ISSN: 2278-4632
Vol-08 Issue-01 : 2018

DESIGN OF A DISTRIBUTED SYSTEM FOR PARALLEL EXECUTION OF NUMERICAL EXPERIMENTS ON HUGE DATA SETS

Edi Laxmi[1], Manaswini Pattanayak[2], Biswaranjan Barik[3]

[1]Prof. Einstein Academy of Technology and Management, Bhubaneswar, India
[2]Asst. Prof. Einstein Academy of Technology and Management, Bhubaneswar, India
[3]Student, Einstein Academy of Technology and Management, Bhubaneswar, India

The main difficult in computer modeling is getting numerical results from computational heavyweight models for large input parameters sets. This problem arises when researchers need to get exact outcomes for data conception. For instance, execution of a simulation model on thousands of input parameters combinations can take days or weeks. In this article we describe an architecture of a distributed system *which* allows to run computations in parallel on large input Basing on performance we may consider two models simulation model and of a wireless network

To show the performance gain, we consider two models: simulation model of a wireless network implemented in NS-3 and analytical model of a complex queuing system written in Python, and demonstrate the increase of computation speed.

Key words and phrases: architecture of a distributed system ,computer science, parallel tasks execution, simulation model.

1. Introduction

In research work we often find ourselves in a situation when we need to get numerical results from a model or algorithm on a large set of input parameters. These models and algorithms can be rather complex and require lots of time and computational resources. For instance, simulation model of computer network may require several minutes to properly estimate average network throughput or packet loss probability. If we need to study dependencies between these characteristics and input parameters, we may needto run the simulation on a set of several hundreds or even thousands of parametersvalues, and this will take many days or weeks.

One of the most effective approaches to improving performance is the use of parallel computing. Basically, there are two ways to make use of multiple CPUs in computations. The first one is called program (or function) parallelism. According to this approach,the algorithm should be developed in such a way that different parts can be executedin parallel by different threads. If several threads need to be executed in the same time, each of them can use separate CPU (of course, if there are enough CPUs). However,not all algorithms may utilize parallelism effectively, and also this approach require algorithm rethinking and lots of manual work. The second way to use multiple CPUsis to simultaneously execute several instances of the same program with different input data on separate CPUs. This approach is more simple and universal, it doesn't require parallelism inside each program. To implement this approach, we developed a systemfor distributed computations, which we describe in this paper. The system provides an intuitive Web interface for users, as well as REST API, and can be used for a very wide range of computational tasks.

As an example, we consider two computational problems: performance evaluation ofa wireless network using NS-3 simulation model, and computation of numerical solution of a complex queuing system analytical model (implemented in Python language). We studied time required for getting numerical solutions of each problem on one hundredof different input vectors, depending on the number of worker threads used by our distributed system, and workers properties. We found that under certain conditions the system allows to effectively use all CPUs and decrease the time required for the problem solution proportionally to the number of threads.

The paper is organized as follows. In the second part we provide a brief overviewof related works and outline the key differences of our approach. In the third part we describe the system architecture. Part 4 is devoted to details on tasks execution, and part 5 presents numerical results obtained on the sample problems. Part 6 concludes the paper.

2. Related Works

A similar problem of computing tasks on a large number of inputs was consideredin articles [1–4]. In

ISSN: 2278-4632
Vol-08 Issue-01 : 2018

these articles web application for running calculations jobs called Everest [5] was described. The application has user-interface, where user can input parameters of computed job. After that, jobs are deployed and computed on services connected to the system.

The system Nimrod [6, 7] is the tool for planed computing. User can create a plan for an experiment and use the system to dispatch the tasks over multiple computing resources and collect the results. To configure tasks scheduling user creates declarative "plan" file which describes the parameters, then system schedules this job on the first available service connected to the system.

There are a lot of differences in implementation between our system and Everest or Nimrod. The key different is the process of tasks and environment preparation. Our system uses Docker to support all kind of tasks independently of infrastructure. All executable tasks are wrapped into Docker containers. To start running, a server with workers need only Docker installed and the Supervisor which is a part of our platform.

3. System Architecture

The main goal of the distributed system is to simplify programs executions on large input data sets. It should be a user-friendly application for running and managing tasks, provide means for storing input parameters, results and tasks execution statistics. The application should satisfy the following requirements:

- support various kinds of tasks, e.g. running computer networks simulations, perfor- mance evaluation of stochastic models, optimization problems solving, evaluation of physical and economical numerical models, running machine learning algorithms and so on;
- provide user-friendly interface, where user can control tasks and track execution status;
- provide means for system configuration, including the number of workers (pro- cesses for tasks execution);
- store results;
- the system deployment should be as easy as possible.

The distributed system includes a web application and computational core. The web application contains a frontend and backend. It also includes SQL database (Post-greSQL) for parameters and results storage, and Redis queue to move tasks between services. The client part of application (frontend) is written in JavaScript language using React framework. In the server side of application we used Python language and Fast API framework. The client side interacts with the server via HTTP requests and websockets.

Computational core includes a separate service called Supervisor, which manages a pool of Workers. The count of Workers depends on the number of CPUs. As a rule of thumb, a CPU should be allocated for each worker. The task processing workflow is illustrated in Fig. 1.

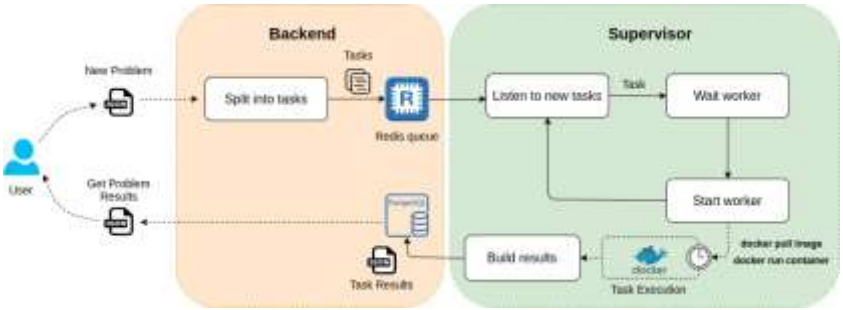

Figure 1. System workflow for handling and execution of tasks

4. Tasks management and execution

As previously mentioned, the main goal of the system is to solve computational problems on a large number of input values sets. To describe the problem, user should prepare a JSON array, where each item is a set of input values that correspond to a separate computational task. After JSON file is uploaded via UI into system or REST API, the problem is decomposed by the backend side into tasks. These tasks are inserted into Redis queue, from which they can be received by the computational core. Supervisor pulls the task from the queue, creates a worker and begins task execution.

The task life cycle is as follows. After creation the task is inserted into database with status

ISSN: 2278-4632
Vol-08 Issue-01 : 2018

INITIALIZED. In the first stage, the task is appended to the queue and receives status IN_QUEUE. When the Supervisor finds that task queue isn't empty, it checks whether any worker is ready for a new task. Each worker can be either free, or busy. In free state the worker is waiting for a task and doesn't consume CPU resources. In busy state the worker executes a task. After one of the worker becomes free, Supervisor pulls a task from the queue and assigns this task to the free worker. Then task status becomes RUNNING. After execution was completed and result is saved into the database, task status is changed to COMPLETED. In case when execution completed with error, diagnostic information is saved into database and task status is set to ERROR.

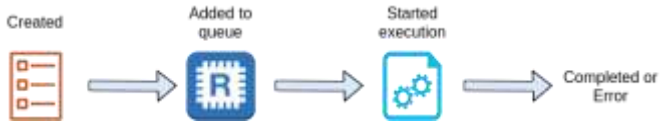

Figure 2. Task status change during execution

The system should support any kind of computational tasks. However, different tasks require unique environments for execution. For example, network simulation model may require a C++ compiler and NS-3 framework, Machine Learning tasks require Python interpreter and statistical libraries. The obvious solution is to customize environment for each individual problem, but it is not the best option since it comes with lots of overhead work. To solve this problem in a more universal and easy way, we decided to use Docker containers technology. User should define a Dockerfile, build docker image,

upload it to online container registry and pass the image URL when submitting a new problem. Before the worker starts task execution, the system pulls docker image. Worker creates a container from the image and executes it. Thus each task can be seen as a Docker container. After the execution is over, worker writes results into the database and removes the container.

Figure 3. Problem execution workflow

5. Results

To show the system performance we carried out experiments with simulation model of a wireless tandem network implemented in NS-3 and analytical model of a queuing system written in Python. All computations were performed on a virtual server at Yandex Cloud service with 16 vCPUs, 64 Gb RAM, running Ubuntu OS v20.04.

The input parameters of the simulation model consist of the IEEE 802.11 protocol settings, number of packets to generate, stations positions, antennas gain and transmit-ters power. The input parameters of analytical model include queue capacity, number of servers, parameters of inter-arrival and service times distributions. For both cases we created data sets consisting of 100 tasks. We packed tasks into problems (JSON input files) for each case and measured how long did it take to execute depending on the number of workers.

According to theoretical calculations, if problem execution on 1 worker takes T_1 time, then time of execution on n workers T_n equals:

$$T_n = \frac{T1}{n} + \delta_n(k), \quad (1)$$

ISSN: 2278-4632
Vol-08 Issue-01 : 2018

where δn is a total time of side effects: getting task from queue, start docker container, complete docker container and write results into database. Generally, $\delta n(k)$ depends on the number of tasks k in the queue. Equation (1) is precise only when each task is computed on one vCPU and the number of workers is less or equal to vCPUs count. Otherwise, if the number of workers is more than vCPUs count or workers use multiple threads, equation (1) doesn't provide a good measure, since workers interfere with each other for vCPU time and computational time is increased. To demonstrate this effect we used a multithreaded library for matrix operations in analitycal model, but set limits on the number of vCPUs each Docker container could use.

Fig. 4 shows the dependency between time required to solve the problem and the number of workers for three cases: analytical model without limits on the number of vCPUs for each worker, analytical model with 4 vCPUs per worker and NS-3 simulation model, which uses one vCPU per worker. For simulation model the global minimum is achieved when the number of workers is equal to sixteen – the number of vCPUs in the system. In case of analytic model with four threads per worker, time is almost the same starting from four workers (when all 16 vCPUs are busy), but slightly decreases still the number of workers grows up to 16. As for analytic model without limits on vCPUs usage by workers, increase in the number of workers doesn't accelerate the overall execution. Moreover, problem solution time is even increased due to overhead caused by the concurrent access to vCPUs by multiple threads.

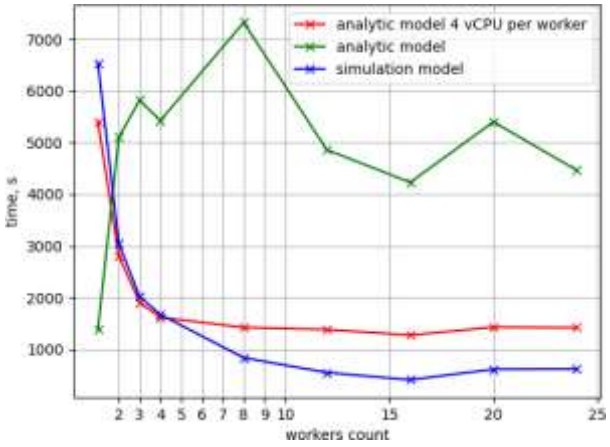

Figure 4. Time execution dependently on workers count

6. Conclusion

In this paper we described the problem of getting numerical results for problems with large number of input parameters sets. Usually, when one needs to get numerical results of computational heavy problems on thousands of different input sets, it requires lots of time and computational resources, up to days or weeks. As a solution of this problem, we proposed a system for running several instances of program in parallel. The system includes a web application with user interface, backend, SQL and Redis databases, and computational core comprised of Supervisor and worker processes. To unify the problem specification environment, the system expects computational problems in the form of Docker images accepting JSON inputs and producing JSON outputs. We studied the decrease of time needed to solve the problem on two kinds of problems: analytical solution of a multi-server queueing model with priorities and wireless network simulation in NS-3. Our experiments show, that while generally our systems allows to decrease time approximately like $1/n$, where n is the number of workers, in some cases when executables in Docker containers use multithreaded libraries, there is no performance gain, so limiting the number of CPUs per worker is required.

The system presented in this paper is a work in progress. We plan to add many fea- tures: integrations with popular cloud platforms (Yandex Cloud, Vscale, Digital Ocean, Amazon AWS, etc.) to automate computational core deployment, dynamic resource al- location and scheduling, user and API authentication. We also plan to improve UI/UX, add statistic and monitoring dashboards. Finally, we plan to build accurate mathemati- cal models for expected execution time estimation and dynamic computational resources allocations.

Source code of the system is available at https://gitlab.com/lab69/multi-runner.

ISSN: 2278-4632
Vol-08 Issue-01 : 2018

References

1. *Voloshinov V., Smirnov S., Sukhoroslov O.* Implementation and Use of Coarse- grained Parallel Branch-and-bound in Everest Distributed Environment // Procedia Computer Science. Volume 108, 2017, pp. 1532-1541.
2. *Sukhoroslov O., Putilina E.* Cloud Services for Automation of Scientific and Engineering Computations // Science. Business. Society. Issue 2, 2016, pp. 6-9.
3. *Sukhoroslov O., Volkov S., Afanasiev A.* A Web-Based Platform for Publication and Distributed Execution of Computing Applications // 14th International Symposium on Parallel and Distributed Computing (ISPDC). IEEE, 2015, pp. 175-184.
4. *Alexander Afanasiev, Oleg Sukhoroslov, and Vladimir Voloshinov* MathCloud: Pub- lication and reuse of scientific applications as restful web services. In Parallel Com- puting Technologies, pages 394–408. Springer, 2013.
5. Everest. [online]. http://everest.distcomp.org/.
6. *David Abramson, Jon Giddy, and Lew Kotler* High performance parametric model-ing with nimrod/g: Killer application for the global grid? In Parallel and DistributedProcessing Symposium, International, pages 520–520. IEEE Computer Society, 2000.
7. *David Abramson, Rok Sosic, Jonathan Giddy, and B Hall.* Nimrod: a tool for performing parametrised simulations using distributed workstations. In High Per- formance Distributed Com- puting, 1995., Proceedings of the Fourth IEEE Interna-tional Symposium on, pages 112–121. IEEE, 1995.