Performance Evaluation of Parallel Programs

Master M1/M2 MOSIG, Grenoble Universities
Arnaud Legrand, Sascha Hunold

November 2011

1 On Parallel Sorting

In this assignment you will examine the performance of a parallel sorting algorithm.

Let's start with the sequential case. The program that you are going to investigate is based on the Quicksort algorithm. For a given size, a vector is created and filled with random numbers. Then `qsort()` from the standard C library is called to sort the array.

In the parallel case, we also create an array of a specific size and insert random integers at each array position. Then, depending on the user's choice several threads are started. Each thread is responsible for sorting one part of the array. The array is chunked into equal sized shares. When all threads have finished the sorting, one thread will collect the results. This thread creates a new array and fills the results from left to right. For an array position i the collecting thread will check which of the p other threads has the smallest value. This value is removed from this thread p_j and inserted into the position i of the result array. This procedure is continued until all elements have been inserted into the result array.

Models

Develop a mathematical model of the execution time of this parallel algorithm. The model should depend on n, the input size, and p the number of processors. We assume that all operations of the sort take time α, the operations for the merge take time β, and the operations for managing (create, join) the threads take time γ.

$$T(n,p) = T_{sort}(n,p) + T_{merge}(n,p) + T_{manage}(n,p)$$

Create a similar model for the sequential version!

- Pick reasonable numbers for n, p, and α and plot the expected run-time of the sequential version of the code and the parallel version. For example, one can plot the execution time in relation to the number of processors or the size of the array.
- Could you think of a way to improve the performance of the parallel sort?

2 Practical exercise - source package

The programs for this week's exercise are in the archive `sort-1.0.tar.gz`. You can unpack them like this:

```
tar xvfz sort-1.0.tar.gz
```

You can compile the programs using the following commands:

```
./configure
make
```

The test target in the Makefile produced a series of executions that are used to illustrate how the programs provided.

3 Some words on measuring time

There are two ways of measuring the time spent in a program.

One is the function

```
int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

and the other is the function

```
int getrusage (int who, struct rusage *r_usage);
```

With `gettimeofday()` one can measure the absolute time between two distinct points in time. The functions returns the number of microseconds that have passed since midnight of January 1, 1970. The problem of using `gettimeofday()` to measure the time of a code segment is that it includes the time that other processes have been scheduled to the CPU. Thus, this time is imprecise if many concurrent processes are running.

The function `getrusage()` can be used to retrieve information about the resources utilized by the current process. The structure returned by this function contains the time that the process has spent in user and in system mode.

4 Evaluation of the parallel performance

The sort package contains two binaries. One is a sequential sort (`sort_seq`) and the other is a parallel version of a sort algorithm (`sort_par`).

You can start the programs like this

```
./sort_seq -n 10 -r

./sort_par -n 10000000 -p 2 -r
```

To see the different options that can be passed to both programs you can call each program using the `-h` flag.

```
./sort_seq -h
Usage :
./sort_seq --nsize number [ --parallelism number ] [ --verbose]
[ --rusage ] [ --seed number ] [ --help ]
```

The meaning of the parameters are:

- `nsize` size of the vector to be sorted
- `parallelism` number of threads to be started
- `seed` define a seed for the random number generator
- `verbose` print sorting results
- `rusage` print user and system time obtained from `getrusage()`

On Linux, you can check how many processors your system has by executing the command:

```
cat /proc/cpuinfo
```

Exercise 1

- Get familiar with the programs and play with different parameter settings and try to reason about the different outcomes!
- Try to increase the number of processors while keeping the size of the vector fixed. Also experiment with different sizes of the input vector for the sequential and the parallel case, and for the parallel version also with the number of threads.
- Compare the run-times of the sequential version to the parallel version!
- Compare the execution time obtained from `gettimeofday()` to the times obtained from `getrusage()`!
- What happens in the case of oversubcribing the system (using more threads than there are processors in the machine)?
- Execute one program with a specific parameter setting several times and observe the run-time.
- How noisy are your results? What can you do to quantify the dispersion of the results and what can you do to strengthen the validity of your experiments?

Exercise 2

Record the outcome of a series of experiments using the parallel sort. The following parameters should be used in this evaluation:

$$n = \{10^2, 10^5, 10^8\}$$
$$p = \{1, 2, 3, 4\}$$

You should fix the seed to a pre-defined value. Measure the run-time (user+system) of the program for each parameter setting! Repeat each experiment **several** times!

To help you performing the experiments you can modify the bash script `run_exp1.sh` of the sort package.

Now, we want to visualize the recorded data of the experiments. First, compute the mean run-time, the standard deviation (of the sample) and the standard error of the mean for each experiment. Then plot for each number of processors the average run-time with the standard error of this run-time! Thus, the x-axis holds the number of processors and the y-axis holds the run-time. The plot should show one line for every vector size.

$$\begin{aligned} \sigma &= \sqrt{\frac{1}{n-1}\sum_{1}^{n}(X_i - \overline{X})^2} \\ SD_{\overline{X}} &= \frac{\sigma}{\sqrt{n}} \end{aligned} \tag{1}$$

95% confidence interval

$$\left[\overline{X} - \frac{2\sigma}{\sqrt{n}}, \overline{X} + \frac{2\sigma}{\sqrt{n}}\right] \tag{2}$$

Exercise 3

In this exercise, you will examine the data in a different manner. Instead of displaying the run-time vs. the number of processors, in this plot we compare the run-time (y-axis) for every vector size (n on x-axis). This exercise will also include the run-time that was measured for the sequential version of the code. The plot should contain a line for the sequential version of the sort and a line for each number of processors. We will use

$$n = \{10^2, 10^5, 10^8\}$$
$$p = \{1, 2, 3, 4\}$$

in this plot.

Where is the crossover point between the sequential version of the sort and the parallel versions. In addition, observe the overhead of the parallel version utilizing only 1 processor/thread compared to the pure sequential version!

Exercise 4

Compute the speedup of your parallel program for each number of processors $p \geq 2$! The speedup is computed by dividing the sequential time for a given vector size by the corresponding parallel time ($S(n,p) = \frac{T_{seq}(n)}{T_{par}(n,p)}$). What is the theoretical maximum speedup? The proper way of computing the speedup is by using the best sequential version of the algorithm. So, using the run-time of the parallel program with one thread to compute the speedup improves the computed speedup but "it would be a kind of cheating".

Plot the speedup for

$$
\begin{aligned}
n &= \{10^2, 10^5, 10^8\} \\
p &= \{2, 3, 4\}
\end{aligned}
$$

Thus, the a-axis defines the number of processors while the y-axis holds the computed speedup.

Exercise 5

Plot the same data as in Exercise 4 but using parallel efficiency. The parallel efficiency is defined as

$$E(n,p) = \frac{T_{seq}(n)}{T_{par}(n,p) \cdot p} \qquad .$$

What can you report about the efficiency of your parallel sorting program?