Implementation of stream processing using the actor formalism for simulation of distributed insertion sort

S V Vostokin¹ and I V Kazakova¹

¹Samara National Research University, Moskovskoe Shosse 34, Samara, Russia, 443086

e-mail: easts@mail.ru

Abstract. In this paper we study the applicability of actor formalism in the field of distributed stream processing. The simulation model of stream processing is considered using the example of a distributed insertion sort algorithm. We built a simulation model calibrated by the results of full-scale measurements of the sorting time on the computing cluster. We considered possible applications of the proposed approach to the organization of calculations.

1. Introduction

Nowadays, parallel programming is increasingly relevant to develop of large system designed both for scientific research and for commercial use. Actor model is a universal model of computation that allows to implement the mechanism for parallel computing. Its main advantages include asynchronous messaging, opportunity for dynamic growth of the system within distributed environment [1].

The purpose of the article is to investigate the applicability of the Actor model in order to describe calculations as a stream of independent operations. Such organization of computing is used in batch system TORQUE, Slurm, compositional applications in service-oriented architectures, for example, Everest [2].

As an example, software implementation of the simulation model of stream processing for distributed insertion sort algorithm is considered. The time of calculations in a distributed algorithm with an arbitrary number of cluster nodes is studied using a simulation model, that has been calibrated according the results of full-scale measurements in one cluster node.

2. Method

The following algorithm is taken as a basis for parallel computing:

for (int i = 0; i<NUM_BLOCKS; i++) block_sort(i);
for (int i = 1; i<NUM_BLOCKS; i++) for (int j = 0; j<i; j++) block_merge(j, i),

from which it appeared, that all block operations of block_sort (i) are executed independently of each other, some merge operations - block_merge (j, i) - are also independent. Our goal is to describe a parallel algorithm so that independent operations are performed in parallel.

To implement the goal, the source code was developed in the C ++ programming language, the basis of which is the Actor model. Actors have not shared variables and interact with each other through message passing. In response to the received message, they are capable of:
• send messages to other actors;
• create new actors;
• set its own behavior in response to the next message received.

All the calculations performed in the execution of the code are organized according to a certain algorithm, performing by the control subsystem and the task execution subsystem (figure 1).

![Figure 1. Scheme of the calculations.](image)

The control subsystem determines the calculation strategy and simulates the control node, the mechanism of which is similar to the control node of the system given in [3]. Its main task is to generate a sequence of instructions block_sort() and block_merge() for the task execution subsystem based on the received messages. The control subsystem contains four main types of actors: sorter, merger, producer and stopper.

The producer and the stopper types of actors are auxiliary. The producer performs two functions: it waits for the completion of all sorts of blocks in the sorter, and then forms a sequence of block numbers for the merger. The stopper performs the only one function - it stops the calculation.

The main function of the sorter is to sort (by the Hoare method) the received data blocks. The Hoare method can be called one of the fastest. It chooses the pivot element, and places all other elements relative to the pivot element according to their values either to the right side of the array (if they are less than the selected one) or to the left side of the array (if they are larger than the selected one). After that, the both parts of the array are processed recursively, and so on [4].

The merger implements insertion sorting for the array. Scalar insertion sorting is an algorithm in which the movement of the elements of the block occurs one by one in the created, previously sorted sequence. This method is effective when merging previously sorted blocks. The merger is implemented using the following code:

```cpp
if (!access(_in) && access(out))return;
if (is_first){
    is_first=false; j = _in->i;
    _in->send();
} else{
    #if defined(SIMULATED_EXECUTION)
    double time;
    time = omp_get_wtime();
    #endif
    block_merge(j, _in->i);
    #if defined(SIMULATED_EXECUTION)
    time = omp_get_wtime() - time;
    delay(time*(1 + TRADEOFF));
    #endif
    out.i = _in->i;
    _in->send(); out.send();
} .
```
Let us consider the process of actor interaction during the block-sorting (figure 2).

The original array consists of 128000000 integers, filled with random values, divided into a specified number of equal blocks. The resulting blocks are addressed to perform a quick sorting by the actors of the sorter type. A unique a_sorter[] actor is created for each block. The number of actors of the sorter type varies depending on the number of blocks, resulting in the sequence a_sorter[0], a_sorter[1], ..., a_sorter[num_blocks-1]. At the end of sorting, a message about the end of the sorting stage is sent to a_producer. As soon as a_producer receives the proper number of confirmations from the a_sorter actors necessary for the start of the merging stage operations, it generates and passes the instructions to the next type of actors - the actors of the merger type. The actor a_merger[] receives the message and performs the actions in accordance with the following algorithm. If the message was received for the first time, then it is assumed that the iteration number in j is transmitted in it; the number j is remembered and the response is sent to the actor on the left in the chain (figure 2). If the message is received the second and the next time, the a_merger[] actor understands that the message has an iteration number in i, calls the block_merge(j, i) function, passes the iteration number i next to the chain to the right a_merger[] actor, and sends the message to the actor on the left in the chain (figure 2). Once the message is received by the a_stop actor, this actor stops the calculations.

Concurrency in this algorithm is accomplished through the asynchronous processing of messages by actors. After performing some operation, the next operations, dependent on it, are executed immediately. Several actors of sorter type or merger type can be executed at the same time.

3. Experimental results
To investigate the possibility of acceleration of the computations, a simulation model was created for the actor system (figure 2). The model was calibrated based on the results of field measurements on the cluster. During the experiment, calculations were made for sequential and parallel algorithms. For the initial data, an array of 128000000 integers is taken and divided according to table 1.

| Number blocks | Block size   |
|---------------|-------------|
| 2             | 64000000    |
| 4             | 32000000    |
| 8             | 16000000    |
| 16            | 8000000     |
| 32            | 4000000     |
| 64            | 2000000     |
| 128           | 1000000     |

Figure 2. Actor system. Block-sorting algorithm.
All phases of the study were carried out using the Templet Web service [5], used to manage and monitor the tasks launched on the cluster Samara university cluster system. The cluster consists of one manager and 165 computer servers. Each cluster node has a hard disk for storing temporary data. At the n1-n34 nodes is available 56 GB of storage capacity, on n35-n42, n57-n70, n113-n168 - 120 GB, n239-n242, n246-n250 - 244 GB, n281-n308 - 95 GB [6].

In the first experiment, a study was made of the possibility of accelerating calculations in the operating mode. To accomplish the task, 8 processors were involved, the total execution time of each algorithm was limited to 5 minutes. The nodes listed in table 2 were used.

| Number blocks | Block size  | Cluster node |
|---------------|-------------|--------------|
| 2             | 64000000    | n42          |
| 4             | 32000000    | n128         |
| 8             | 16000000    | n228         |
| 16            | 8000000     | n118         |
| 32            | 4000000     | n126         |
| 64            | 2000000     | n118         |
| 128           | 1000000     | n126         |

In the experiment, the computation time was measured for sequential sorting, sequential sorting by blocks, and parallel sorting by blocks using the model of actors. The results are shown in figure 3.

![Figure 3. Dependence of the execution time of sorts on the number of blocks in the array partition (operating mode).](image)

The second experiment was conducted in the simulation mode. Calculations have been made for sequential sorting in operating mode, sequential sorting and parallel sorting in simulation mode. To execute the sequential sort algorithm in the operating mode, 8 processors were used. To measure the calculation time in the simulation mode, additional processors were simulated, the number of which is equal to the number of blocks. The involved nodes are shown in table 3:
Table 3. Involved cluster nodes for the second experiment.

| Number blocks | Block size   | Cluster node |
|----------------|--------------|--------------|
| 2              | 64000000    | n36          |
| 4              | 32000000    | n6           |
| 8              | 16000000    | n35          |
| 16             | 8000000     | n7           |
| 32             | 4000000     | n136         |
| 64             | 2000000     | n149         |
| 128            | 1000000     | n145         |

The results of the second experiment are shown in figure 4.

![Figure 4](image-url)

Figure 4. Dependence of the execution time of sorts on the number of blocks in the array partition (simulation mode).

When sorting, data is exchanged between processors, which leads to an increase in the execution time of the algorithm. The percentage of overhead costs may vary depending on the site remoteness, for example, when transmitted over a network. In the third experiment, the time spent on interaction between processors in the simulation mode was measured, the influence of overhead on the execution time of block sorting was revealed. To accomplish the task, 32 blocks of 4,000,000 integers are taken, 8 processors are involved. To measure the calculation time in the simulation mode, 32 processors are used. The involved nodes are shown in table 4.

Table 4. Involved cluster nodes for the second experiment.

| Percentage of trade-off | Cluster node |
|-------------------------|--------------|
| 0                       | n136         |
| 10                      | n134         |
| 20                      | n147         |
| 30                      | n152         |
| 40                      | n130         |
| 50                      | n151         |
| 60                      | n145         |
| 70                      | n12          |
| 80                      | n117         |
| 90                      | n118         |
| 100                     | n228         |
Figure 5 shows the results of the third experiment, which demonstrate the effect of overhead on the execution time of block sorting in simulation mode.

![Simulated trade-off](image)

**Figure 5.** The effect of overhead on the execution time of block sorting in the simulation mode.

The fourth experiment demonstrates the overhead costs for sorting out two blocks of different sizes with data transmission over the network in the operating mode. The transmission was carried out between two nodes: n36 and n136. Blocks were used, including 1,000,000, 2,000,000, 4,000,000, 8,000,000, 16,000,000, 32,000,000, 64,000,000 integers. The results of the fourth experiment are shown in figure 6.

![Real trade-off](image)

**Figure 6.** Overhead costs for performing sorting from two blocks of different sizes with data transmission over the network in the operating mode.

The first experiment showed that the calculation time for parallel sorting of blocks is significantly reduced compared to sequential sorting of blocks. The most effective algorithm was for sorting 16 blocks, consisting of 8,000,000 integers, the execution time of which is 4,848.45 s. For sequential sorting of blocks with the same initial conditions, it took 19.245 seconds.

A feature of the second experiment was the simulation implementation of calculations. Templet Web [5], used in the experiment, makes it possible to calculate the algorithm running time on the assumption that each actor is executed on a separate processor. In this case, the actual operation of the algorithm is performed on one core of the real processor, and the time delays of individual operations
are transferred to the simulation system to calculate the model time. In the Merger actor code (see Sec. 2), you can see the call to the delay() method, which passes the measured merge time of the two blocks, taking into account the overhead in the simulation system. Thus, in the second experiment, the number of model processors is equal to the number of blocks. The results of the second experiment also demonstrate the effectiveness of the applied parallel block sorting method. For 2, 4, 8 and 16 blocks, including 64,000,000, 32,000,000, 16,000,000 and 8,000,000 integers, the result in the simulation mode is slightly different from the result in the operating mode. But for 32, 64 and 128 blocks, including 4,000,000, 2,000,000 and 1,000,000 integers, there is an intensive growth in time for the algorithm for sequential sorting of blocks and reducing the time required for parallel sorting of blocks. For 128 blocks of 1,000,000 integers, the result of sequential sorting of blocks is 50,656 s, and the result of parallel sorting of blocks is 2,066 s.

In the third experiment, the influence of overhead on the execution time of block sorting in the simulation mode was investigated. Figure 5 shows that an increase in the percentage of overhead costs leads to an increase in the execution time of calculations. If the time for parallel sorting of blocks with zero percent overhead is 2.542 s, then with an increase in the percentage of overheads to 100 percent, it will be 4.85475 s. Based on the indicators of the third experiment, it can be argued that an increase in overhead costs to 100 percent leads to an increase in the execution time of the algorithm by 190.97 percent. However, the results of the fourth experiment show that in the operating mode, the overhead is no more than 10 percent for the cluster system, which has little effect on the time of parallel block sorting.

4. Conclusion
The purpose of this paper was to investigate the possibility of using the model of actors to describe computations in the form of a stream of independent operations. As an example, the software implementation of the simulation model of stream computations of the distributed insertion sort algorithm is considered.

The conducted experiments demonstrated that the distributed sorting by the proposed method allows to reduce the time costs with respect to the methods of sequential sorting, even taking into account overhead costs. The obtained results are a proof of the efficiency of the organization of calculations considered in the article.

The results of the work done and the proposed approach to the organization of calculations can be applied to various algorithms that are based on the stream processing of large data blocks, for example in the area of computer optics [7,8]. The source code of the sorting method discussed can be found on the GitHub service [9].

5. References
[1] Fedotov I E 2012 Parallel programming models (Moscow: Solon-Press) p. 384
[2] Sukhoroslov O, Volkov S and Afanasiev A 2015 Web-Based Platform for Publication and Distributed Execution of Computing Applications IEEE Xplore 14 175-184
[3] Knyazkov K V, Kovalchuk S V, Tchurov T N, Maryin S V and Boukhanovsky AV 2012 Clavire: e-science infrastructure for data-driven computing Journal of Computational Science 3 504-510
[4] Knuth D E 1998 The Art of Computer Programming: Volume 3: Sorting and Searching (Boston: Addison-Wesley) p 780
[5] Vostokin S, Artamonov Y and Tsarev D 2018 Templet Web: the use of volunteer computing approach in PaaS-style cloud Open Engineering 8 50-56
[6] Samara University Supercomputer Center (Access mode: http://hpc.ssau.ru/node/22)
[7] Vorotnikova D G, Golovashkin D L 2017 Difference solutions of the wave equation on GPU with reuse of pairwise sums of the differential template Computer Optics 41(1) 134-138 DOI: 10.18287/2412-6179-2017-41-1-134-138
[8] Malysheva S A, Golovashkin D L 2016 Implementation of the FDTD algorithm on GPU using a pyramid method Computer Optics 40(2) 179-187 DOI: 10.18287/2412-6179-2016-40-2-179-187

[9] The block sorting algorithm with actors (Access mode: https://github.com/Templet-language/newtemplet/tree/master/samples/blocksort)

Acknowledgments
This work is partially supported by the Ministry of Education and Science of the Russian Federation in the framework of the State Assignments Program (#9.1616.2017/4.6).