for (k=0; k<N; k++) {
                      c[i][k] += a[i][j] * b[j][k];
              }
      }
}
```

Figure 10: reordered matrix multiplication, less cache misses

Question 3

It seems that the execution time of func(N) is proportional to the application argument (N) and it's random.

```
if(argc<2){
      printf("Usage: %s matrix_size\n", argv[0]));
      exit(-1);
}
N = abs(atoi(argv[1]));
func(N);
```

Figure 11: file func.c

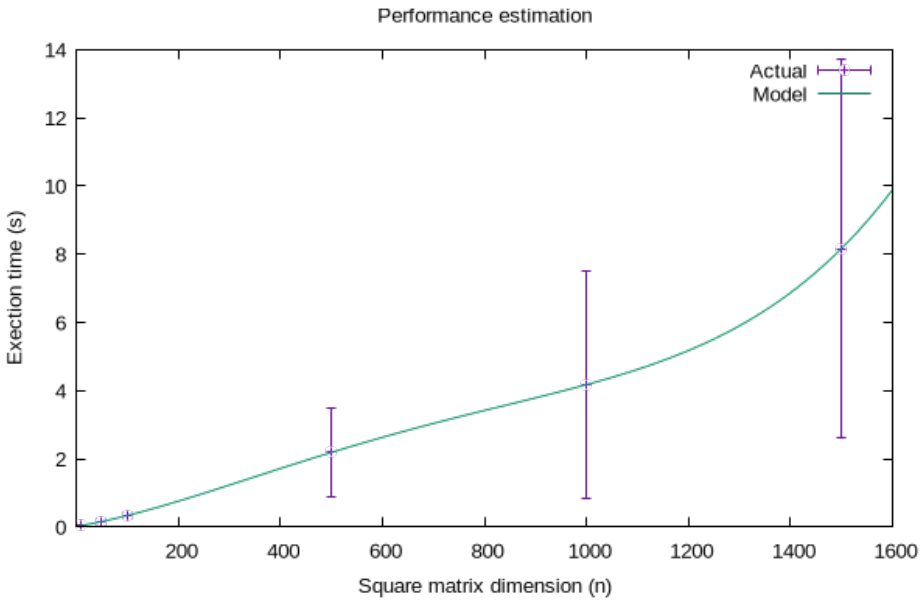

Figure 12: Time needed to run func.

```
gcc func.c lib.o -o func
./s2.sh -n func | sh
python3 s4.py -n func > func.dat
gnuplot -e "datafile='func.dat'; outputname='func.png'" et.gnu
```

Figure 13: commands to reproduce the plot

Question 4

FCFS

		Time																													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
CPU	1	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J8	J10	J10	J11	J11	J11	J11	J16	J16	J17	J17	J17
	2	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J8	J10	J10	J11	J11	J11	J11	J16	J16	J17	J17	J17
	3	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J8	J10	J10	J12	J12	J12	J12	J16	J16	J17	J17	J17
	4	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J8	J10	J10	J12	J12	J12	J12	J16	J16	J17	J17	J17
	5	J1	J1	J1	J1	J1	J1	J1	J1	J5	J5			J6	J6	J6	J9	J9	J9	J9			J10	J10	J13	J13	J13	J13	J16	J16	J17	J17	J17
	6	J1	J1	J1	J1	J1	J1	J1	J1					J6	J6	J6	J9	J9	J9	J9			J10	J10	J14	J14	J14				J17	J17	J17
	7	J1	J1	J1	J1	J1	J1	J1	J1					J6	J6	J6	J9	J9	J9	J9			J10	J10	J15	J15	J15	J15	J15	J15	J17	J17	J17
	8	J1	J1	J1	J1	J1	J1	J1	J1					J6	J6	J6	J9	J9	J9	J9			J10	J10							J17	J17	J17
	9		J2	J2	J3	J3	J3	J3	J3	J3	J3	J3		J6	J6	J6							J10	J10									
	10		J2	J2										J7	J7	J7	J7						J10	J10									

Table 4: FCFS.

EASY-backfilling

		Time																									
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
CPU	1	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J8	J10	J10	J17	J17	J17
	2	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J8	J10	J10	J17	J17	J17
	3	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J8	J10	J10	J17	J17	J17
	4	J1	J1	J1	J1	J1	J1	J1	J1	J4	J4	J4	J4	J6	J6	J6	J8	J8	J8	J8	J8	J8	J10	J10	J17	J17	J17
	5	J1	J1	J1	J1	J1	J1	J1	J1	J9	J9	J9	J9	J6	J6	J6	J11	J11	J11	J11	J16	J16	J10	J10	J17	J17	J17
	6	J1	J1	J1	J1	J1	J1	J1	J1	J9	J9	J9	J9	J6	J6	J6	J11	J11	J11	J11	J16	J16	J10	J10	J17	J17	J17
	7	J1	J1	J1	J1	J1	J1	J1	J1	J9	J9	J9	J9	J6	J6	J6	J12	J12	J12	J12	J16	J16	J10	J10	J17	J17	J17
	8	J1	J1	J1	J1	J1	J1	J1	J1	J9	J9	J9	J9	J6	J6	J6	J12	J12	J12	J12	J16	J16	J10	J10	J17	J17	J17
	9		J2	J2	J3	J3	J3	J3	J3	J3	J3	J3	J3		J6	J6	J6	J14	J14	J14		J16	J16	J10	J10			
	10		J2	J2			J5	J5	J7	J7	J7	J7	J13	J13	J13	J13	J15	J15	J15	J15	J15	J15	J10	J10			

Table 5: EASY-backfilling - backfill does not allow delaying an existing queued job.

Resource utilization

Scheduling Algorithm	Resource Utilization (%)
FCFS	77.5
EASY-backfilling	95.38

Table 6: Resource utilization (%) for FCFS and EASY-backfilling.