Improving the performance of processing special computing tasks using an asynchronous actor model

E G Andrianova¹, V E Sachkov¹ and D O Zhukov¹

¹MIREA - Russian Technological University, 78 Vernadsky Avenue, Moscow, 119454, Russia

E-mail: dtghmflysq@gmail.com

Abstract. The use of third-party ready-made solutions to perform special computing tasks (SCT): processing text arrays in a natural language, the use of cryptography methods in data transmission networks, machine learning in solving applied problems, scientific computations and many other ones is a common practice in modern development of software systems. However, the use of third-party ready-made software solutions often leads to their poor optimization for parallel and distributed operation and to the difficulty in vertical and horizontal scaling, impossibility of modifying a solution, e.g., due to its legal protection, etc. In order to solve these problems, an asynchronous actor model can be used that facilitates developing software for processing special computing tasks. In addition, an asynchronous actor model allows parallel processing of input tasks, provides asynchronous operation, and extends its functionality to third-party ready-made solutions, that is impossible or forbidden to modify to meet user’s or optimization requirements. Methods, algorithms, and the entire logic of asynchronous actor operation involve transitions of actors from one state to another under the effect of input messages and returning the results of processed tasks to the frame system. This allows setting an automatic grammar and for using methods and approaches for developing software systems based on finite-state automata. The asynchronous actor operation scheme is represented by a directional state graph, which allows using the mathematical apparatus of the finite-state automaton theory (the Mealy model) to describe the same. The developed model is convenient for designing and organizing the SCT processing using computer systems. The use of the asynchronous actor model makes it possible, in special operation modes, to increase the performance of poorly optimized SCT without modifying the same, as compared to task processing without using the asynchronous actor model.

1. Introduction

Processing large arrays of data to extract valuable knowledge using powerful computing systems is an urgent problem [1, 2]. The active development of a declarative approach to software development, a functional programming paradigm, NoSQL, cluster solutions, (Hadoop, Spark), coprocessors and FPLD require finding new solutions for special data processing.

According to Amdahl’s Law and scalability properties [3, 4], to increase the performance when solving special computing tasks (SCT), all available resources have often to be used: CPUs, GPUs, and communication channels to maximize the performance while minimizing the related overhead costs. This is especially relevant when building large, geographically distributed automated systems [5] or in cloud computing [6].

There is no single SCT standard describing their solution. Each type of SCT is unique and requires developing its own solution method, optimizing calculations and algorithms. The complexity of
methods and algorithms for solving SCT requires using special ready-made third-party solutions, in addition to in house developments. This is a common practice in the development of special modern software, but it is associated with overcoming some new problems: the impossibility of modifying a proprietary solution; a poor optimization of parallel SCT processing (no distribution of tasks by computing clusters), and some others. Using the existing tools to solve SCTs, such as OpenMP / MPI, gives rise to other problems such as platform incompatibility, high development complexity and deployment and configuration complexity. Therefore, new methods for solving SCT shall be found, such as asynchronous actor model that facilitates developing solutions and allows increasing the SCT processing performance using third-party ready-made solutions without modifying the same on a computer system or cluster.

2. Asynchronous actor model
Actor is a high-level primitive parallelism which allows simulating parallel computing using entities that interact with each other by sending messages [7, 8]. In fact, an actor itself is a computing entity that, in response to a received message, can simultaneously:
- send a finite number of messages to other actors;
- create a finite number of new actors;
- choose a behavior scenario for processing the next received message.

The suggested model preserves these properties when supplementing them with an event-oriented architecture. The operation basis for the asynchronous actor model is the actor transition from one state to another under the effect of input messages (containing a computational task for processing) and returning the processed task result to another actor or the frame system (the actor can be integrated in a program; it can be a service in the corporate bus; it can be controlled by a cluster job manager).

The directional state graph of the developed model is presented in figure 1.

![State graph of the developed model of asynchronous actors.](image-url)

**Figure 1.** State graph of the developed model of asynchronous actors.

The model describes the following states of the actor:
- message waiting: the actor receives input messages; the number of messages is limited by the size of its mailbox;
- message checking: the actor identifies the message recipient, integrity, and correctness;
operating mode setting: selecting the actor operating mode when processing input messages that contain a computational task;
processing: the actor asynchronously processes the input message;
sending: sending the results after processing.

To formalize the mathematical representation of the model, we used the finite-state automaton theory. Our choice is due to the logic of asynchronous actor operation. This allowed presenting input/output messages and the operation algorithm as a state graph, transition/exit tables, and a set of transition rules.

Define many model states
\[ Q = \{ \text{message waiting, message checking, setting operating mode, processing, sending} \} = \{ q_0, q_1, ..., q_n \}. \]

Define the input alphabet \( x \), for convenience, we divide the alphabet into input symbols from each state of the model,
so that \( x = \{ x'_0 \cup x'_1 \cup ... x'_n \} \sim \{ x_0, x_1, ..., x_n \} \), where \( x'_i \) are input characters for \( q_i \).

Input Alphabet:
\[ x'_0 = \{ \text{message, waiting} \}; \]
\[ x'_1 = \{ \text{for the current actor, not for the current actor; the total} \}; \]
\[ x'_2 = \{ \text{user defined mode, undefined mode} \}; \]
\[ x'_3 = \{ \text{there is an adapter, no adapter} \}; \]
\[ x'_4 = \{ \text{recipient address is indicated, recipient address is missing} \}. \]

Define the output alphabet, for convenience, we divide the alphabet into output symbols from each state of the model, so that \( y = \{ y'_0 \cup y'_1 \cup ... y'_n \} \sim \{ y_0, y_1, ..., y_n \} \), where \( y'_i \) are output characters for \( q_i \).

Output Alphabet:
\[ y'_0 = \{ \text{received a message in the mailbox, waiting for a message} \}; \]
\[ y'_1 = \{ \text{taken into processing, rejected} \}; \]
\[ y'_2 = \{ \text{user mode, balancer operating mode} \}; \]
\[ y'_3 = \{ \text{sending result, error} \}; \]
\[ y'_4 = \{ \text{sending to recipient, sending to decision queue} \}. \]

For a more accurate description of the logic of the model of a finite state machine of an asynchronous actor, we define table 1 transitions / outputs, where \( x_i \in X, y_i \in Y, q_i \in Q \):

**Table 1.** Table of transitions / outputs of the asynchronous actor model.

| \( Q \times X \) | \( x_0 \) | \( x_1 \) | \( x_2 \) | \( x_3 \) | \( x_4 \) | \( x_5 \) | \( x_6 \) | \( x_7 \) | \( x_8 \) | \( x_9 \) | \( x_{10} \) |
|----------------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| \( q_0 \)      | \( q_1 \) | \( q_2 \) | \( q_3 \) | \( q_4 \) | \( q_0/y_0 \) | \( q_0/y_1 \) | \( q_0/y_2 \) | \( q_0/y_3 \) | \( q_0/y_4 \) | \( q_0/y_7 \) | \( q_0/y_7 \) |
| \( q_1 \)      | \( q_2 \) | \( q_3 \) | \( q_4 \) | \( q_0 \)  | \( q_0/y_0 \) | \( q_0/y_1 \) | \( q_0/y_2 \) | \( q_0/y_3 \) | \( q_0/y_4 \) | \( q_0/y_7 \) | \( q_0/y_7 \) |
| \( q_2 \)      | \( q_3 \) | \( q_4 \) | \( q_1 \) | \( q_0 \)  | \( q_0/y_0 \) | \( q_0/y_1 \) | \( q_0/y_2 \) | \( q_0/y_3 \) | \( q_0/y_4 \) | \( q_0/y_7 \) | \( q_0/y_7 \) |
| \( q_3 \)      | \( q_4 \) | \( q_1 \) | \( q_0 \)  | \( q_0 \)  | \( q_0/y_0 \) | \( q_0/y_1 \) | \( q_0/y_2 \) | \( q_0/y_3 \) | \( q_0/y_4 \) | \( q_0/y_7 \) | \( q_0/y_7 \) |
| \( q_4 \)      | \( q_1 \) | \( q_2 \) | \( q_3 \) | \( q_4 \) | \( q_0/y_0 \) | \( q_0/y_1 \) | \( q_0/y_2 \) | \( q_0/y_3 \) | \( q_0/y_4 \) | \( q_0/y_7 \) | \( q_0/y_7 \) |

If errors occur in the processing of input signal due to incorrect data or during calculations, the model seeks to return to its final state: “message waiting” (state \( q_0 \)).

We will visually present the data in table 1 as a state graph. As the automaton model operation depends on its state and input messages (signals), we will use a Mealy automaton to draw a state graph (figure 2).

Let us determine the state transition rules on the graph arcs as per table 1. To draw arcs denoting transitions in case of errors, let us introduce transition rules for the following errors: \( E_i \rightarrow (x_{j+1} \lor x_{j+2} \lor ... x_{j+n}), x_j \notin x'_i, i \) is sequence number of the state where the input signals are defined, with which the automaton in the current state returns to the final state as per table 1.
Let us consider the main elements interacting in the asynchronous actor model:

1. Asynchronous actor is a model element that implements an asynchronous logic: receiving/sending messages between actors, launching SCT processing adapters, error processing, and recording errors to log files.

2. Message processing adapter is an element designed to use the functionality of an object the modification of which is prohibited or unavailable, through a specially designed interface. The implementation of the above object allows connecting any third-party solution that is not compatible with the actor platform. In the asynchronous actor mode, the following elements are connected as adapters: an in house development, a third-party development, and open programs, modules, and libraries responsible for processing some special computational task according to the black box model. An adapter can process only one type of task.

3. Load balancer is an element designed to optimize the resource use and to increase computing performance, by defining the asynchronous actor mode and parameters. The load balancer determines:
   - the asynchronous actor operating mode;
   - the number of the actor’s active copies;
   - the number of the actor’s workflows/processes;
   - the child copies operating mode. The optimal mode is determined by analyzing the SCT processing statistics and by selecting the best parameters from the same, that provide the highest processing speed.

   The performance increase in the model is due to launching adapters in special modes. Having analyzed different approaches to optimization and parallel development methods [9, 10], we identified the following adaptor operating modes for launching an asynchronous actor and for free switching between actors:
   - single-threaded mode: sequential message processing without creating parallel threads. This is a blocking operation mode; an actor can not perform more than one task at a time;
   - multi-threaded mode: the actor processes messages in parallel, using threads;
   - multi-process mode: for parallel message processing, the actor uses lightweight processes that do not require a large number.
   - copy mode: an additional mode of creating a copy by the actor and launching the same in one of the above modes.

   These modes provide the maximum processing flexibility without the need to modify asynchronous actors and message processing adapters.
Let us consider the algorithm of the developed asynchronous actor model, as represented by a UML sequence diagram.

The main steps of the algorithm operation are as follows:
- after launching, an asynchronous actor monitors the data bus via which it can communicate with other actors (task queue);
- having received a message, the actor processes it asynchronously;
- after initial processing (checking the headers, addresses, and data integrity), the actor asks for an operating mode to process SCT from the load balancer;
- after receiving and setting the operating mode, the actor calls the message handler containing loaded adapters. As each SCT is unique, a separate adapter is loaded for each SCT type. If the required adapter is not available, the message handler returns an error message;
- all these actions are performed asynchronously, and during the SCT processing, the actor waits for the next message.
- at the end of processing, the actor sends the result to the decision queue or to another actor and collects statistics (load, the use of random-access memory, the current SCT processing time, the average SCT processing speed, the number of processed messages since the launch, the current SCT metadata);
- the collected statistics are sent to the load balancer module and recorded to the actor statistics files.

Figure 3. UML asynchronous actor model sequence diagram.

3. Testing the Asynchronous Actor Model
The main purpose of developing the asynchronous actor model is creating software that increases the SCT processing performance without modifying third-party solutions used to solve SCT.

When testing the developed model, we checked the increase in the SCT solving performance. During the tests, we used a third-party ready-made solution that was not adapted for operating in a parallel mode with a computer system. We performed task parallelization in multi-threaded/multi-process modes using the asynchronous actor tools without modifying the ready-made solution.

To demonstrate the advantages of using actors, we selected a relevant and demanded SCT: processing a text in a natural language [11, 12]. The above SCT is used for text processing, extracting
valuable knowledge, training models, creating document bodies, semantic ontologies, requiring a huge processing power for processing hundreds of millions of documents.

Let us present the results of processing an unoptimized SCT: natural language text preprocessing, including tokenization, stemming, and lemmatization. During the tests, we used a computer system with the following hardware parameters:
- Intel Xeon E5-2660V2 x processor – 2 pcs;
- 128 Gb RAM;
- CentOS 7 operating system.

The results of measuring the asynchronous actor operation are presented as a graph of CPU load via a Zabbix monitoring system (figure 4), where the ordinate shows the CPU load in percent, and the abscissa shows the operating time with an interval of one minute. Figure 5 shows the graph of message processing speed per second by an asynchronous actor in various modes, where the ordinate shows the number of messages processed per second (msg/sec), and the abscissa shows the operating time with an interval of one second.

![Figure 4. Schedule CPU load through the monitoring system Zabbix.](image)

![Figure 5. Graph of message processing speed per second by an asynchronous actor in different modes.](image)

All graphs clearly show the mode switching steps, whereas the graph nodes indicate the increase/decrease in the processing performance. We should also pay attention to the CPU operation graph, where, with a slight difference in speed, we can see a difference in the system resource consumption. The following is a breakdown of an asynchronous actor statistics when processing an unoptimized SCT in table 2.
Table 2. Asynchronous actor statistics when processing an unoptimized SCT.

| Mode average speed | Average speed msg/sec | Min msg/sec | Max msg/sec | Number of copies | Number of threads | Average processing time in the adapter sec |
|-------------------|-----------------------|-------------|------------|------------------|------------------|-------------------------------------------|
| SINGLE            | 23.36                 | 0.40        | 37.08      | 1                | 1                | 0.0338                                    |
| SINGLE            | 25.44                 | 0.40        | 67.12      | 2                | 1                | 0.0323                                    |
| SINGLE            | 31.92                 | 0.40        | 156.40     | 5                | 1                | 0.0341                                    |
| SINGLE            | 53.65                 | 0.40        | 396.93     | 10               | 1                | 0.0364                                    |
| SINGLE            | 66.67                 | 0.40        | 490.93     | 20               | 1                | 0.0448                                    |
| SINGLE            | 105.87                | 0.40        | 520.11     | 40               | 1                | 0.0696                                    |
| THREAD            | 23.36                 | 0.40        | 37.08      | 1                | 40               | 0.0356                                    |
| THREAD            | 25.12                 | 0.40        | 65.62      | 2                | 20               | 0.0364                                    |
| THREAD            | 30.83                 | 0.40        | 194.21     | 5                | 8                | 0.0379                                    |
| THREAD            | 44.46                 | 0.40        | 380.52     | 10               | 4                | 0.0381                                    |
| THREAD            | 69.87                 | 0.40        | 716.01     | 20               | 2                | 0.0481                                    |
| THREAD            | 111.39                | 0.40        | 732.16     | 40               | 1                | 0.0650                                    |
| PROCESS           | 142.86                | 7.39        | 492.03     | 1                | 40               | 0.0402                                    |
| PROCESS           | 158.99                | 3.78        | 439.37     | 2                | 20               | 0.0678                                    |
| PROCESS           | 244.78                | 1.68        | 798.99     | 5                | 8                | 0.0662                                    |
| PROCESS           | 241.81                | 57.79       | 1126.64    | 10               | 4                | 0.0716                                    |
| PROCESS           | 217.05                | 23.82       | 934.75     | 20               | 2                | 0.0829                                    |
| PROCESS           | 193.74                | 15.81       | 263.60     | 40               | 1                | 0.0868                                    |

Having analyzing the results, we can draw the following conclusion: the SCT is poorly optimized for parallel operation in a computer system when running in single-threaded mode with one copy of the actor, which corresponds to a normal operation of the solution. The SCT processing speed averages 23.36 messages per second; the CPU load is ~ 10%.

During the asynchronous actor operation, the graphs clearly show the mode switching steps. There is an increase in the CPU load, the network traffic, the number of processed messages. Table 2 suggests that changing the modes, the number of running copies, and the number of workflows affects the average message processing speed, the maximum and minimum message processing speed, the average message processing time in the adapter (overheads for receiving, sending, checking, etc. are not taken into account). On the example of single-threaded and multi-threaded modes, we can monitor an increase in performance in terms of the average processing speed. In the multi-process mode, we can monitor the implementation of the Amdahl law. Although the multi-process mode is the best in terms of processing speed, increasing the number of copies/threads decreases its performance. Figure 4 proves that this mode is the most resource intensive.

Processing SCT using the asynchronous actor model allows the user to modify (manually or automatically) the operating modes, the number of copies/threads. This allows finding a compromise between performances and computing resources consumption. Operating in various asynchronous actor modes increases the performance of poorly optimized third-party solutions connected through an adapter, without modifying the solutions themselves. Based on the monitoring data of asynchronous actor operation, we can determine the optimal operating modes and increase the performance 1.3 to 10.4 times, as compared to the regular solution operating mode, without using actors for the current SCT.

4. Conclusion
Due to running in special operating modes, the asynchronous actor model allows increasing the performance of poorly optimized SCTs without modifying the same, as compared to task processing without using actors. Methods, algorithms, and the entire logic of asynchronous actor operation involve actor transitions from one state to another under the effect of input messages and returning the results of processed tasks to the frame system. This allows setting an automatic grammar and for using the methods and approaches for developing software systems based on finite-state automata.
developed model is convenient for designing and organizing the SCT processing using computer systems.

References

[1] Lynch C 2008 Nature 455(7209) 28–29. DOI:10.1038/455028a
[2] Laney D 2017 International Transaction of Electrical and Computer Engineers System 4(1) 14-25. DOI:10.12691/iteces-4-1-3
[3] Amdahl G M 1967 Proc. of the April 18-20, 1967, spring joint computer conference (AFIPS '67 (Springer), NY.: Association for Computing Machinery) pp 483–485. DOI:10.1145/1465482.1465560
[4] Popov G, Mastorakis N and Mladenov V 2010 ICCOMP’10: Proceedings of the 14th WSEAS international conference on Computers: part of the 14th WSEAS CSCC multiconference vol. 2 pp 411-414
[5] Tretyakov V A, Kulikov G V and Lukyanets Y F 2020 Russian Technological Journal 8(1) 34-42. DOI:10.32362/2500-316X-2020-8-1-34-42
[6] Nazarov A N 2019 Russian Technological Journal 7(6) 56-67. DOI:10.32362/2500-316X-2019-7-6-56-67
[7] Hewitt C, Bishop P and Steiger R 1973 Proc. International Joint Conference on Artificial Intelligence pp 235–245
[8] Agha G 1986 ACTORS: A Model of Concurrent Computation in Distributed Systems (Cambridge: MIT Press) p 190
[9] Ungerer T, Robie B and Silc J 2003 ACM Comput. Surv. 35(1) 29-63. DOI:10.1145/641865.641867
[10] Liu D and Deters R 2008 Service-Oriented Computing (ICSOC 2008 Workshops. Berlin, Heidelberg) vol. 5472 pp 166-177. DOI:10.1007/978-3-642-01247-1_16
[11] Sachkov V E, Zhukov D O, Korablin Y P, Raev V K and Akimov D A 2018 Convergent Cognitive Information Technologies. (Springer, Cham: Communications in Computer and Information Science) vol 1140 pp 78-70. DOI:10.1007/978-3-030-37436-5_6
[12] Sachkov V E, Akimov D A and Pavelyev S A 2018 International Journal of Engineering & Technology 7(4) 422-425. DOI:10.14419/ijet.v7i4.38.24595