MODIFIED MASTER-SLAVE ALGORITHM FOR LOAD BALANCING IN PARALLEL APPLICATIONS

Luka Filipović[*]*, Božo Krstajić*[**]

Keywords: *Parallel computing, load balancing, task scheduling, MPI*

Abstract: This paper presents modified master-slave algorithm for load balancing improvements in parallel applications. The proposed algorithm combines static and dynamic algorithms and improves load balancing during critical parts of task execution. Use of the proposed algorithm causes a reduction in execution time of parallel application and increase of utilization of parallel computer resources. Simulation results using the proposed modified master-slave algorithm approved reduction in execution time and better occupancy of parallel infrastructure.

1. INTRODUCTION

In last decade, trends based on faster computer networks, powerful distributed computers and multicore computer architecture suggest that the parallelism is future of computing. Parallel programming has a reputation as the most modern sectors in the IT world, whose development is stimulated by simulations of complex systems [1]. Parallel programming is based on the concept of accelerating the execution of the program by dividing the program into multiple parts that can be executed simultaneously on separate servers. The main reasons for the use of parallel processing of are savings of time, solving major problems and ensuring competitiveness [2].

There are several parallel programming models that define different levels of parallelism, as well as different methods of communication between the computers involved in parallel processing. MPI (Message Passing Interface) is one of the specific programming model for developing applications with parallel programming. Its main task is to send a message in the domain of parallel data processing and is used as a medium for communication. Standard MPI was originally designed for writing applications and libraries for the environment with distributed memory. MPI provides functions and routines for the exchange of information required in a homogeneous and heterogeneous computing environment [3].

[*] Mr Luka Filipović, Center of Information Systems, University of Montenegro, Cetinjska 2, 81000 Podogrica, Montenegro, e-mail : lukaf@ac.me

[**] Prof. dr Božo Krstajić, Faculty of Electrical Engineering, University of Montenegro, Džordža Vašingtona bb, 81000 Podgorica, Montenegro, e-mail: bozok@ac.me

In a real-distributed environment (HPC cluster, grid cluster ...) workload of resources varies and it's not always possible to get for execution the resources that are completely free or equally burdened. The duration of the execution of parallel applications, which consist of unrelated tasks, is equal to the duration of the execution of set of tasks on the slowest core. Sometimes great losses may appear because the parallel application is waiting for slowest core or group of cores in order to complete all application tasks. This phenomenon occurs in clusters, which are composed of servers with different power (heterogeneous clusters) or clusters with a variable load, for example, the clusters where multiple users at the same time executes parallel applications which make different load of resources. In order to decrease looses, we used scheduling (load balancing) algorithms based on domain decomposition and master-slave paradigm to redistribute tasks and increase efficiency. Algorithms in this paper, static and dynamic, consider applications that consist of independent parallel tasks.

The paper will present the domain decomposition and master-slave algorithms for load balancing of applications parallelized in MPI technology and proposed modified version. Algorithm results will be compared with domain decomposition algorithm on the example of parallel CQ Simulator [4, 5]. Testing was performed on HPCG cluster which is located at the Institute of Information and Communication Technology in Sofia, Bulgaria. The cluster consists of 36 servers with 576 cores (dual Intel Xeon X5560@2.8Ghz) integrated in Blade Cluster HP Cluster Platform Express 7000 [6].

2. THE MPI PROGRAMMING MODEL

MPI [3] is the most widely used programming paradigm for developing applications in distributed memory parallel environment. The features of this model are imposed as standard for communication between processes in a parallel programming with distributed memory model. It belongs to the models of the transmission of messages and is meant to MIMD systems. MPI's features are high performance, good scalability and portability. It stands out as a dominant model, widely accepted and used in the high-performance computer systems. The MPI provides programming model where processes communicate with other processes by explicitly calling library routines to send and receive messages. The MPI standard provides library routines for multiple separate processes to collaborate and to communicate with each other. It include two-sided send/receive operations for exchanging data between process pairs, a variety of powerful collective operations, virtual topologies, and explicit grouping operations [7].

The advantages of the MPI programming model include the programmer's complete control over data distribution, process synchronization, explicit communication, and a permitting of the optimization of data locality [8]. This gives MPI programs high performance and scalability; however, it also has the drawback of making the MPI difficult to program and debug.

For the specification of the MPI model [9] is responsible MPI forum, open community with representatives from many organizations that define and maintain the MPI standard. The official languages of interface standards are FORTRAN, C and C++. Implementations can be found in C #, Java, Python, Perl and other languages.

3. SCHEDULING ALGORITHMS

Scheduling or load balancing algorithms are methods which determine an efficient execution order for a set of tasks of a given duration on a given set of execution units. Typically, the number of tasks is larger than the number of execution units. Since the number of execution units is fixed, there are also capacity constraints. Both types of constraints restrict the schedules that can be used. The overall goal of a scheduling algorithm is to find a schedule for the tasks which defines for each task a starting time and an execution unit such that the precedence and capacity constraints are fulfilled and such that a given objective function is optimized. Often, the overall completion time should be minimized. This is the time elapsed between the start of the first task and the completion of the last task of the program. Often, the number of processes or threads is adapted to the number of execution units such that each execution unit performs exactly one process or thread, and there is no migration of a process or thread from one execution unit to another during execution. [10].

Execution time of each parallel application is equal or dependent on the duration of slowest process or process on slowest core. During application runtime, computing core can be in active mode or idle state, waiting for others to finish their list of tasks. If it is determined that a particular core or group of cores longer period of parallel applications runtime are waiting for the others to finish the process, or working in stand-by mode, it is necessary to optimize the reallocation of tasks and execution process. Performance optimization of parallel applications can be done using load balancing algorithms by controlling tasks execution during application runtime.

Load balancing algorithms can be classified as static and dynamic.

A. Static scheduling

In static scheduling, the assignment of tasks to processors is performed before program execution begins. Information regarding task execution times and processing resources is assumed to be known at compile time. A task is always executed on the processor to which it is assigned. Typically, the goal of static scheduling methods is to minimize the overall execution time of a concurrent program and minimizing the communication delays. Static scheduling methods [11-14] attempt to: predict the program execution behavior at compile time, perform a partitioning of smaller tasks into coarser-grain processes in an attempt to reduce the communication costs and to allocate processes to processors. The major advantage of static scheduling methods is that all the overhead of the scheduling process is incurred at compile time, resulting in a more efficient execution time environment compared to dynamic scheduling methods. However, static scheduling suffers from many disadvantages, as discussed shortly [15].

One of the basic and simplest static scheduling algorithm is domain decomposition (DD) [16] or Probabilistic routing–First-Come-First-Served (PrFCFS) [17] algorithm. With this policy, tasks are dispatched to processors with equal probability. Cores for execution of each task are defined according random order or according pre-defined rules. The task dispatcher chooses one of the P processors based on the outcome of an independent trial in which the i_{th} outcome has probability $p_i = 1 / P$. Parallel applications using this algorithm are executing tasks on all cores. Large looses can be archived since during of all tasks is not known in advance.

B. Dynamic scheduling

Dynamic scheduling is based on the redistribution of processes among the processors during execution time. This redistribution is performed by transferring tasks from the heavily loaded processors to the lightly loaded processors with the aim of improving the performance of the application [18-21]. A typical load balancing algorithm is defined by three inherent policies: information policy, which specifies the amount of load information made available to job placement decision-makers; transfer policy, which determines the conditions under which a job should be transferred, that is, the current load of the host and the size of the job under consideration and placement policy, which identifies the processing element to which a job should be transferred.

The load balancing operations may be centralized in a single processor or distributed among all the processing elements that participate in the load balancing process. Many combined policies may also exist. For example, the information policy may be centralized but the transfer and placement policies may be distributed. In that case, all processors send their load information to a central processor and receive system load information from that processor. However, the decisions regarding when and where a job should be transferred are made locally by each processor. If a distributed information policy is employed, each processing element keeps its own local image of the system load. This cooperative policy is often achieved by a gradient distribution of load information among the processing elements [19]. Each processor passes its current load information to its neighbors at preset time intervals, resulting in the dispersement of load information among all the processing elements in a short period of time.

Random scheduling is an example of non-cooperative scheduling, in which a heavily loaded processor randomly chooses another processor to which to transfer a job. Random load balancing works rather well when the loads of all the processors are relatively high, that is, when it does not make much difference where a job is executed.

The advantage of dynamic load balancing over static scheduling is that the system need not be aware of the run-time behavior of the applications before execution. The flexibility inherent in dynamic load balancing allows for adaptation to the unforeseen application requirements at run-time. Dynamic load balancing is particularly useful in a system consisting of a network of workstations in which the primary performance goal is maximizing utilization of the processing power instead of minimizing execution time of the applications. The major disadvantage of dynamic load balancing schemes is the run-time overhead due to load information transfer among processors, decision-making process for the selection of processes and processors for job transfers and communication delays due to task relocation itself.

Master-slave (MS) paradigm, as one of the simpliest and most used dinamyc scheduling algorithms, involves two types of computing cores. On the master cores is performed preprocessing, postprocessing and task allocation. Task execution is performed on slave cores [22] . Master core, in the present case, at the beginning of the execution generates a list of tasks that need to be executed and sent one or more instructions to slave cores. Slave core, upon completion of given tasks, signals the end of assigned tasks and master core allocates them next list of tasks. This routine is repeated until all processes are finished. The advantage of the algorithm is reflected in a good management process. The lack of an algorithm is increased communication between the master and slave cores and potential

waiting of slave cores for allocation of new tasks for execution. Tasks cannot be executed on master core, so this is another disadvantage, especially during the execution on the smaller number of cores. In the case of the execution of applications on 64 core loss is only 1.5%, while the loss during the execution of the 8 cores increases to 12.5%.

4. MODIFIED MASTER-SLAVE ALGORITHM

With proposed algorithm, we tried to minimize imperfection and improve efficiency of master-slave algorithm combining domain decomposition and classic master-slave algorithm. Modified master slave algorithm (mMS) executes schedules tasks at begging using data decomposition algorithm, running tasks on all cores with minimum communication on beginning of parallel program. DD algorithm runs until completion of tasks execution on the fastest core. After that, MS algorithm is initialized and all cores are sending reports about unrealized tasks to master core. Master core are sending instructions to other (slave) cores and task are finishing using MS algorithm.

Task execution on all cores during most of the runtime corrects largest weakness of MS algorithm. Execution of MS algorithm at the end of process makes scheduling efficient and reduces time. Weakness of the algorithm is reflected in the short termination of work on all cores after receiving signal from fastest one. The proposed algorithm reduces communication between cores only the critical application period, i.e. after the end of the work the fastest core.

In ideal case, when cluster has balanced load and all cores finishes set of tasks at the same time, MS algorithm starts, verifies that all processes are executed and suspends operation of parallel applications. In this case, the losses are reduced to a few seconds during each core is sending report with list of completed processes.

5. SIMULATION RESULTS

Load balancing algorithms, described in chapter 4, are adapted for parallel applications which consists of multiple and independent tasks. Algorithms were tested on performance analysis simulator of the crosspoint-queued switch [4], but it can be applied on all parallel applications which workflow can be divided into multiple tasks.

Switches with small buffers in crosspoints have been evaluated in the late Eighties, but mostly for uniform traffic. However, due to technological limitations of that time, it was impractical to implement large buffers together with switching fabric. The crosspoint queued switch architecture has been recently brought back into focus since modern technology enables an easy implementation of large buffers in crosspoints. An advantage of this solution is the absence of control communication between line cards and schedulers [23].

Application for performance analysis CQ switch consists of two parts - the uniform traffic generator and simulators of CQ switch, parallelized with MPI. Traffic generator generates uniform traffic for 32 different input loads. The simulator performs analysis of 12 different buffer sizes and 8 algorithms (LQF, RR, ERR, FBRR, EELQF, ELQF, FBLQF and RAND) on 32 files of generated uniform traffic. In preprocessing, simulator generates

the 3072 independent tasks which are executed using MPI on different cores using different algorithms. Testing was performed for matrix of 16x16 and 1.000.000 time slots.

Tests were performed on 64 cores using 4, 8 or 16 cores per server which executes application. Average execution time and standard deviation of 10 measurements for 3 algorithms is shown at Fig. 1.

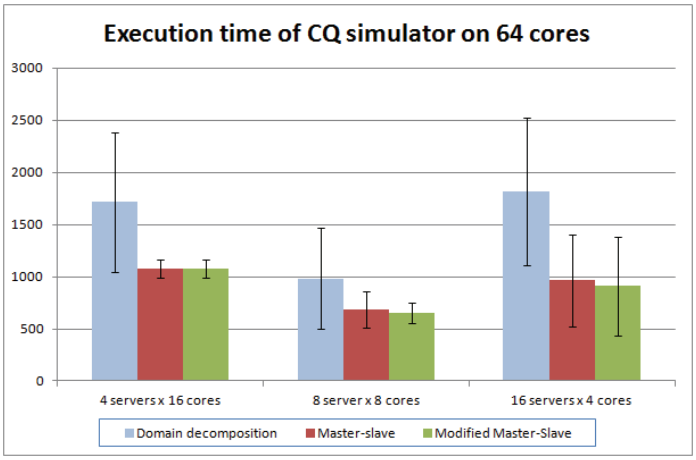

Fig. 1. Average execution time of parallel application using 3 algorithms and 64 computing cores

Results at Fig. 1 shown that domain decomposition algorithm had worst results and in general weakness of static scheduling. Master-slave algorithm gave better results in all segments. Proposed improved master-slave algorithm gave same results as master-slave or better (using 8 or 16 servers). We notice the greatest deviation of measurements results using the domain decomposition algorithm.

All algorithms gave best results using 8 servers with 8 cores. Application running on 4 server worst time of execution due enabled hyper-threading on server and increased load inside server, while execution on 16 servers with 4 cores had looses due time lost in communication and balancing load. Largest deviation of results was observed on execution using 16 servers.

A. Execution analysis of load balancing algorithms

In the remainder of this paper, we will analyze one case of execution on 64 cores, using 16 servers with 4 cores per server and scheduling algorithms described in chapter 4. We performed an analysis of the task execution time on each core, not just the slowest, which initiates the postprocessing and end of parallel application.

Domain decomposition algorithm	Master-slave algorithm	Modified master-slave algorithm
1889.86	1016.85	991.69

Table 1. Execution time using DD, MS and mMS algorithm on 64 cores

Table 1 shows execution time of 3 scheduling algorithm for CQ simulator, executed on 64 cores. Domain decomposition algorithm shown worst results where all computing cores got same number of tasks to finish computation. Master slave algorithms, classic and improved, shown better results and reduced execution time up to 48%.

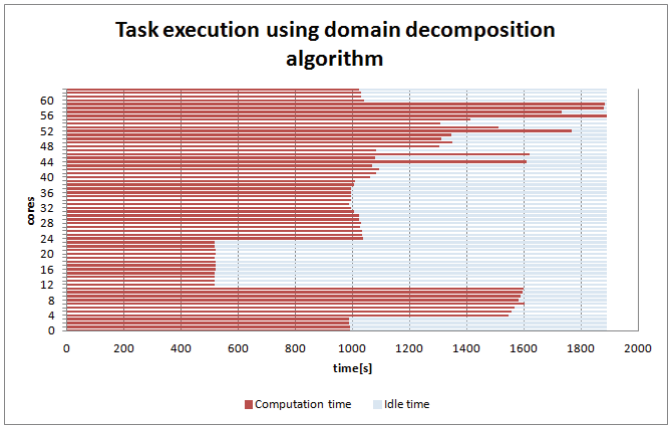

Figure 2. Distribution of the CQ Simulator tasks using domain decomposition algorithm

Figure 2 shows analysis of execution time on each core using domain decomposition algorithm. Red color illustrates computation time of each process. Total execution time and execution time on slowest core was 1889.86 seconds, while the fastest core finished their assigned tasks in 512.68 seconds (illustrated by yellow line of Figure 2). Execution time per core varied depending on the current server load, storage load and complexity of assigned analysis. Blue color represents idle time for each process, time spent in waiting for other processes to finish their assignment. In this example, CPU time was 33.57 CPU hours, while the time of active computation was 19.84 CPU hours (59%). The duration of individual processes varied from 9.71 s to 46.95 s

Figures 3 and 4 shown distribution of computational time on each core using master-slave algorithm and modified master-slave algorithm.

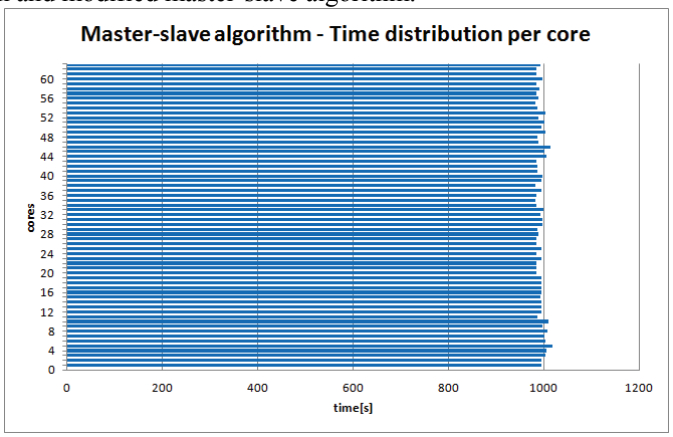

Fig 3. The distribution per core over time with the use of master-slave algorithm

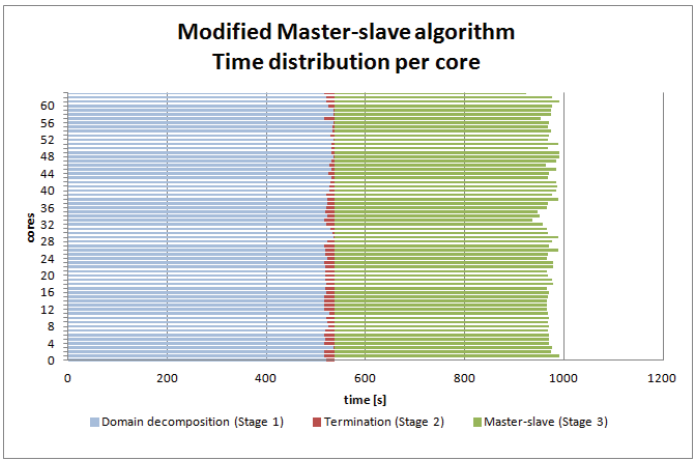

Fig 4 Distribution of the process execution per core over time using a modified master-slave algorithm

Core No. 0 was used as master core in both master-slave algorithms. Master-slave algorithm reduced duration of parallel CQ simulator to 1016.85 seconds, or 18.06 CPU hours with utilization 96.07 %. Master core was used only for communication and monitoring of computation on other cores and caused the loss of 0.28 CPU hours or 1.56% of utilization.

Execution time of CQ simulator using modified master-slave algorithm was 991.69 seconds, or 17.63 CPU hours. Utilization of cluster was increased to 96.29%. Program execution was divided in three stages. In first stage, application was running using domain decomposition algorithm. The fastest core finished assigned tasks in 515.92 seconds and sent signal to other to terminate their work. In second phase (illustrated by red color) remaining cores received termination signal, finished current tasks, made synchronization and sent report about unfinished tasks to master process. Duration of second phase was up to 21.08 seconds and produced loss of 0.23 CPU hours or 1.32% of utilization. Measurement shown that the maximum time the interruption of application (stage 2) lasted as long as the maximum duration of the individual processes that run on the cores at the time of sending a signal to terminate the execution. After that, application continued execution using master-slave algorithm (stage 3) and finished all tasks in 454.69 seconds. Looses form master core in stage 3 were reduced to from 0.28 CPU hours form standard master-slave algorithm to 0.12 CPU hours. These savings would be greater in examples where the master-slave algorithm part (stage 3) performs in shorter time interval or in examples ran on smaller number of cores.

6. CONCLUSION

In this paper, we propose modified master-slave algorithm for load balancing in MPI parallel applications. We combined domain decomposition and master-slave algorithm, as a

examples of static and dynamic scheduling algorithms, and made proposed algorithm. Modified algorithm improves task scheduling during critical part of parallel application.

Numerical example approved improvement of utilization during application execution with proposed algorithm. It was confirmed that the termination for synchronization do not significantly affect the total execution time.

REFERENCES

[1] D. Peleg, "Distributed Computing: A Locality-Sensitive Approach", *Society for Industrial and Applied Mathematics (SIAM)*, 2000, pp 1-11, ISBN 0-89871-464-8

[2] A. S. Tanenbaum, M. van Steen, *Distributed Systems: Principles and Paradigms, 2nd edition*, 2007, ISBN 0132392275 / 9780132392273

[3] W. Gropp, E. Lusk, A. Skjellum, "Using MPI-2: Advanced Features of the Message Passing Interface", *MIT Press*, 1999, ISBN 0-262-57133-1.

[4] M. Radonjić, "Prilog analizi performansi CQ komutatora paketa sa stanovišta veličine i algoritama upravljanja redovima čekanja", Ph.D. dissertation, Faculty of Electrical engineering, University of Montenegro, 2011.

[5] L. Filipović, "Optimizacija simulatora CQ komutatora paketa metodom paralelnog programiranja", IT 2012, Zabljak 2012.

[6] A. Mishev, E. Atanassov, "Infrastructure overview and assessment", HP-SEE project deliverable D 5.2, pp. 21-24, Available : http://www.hp-see.eu/files/public/HPSEE-WP5-MK-001-D5.2-f-2011-08-31.pdf

[7] Z. Bozkus, "Hybrid MPI+UPC parallel programming paradigm on an SMP cluster", *Turk J Elec Eng & Comp Sci*, Vol. 20, NoSup. 2, TUBITAK, doi:10. 3906/elk-1103-11, 2012

[8] W. Gropp, E. Lusk, A. Skjellum, "Using MPI-portable parallel programming with the Message-Passing Interface", *Scientific Programming*, Vol. 5, pp. 275–276, 1996.

[9] D. W. Walker. "Standards for message-passing in a distributed memory environment", Technical report, Oak Ridge National Laboratory, August, 1992. First CRPC Workshop on Standards for Message Passing in a Distributed Memory Environment, April 1992.

[10] T. Rauber, G. Rünger, *Parallel Programming: for Multicore and Cluster Systems Hardcover*, ISBN: 978-3642048173

[11] Lo, V. M., "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans. Computers*, Vol. C-37, No. 11, Nov. 1988, pp. 1384-1397.

[12] Sarkar, V., "Partitioning and Scheduling Parallel Programs for Multiprocessors", *MIT Press*, Cambridge, Mass., 1989.

[13] Shirazi, B., M. Wang, G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *J. Parallel and Distributed Computing*, Vol. 10, 1990, pp. 222-232.

[14] Stone, H. S.,"Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, Jan. 1977, pp. 85-93;

[15] B. A. Shirazi, K. M. Kavi, A. R. Hurson, *Scheduling and Load Balancing in Parallel and Distributed Systems*, ISBN: 0818665874

[16] W.D. Gropp, *Parallel Computing and Domain Decomposition*, Fifth Conference on Domain Decomposition methods for Partial Differential Equations, SIAM, 1990

[17] H. Karatza, R. Hilzer. 2003. "Parallel job scheduling in homogeneous distributed systems", *Simulation 79,* pp. 287-98.

[18] Eager, D. L., E. D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE, Trans. Software Eng.*, Vol. SE-12, No. 5, May 1986, pp. 662-675

[19] Lin, F. C. H., and R. M. Keller, "The Gradient Model Load Balancing Method," *IEEE Trans. Software Eng.*, Vol. SE-13, No. l,Jan. 1987, pp. 32-38.

[20] Shivaratri, N. G., P. Kreuger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer*, Vol. 25, No. 12, Dec. 1992, pp. 33-44; reprinted here.
[21] Wang, Y.-T., and R. J. T. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. Computers*, Vol. C-34, No. 3, Mar. 1985, pp. 204-217.
[22] S Sahni, G Vairaktarakis, "The master-slave paradigm in parallel computer and industrial settings", *Journal of Global Optimization*, April 1977, ISBN 0925-5001.
[23] M. Radonjic, I. Radusinovic, "Impact of scheduling algorithms on performance of crosspoint-queued switch", *Annals of Telecommunications*, Vol 66, No 5-6, May/June 2011, pp.363-376, ISSN: 0003-4347