Université libre de Bruxelles

Project 2

Parallel exhaustive search with progressive saves

Aldar Saranov, Przemyslaw Gasinski

Aldar.Saranov@ulb.ac.be
Przemyslaw.Gasinski@ulb.ac.be

INFO-F-404 Real-Time Operating Systems II (M-INFOS/F277)

Joël Goossens, Nikita Veshchikov

November 2016

Program description

Since we were developing a multi-process application first thing we had to think about was developing of a multi-process algorithm. Comparing of all possible pairs of hashes demanded partition of the search space. Partition was carried out by the first byte of the hash input. The process structure of the program is composed of N processes. One of them is a host-process. N-1 processes are worker-processes.

The host-process assigns search sectors for each of the worker-processes (Figure 1). Sector could be described as space $[SectorID, 0,0, \dots, 0,0] - [SectorID, 255,255, \dots, 255,254]$ for the first word. The second word is iterated in space $[Word_0, 0,0, \dots, 0,1] - [255,255,255, \dots, 255,255]$ i.e. all words which are larger than word1. Obviously there are 256 sectors (0-255) each of which should be assigned to a worker-process by the host-process. The partition is illustrated on Figure 2. The algorithm of worker-process is described on Figure 3.

Storing in memory such quantity of hashes for consequent paired comparison is impossible due to extremely large number of pairs; therefore we will have recalculate hashes for every new pair which provides extra processor load. Openssl SHA1 implementation was chosen as the one to be used in the software.

Concerning the low-level data storage we keep words and hashes in arrays of unsigned char since it is the essential byte word representation in C language.

To compare N LSB of two binary words we use an algorithm:

1. Compare all M full bytes last bytes using standard unsigned char comparing, where $M = \left\lfloor \frac{N}{8} \right\rfloor$.
2. The rest $K = N \bmod 8$ bits should be compared using $mask_k$ (e.g. $mask_3 = 00000111_2$). $Word_1[20 - M] \times mask_k == Word_2[20 - M] \times mask_k$?

Similarly we had to implement low-level incrementing of a binary word represented by unsigned char array.

Protocol transmission

Every worker-process can speak with host-process and vice versa. No other data exchange is conducted.

3 stages have been specified:

1. Stage 1 (assigning sectors).
    a. Host receives a sector request from any worker.
    b. Host sends sector data to the requesting worker. Worker starts computing. When sector is computed, worker requests another (to 1-a).
    c. When all sectors are distributed go to stage 2.
2. Stage 2 (stopping worker's computing).
    a. Host receives a sector request from any worker.
    b. Host sends to the requesting worker "stop" command.
    c. When sent to all the workers go to stage 3.
3. Stage 3 (runtime assembling).
    a. Worker sends to the host a message with his runtime.

All types of messages are described in the Table 1.

Performance

Fairness of work distribution

Fairness is ensured if all worked-processes are being executed roughly same time. For 16 processes and 2 byte search we have 45 seconds of execution for every of the processes, therefore we can state that work is distributed fairly.

Performance overestimating

We may firmly state that performance of 16 processors working for 5 hours is overestimated to conduct full-search on 16 bytes since 16 bytes means checking of 2^{128} options for linear algorithm which is even more than what we estimated in "INFOF405 Computer security project-1". Since the developed algorithm has $O(2^N)$ complexity, where N is the length of the search space the time required to process 16 bytes is even bigger. If we have B bytes search then the number of comparisons will be:

$$Comparisons = \frac{2^{8B}(2^{8B} - 1)}{2}$$

Unfortunately the practical limit of non-finishing the execution starts roughly at 3 byte search space (See results).

Difficulties

Message-less search space distribution

Initially we had an idea of distributing the search space without interacting via messages (processes were supposed to figure out their search space themselves by their id). All propositions had problem of uneven work distribution which is definitely not desired in multi-process application. Eventually we had to develop a protocol which has been described above.

Race condition at runtime assembling

We have encountered a problem of race condition when at stage 3. Many processes were trying to send to the host messages at the same time since initially we used MPI_ANY_SOURCE as source for host's MPI_recv() call. Resolved by receiving messages from every worked in order (of ID ascending) instead of expecting messages from MPI_ANY_SOURCE.

Keeping intermediate results

Since it is rational to store intermediate results for such long computations we had to develop a mechanism of doing so. Every worker rewrites its file by empty content in the beginning of the work and while meeting a new collision it appends it to that file. Therefore we ensure results storing even in case of killing of the program.

Results

The developed software was run on following configurations:

1. 1 byte, 8 LSB, 16 processes (runtime < 1 sec).

2. 2 byte, 24 LSB, 16 processes (runtime = 45 sec).
3. 3 byte, 64 LSB, 16 processes (runtime = 5 hours). Not all the space was checked.

Therefore the largest –b value with which the program will halt within 5 hours is 2 bytes.
A folder with the results is presented in the same archive.

Appendix

Figures

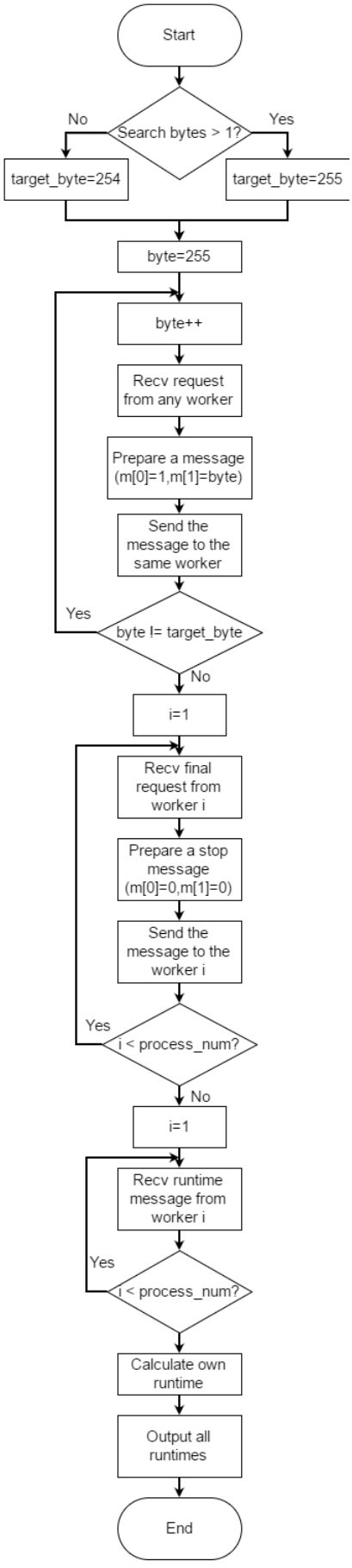

Figure 1. Host-process algorithm.

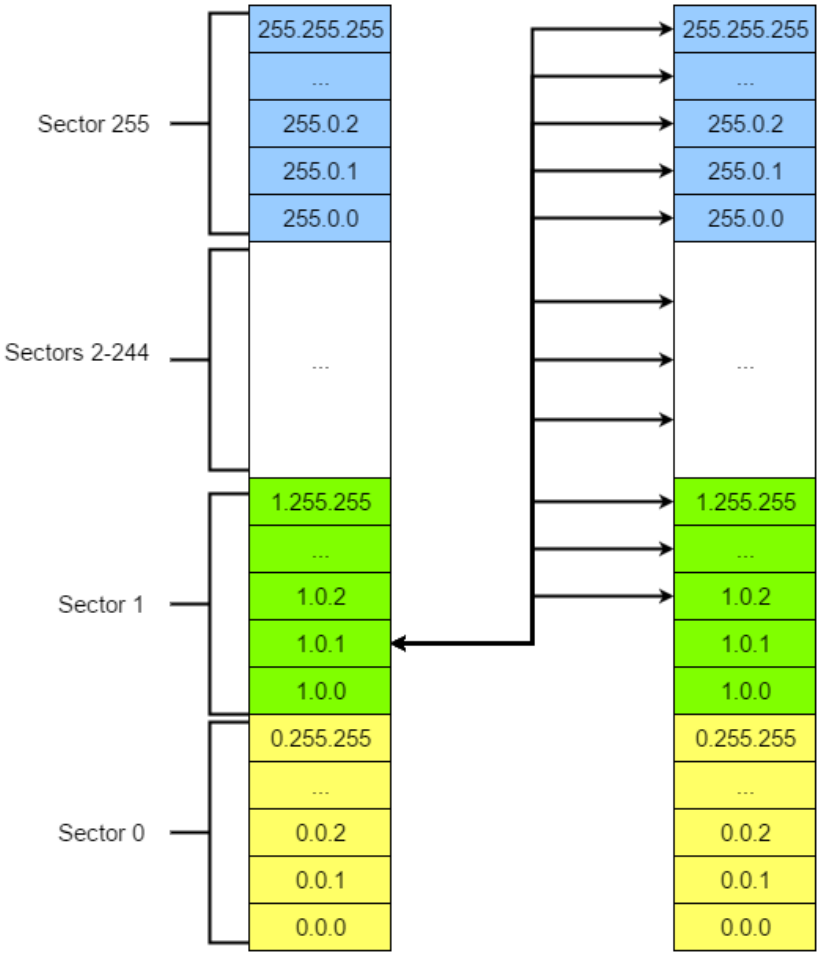

Figure 2. Sector distribution illustration for 3 byte search.

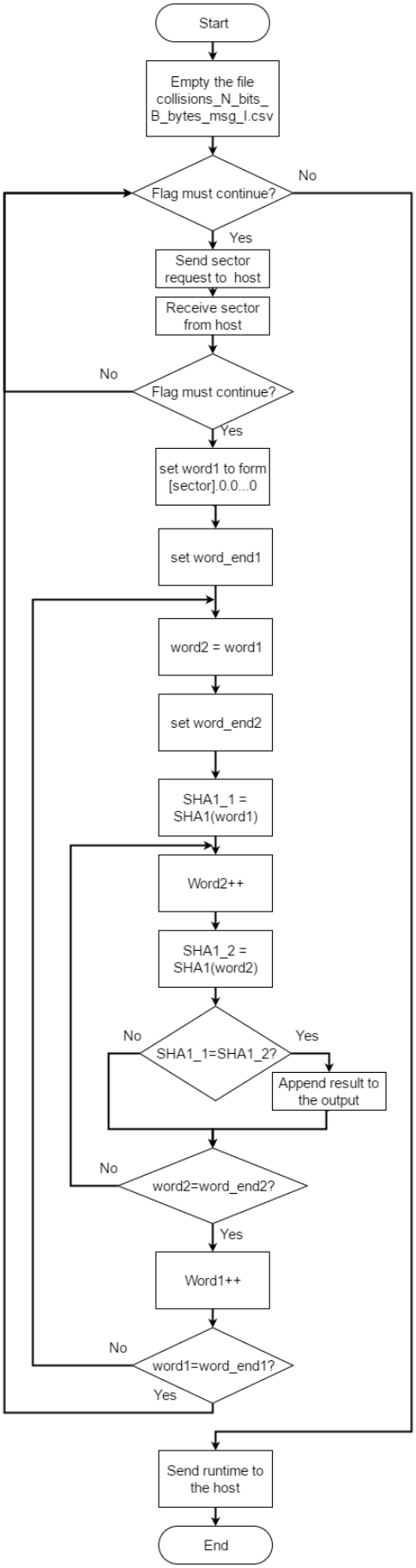

Figure 3. Worker-process flowchart.

Tables

Step (in the algorithm)	Type	Number of elements	Description
1-a	unsigned char	0	No data passed
1-b	unsigned char	2	First element – flag that worker should continue (to distinct 1-b and 2-b). Second element – is sector value (first byte of the word)
2-a	unsigned char	0	No data passed
2-b	unsigned char	2	First element – flag worker should stop (to distinct 1-b and 2-b). Second element – any value.
3-a	int	1	One int element representing number of seconds of the runtime

Table 1. Types of messages represented in the software.