Department of Computing Science
Umeå University March 14, 2022

Operating Systems - Assignment 2

Ansgar Lemke (ens21ale)

Abstract

A simple scatter-gather functionality will be implemented. The execution speed of the implementation will be compared using different parameters for the scatter-gather function calls to compute the same problem. This approach uses shared memory and processes to implement the functionality. Depending on the size of memory, used devices and the computational load of the tasks, there is an optimal number of processes to use.

The source code of this project as well as this report can be found at https://github.com/einansgar/scatter-gather.

Operating Systems
Supervisor: Monowar Bhuyan (monowar@cs.umu.se),
Sourasekhar Banerjee (sourasb@cs.umu.se)

1 Description

scatter-gather is an algorithm to split up a memory area so that multiple processes can work on the same computational problem and combine the results to one field of memory afterwards. MPI implements this functionality as MPI_Scatter_Gather, however, this solution aims to be as easy to use as possible. It's only dependencies are standard libraries.

1.1 Solution

The library is implemented in C99 which makes it available to many applications. With the used of shared memory but separated areas within for the different processes, questions of synchronization fall apart in most of the implementation. In the scattered phase, the processes work independent of each other and do not need any synchronization or communication which offers a good concurrent environment.

1.2 Design Decisions

Each process is assigned to a segment which corresponds to a certain area of the shared memory. The highest segment is assigned to the parent process. This information can be used to exit all other processes in the end to continue on the parent only.

One goal is to provide an interface to the user which grants full functionality while being relatively safe by hiding parts that are not relevant for end users. To transmit data from scatter to gather, a static struct is used internally. There are two functions available to be used by the user:

- scatter takes initialization data and parameters and provides a pointer to a location in shared memory for each process to work on.
- gather ends processes as soon as all of their children have died, except for the root process which has called scatter once. This one is provided with a copy of the shared memory after all the other processes have finished.

Between scatter and gather, multiple processes exist to compute tasks in parallel.

1.3 Assumptions

The spawned processes must only work on their own memory section, proc_memory. They must not change the location of the pointer because then their changes will not be written back. They must strictly remain inside their memory area, otherwise they would write either on memory of another process or even to an invalid location outside the shared memory area.

It is assumed that all processes take nearly the same amount of time, otherwise, the waiting might take up to the time the slowest process needs to finish plus the time to exit the others. In addition, is assumed that processes do enough work to justify the overhead by process creation and destruction.

1.4 Limitations

The amount of available data types is restricted to the basic C types which do not contain any pointers to other memory locations and have a fixed size, e. g. int, char. Inclusion of pointers or other more complex data structures would introduce the problem that these need to be copied to the shared memory as well while it is not known how exactly their structure looks like.

The scatter-gather calls cannot be nested, if scatter is called, gather must follow exactly one time. This greatly simplifies the structure because a lot of the functionality can be hidden from the user.

On multi-processor systems, the behavior of cache pages might reduce performance when data needs to be copied back again or adjacent memory areas might be synchronized as well out of the interpretation that most users might want to access these as well. The effects of false sharing might also decrease the execution speed. And of course, there is a maximum size of memory that can be used in RAM.

2 Examples

To demonstrate the functionality of scatter-gather, two computational problems will be executed with different amounts of processes. The programs will be executed on a Intel©Core^{TM} i7-7500U CPU @ 2.70GHz $\times$ 4. The results on a GPU are expected to be quite different.

2.1 Arithmetic Operations

On each segment of memory, a combination of set, scale, add, scale is executed ten times on an array of size 504000. Each number of processes is tested 100 times.

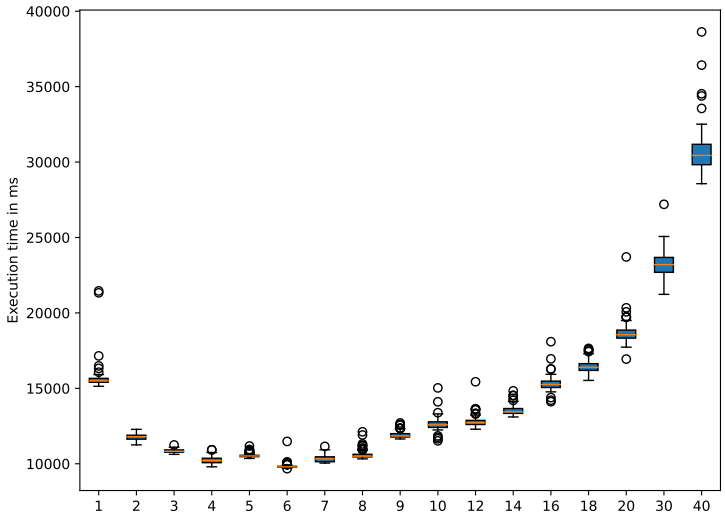

Figure 1: Execution times on arithmetic operations

2.2 Generating Random Numbers

The segment of process memory is being filled with random numbers. The operation is executed on an array of size 504000. Each number of processes is tested 100 times.

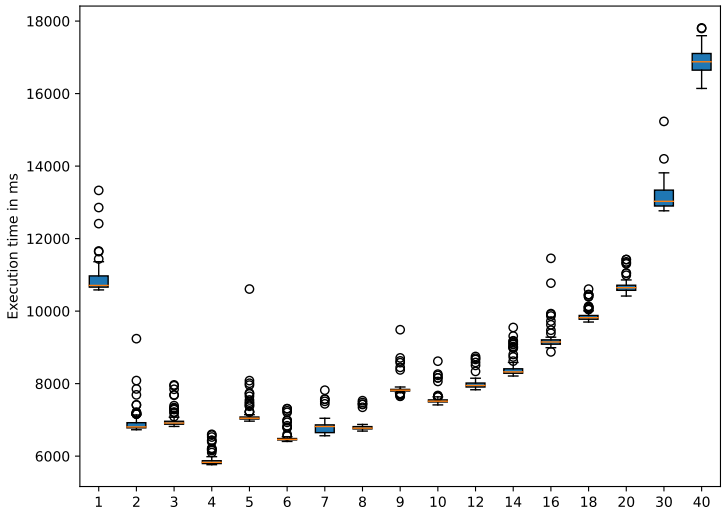

Figure 2: Execution times on generating random numbers

2.3 Discussion

While execution times decrease when using up to eight processes, the performance gets worse with more processes. Each fork introduces extra overhead. At some point, depending on the amount of work inside the forked processes, this overhead is greater than the gains from parallelization. In addition, the ressources for parallelization are constrained. Processor utilization reaches 100% even with a small amount of parallel processes.

As a consequence, the number of parallel processes should not be higher than neccessary. In the presented measurements the number should be between 2 and 16 with an optimum at 4 or 6.

3 Conclusions

In the chosen setup, creation of more than 10-20 processes results in even worse performance than a single process, marking out the importance of overhead, especially in the fork calls. Depending on the tasks, execution times can be reduced significantly.