Mathematical Problems of Computer Science 25, 2006, 18–26.

Dynamic Process Management System Architecture for Computational Clusters

Tigran M. Grigoryan, Vladimir G. Sahakyan

Institue for Informatics and Automation Problems of NAS of RA
e-mals tigrangr@ipia.sci.am, svlad@sci.am

Abstract

The problem of efficient utilization of computational resources of clusters arises as its load and number of users grow. Tasks like the fair use of resources and load balancing are common and should be solved by the operation environment of cluster. Existing mechanisms that solve the mentioned problems work fine as long as parallel programs are run on a fixed number of resources.

Allocating and freeing resources dynamically can highly improve the performance of a parallel program as well as the efficiency of using the cluster. In the following paper the system architecture is described, which supports dynamic resource allocation and process spawning, which is alternate to MPI-2 standard's dynamic process spawning mechanism. It is also introduced, how dynamic task/process spawning can improve the performance of the parallel program.

1 Standard Process Management Schemes for Computational Clusters

The functionality of dynamic process spawning, introduced in MPI-2, provide MPI programmers with means for spawning new processes and parallel programs and establish communication with them dynamically during the execution of the parallel program. It, in fact, also has a number of hidden problems and in some cases can cause non-efficient use of cluster resources or lead to speeding down the performance of parallel program. To understand these weaknesses, let's consider a standard configuration of a computational cluster, consisting of an MPI library implementation for supporting interprocess communication and a task queuing system for supporting task execution control and resource allocation for tasks, and try to analyze the use of MPI-2's dynamic processes functions [1] in this kind of environment. To be more specific, let's consider LAM/MPI [3] package as an MPI library implementation (for its support of dynamic processes functions) and PBS [6]as a task queuing system (for its being a de-facto standard in task queuing systems). Other packages are functionally similar to the selected ones so they can be easily substituted.

To run a parallel MPI program in LAM/MPI environment, first of all a LAM run-time environment or so-called "LAM Universe" has to be initialized on the nodes, where the program should run. The initialization of LAM run-time environment also referred to as "booting LAM". All the MPI functions, called from the parallel program, are executed in the framework of LAM Universe, in which the program runs. Particularly that means the calls to process-spawning functions will result, and which is natural, in spawning of new processes within the same LAM Universe.

Booting LAM is performed by a program from LAM/MPI package called lamboot. Lamboot works in two modes, rsh/ssh and tm [4, 5]. In rsh/ssh mode lamboot must be provided with a list of nodes, on which LAM Universe should be initialized, while in tm mode it will obtain the node list form PBS through PBS's task management (tm) API.

When the parallel program is started from PBS, first of all PBS script should boot LAM on the nodes assigned to that program by scheduler, resulting in a LAM Universe working under PBS. After that the program can run in the newly created LAM Universe. In this scenario everything is good as far as processes are not spawned dynamically. When trying to spawn a process from the working parallel program, it has to be spawned on a cluster virtual processor belonging to the same universe, thus, resulting in overloading cluster node, if all the virtual processors in the universe already handle processes. On the other hand, if not all of the virtual processors were used by the time a process dynamically spawned, resource wasting had taken place. This fact can result in non-efficient overall usage of a computational cluster when some of the computational resources overloaded while other ones are wasted.

2 Cluster management and use model

The above described problem resides not in the implementations of MPI, but in the MPI standard itself. The aim of MPI is to provide a portable interprocess communication mechanism, and the tasks such as process and resource management are beyond of MPI. A cluster operational and functional environment, built on existing packages, described in the previous section, does not provide with means for effective process or resource management from inside of the working parallel program, and lack of such functionality is obvious. This section is dedicated to development of system architecture for computational clusters, which will support message passing and task management as well as provide mechanism for resource allocating and freeing, resource state querying, dynamic process creation with taking into account the overall load of cluster nodes and restrictions and establishing communication between newly create processes.

Cluster management and use model, shown on fig.1, is three-tier system architecture, providing on each layer a new type of functionality.

On the MPI level all interprocess functionality is available in the framework of a single parallel program. MPI layer can be implemented using an MPI-1 library as well as MPI-2 library, meaning that process spawning will not be handled in this layer. The functionality of interprocess communication layer is accessible from parallel program using MPI library functions.

The queuing system layer handles all the resource and parallel jobs management as well as user management and enforces predefined policies on them. The functionality of this layer is accessible to cluster users through a set of command line utilities. On standard cluster system architectures functionality of this layer is provided by packages such as PBS. Queuing system API layer provides functionality for checking the availability of resources, allocating and freeing them and spawning new processes on them. It also includes functions for communication between newly spawned processes, checking the execution status of processes, retrieving results of execution of a process, etc. Being strongly coupled with the queuing system layer this one sends all the requests for spawning processes to the queuing system, avoiding form overloading computational resources. This layer is accessible from the parallel program through its library functions. There is no substitute for this layer in the cluster system architecture described in the previous section.

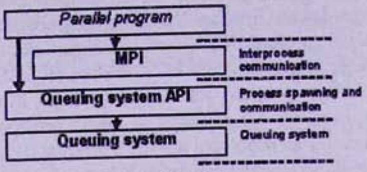

Fig. 1

Described three-tier system architecture solves the problem of non-efficient use of cluster resources, because it provides means for the parallel program to access "the outer world". More this architecture takes care that all process management is being done through the queuing system, which makes it impossible to load a computational resource more than the policy enforced by queuing system states.

Of course, the presented method is not the best solution for all of the parallel programming tasks, which need dynamic processes, but it improves the performance of some kind of parallel programs and the efficiency of using computational cluster as it is shown in the section below.

3 Efficiency of Using Dynamic Resource Allocation with Process Spawning

Consider a parallel program and its oriented macrograph, which does not contain strongly-coupled isolated submacrographs having more than one node [7]. In that kind of graph each node represents a separate calculation or a subtask that takes arguments from input or from the results of previously completed calculations and produces some results, that can be used by other calculations. Lest divide nodes of graphs into groups in following way: put two nodes in a group if and only if they does not have a connecting oriented path. We will get a "layered" graph containing k layers. Lets name the subtasks in the first layer A_{11}, A_{12}, $\dots A_{1n_1}$, in the second layer A_{21}, A_{22}, $\dots A_{2n_2}$, etc., in the k-th layer A_{k1}, A_{k2}, $\dots A_{kn_k}$ (fig. 2).

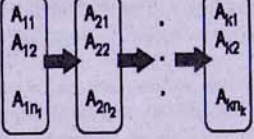

Fig. 2

Obviously, for completing all the subtasks in the i-th layer ($i > 1$), it is necessary that all the tasks in the $i-1$-st layer have been completed. Let's denote the number of processors, needed for execution of task A_{ij} through p_{ij} and the amount of time, needed for its completion through t_{ij}. In this section some possibilities will be introduced for programming and executing such kind of parallel tasks, which can be decomposed into a set of weakly-coupled smaller parallel tasks (subtasks). Different scenarios of running such a program are possible, each having its advantages and disadvantages:

1. Run the whole program at once and execute subtasks in each sequential section sequentially
2. Run the whole program at once and execute subtasks in each sequential section in parallel
3. Run all the subtasks sequentially
4. Run subtasks in each sequential section in parallel, running the sections sequentially
5. Use dynamic resource allocation

Let's define function C as following

$$C = \tau(T) + \pi\xi$$

where $\tau(t)$ is a non-decreasing function (if $t_1 > t_2$ then $\tau(t_1) \geq \tau(t_2)$) showing the cost of waiting for time t, T is the amount of time needed by the program to complete, π is the cost of using one processor for a unit of time and ξ is the amount of used resources, being a product of the number of used processors and time of use. This function represents the cost for running parallel program. In the cases 3, 4 and 5 a parameter θ should also be considered, showing the time needed to spawn a program through the queue management system. The function $\tau(t)$ depends on the parallel program (in other words it is a program-specific parameter), and the values of π and θ depend on the cluster configuration and usage policy (they are cluster-specific parameters).

Let's calculate the cost C of execution of program for each scenario, considering π as the cost of using one processor for a unit of time and $\tau(t)$ as a function showing the cost of waiting for time t.

I. In the first scenario the number of processors P_1 to be reserved and the amount of time T_1 of the program execution will be described by the following equations

$$P_1 = \max_{\substack{1\leq i\leq k\\ 1\leq j\leq n_i}} (p_{ij}) \quad , \quad T_1 = \sum_{i=1}^{k}\sum_{j=1}^{n_i} t_{ij}$$

The cost of execution will be expressed through the following expression

$$C_1 = \tau(T_1) + \pi P_1 T_1 = \tau\Big(\sum_{i=1}^{k}\sum_{j=1}^{n_i} t_{ij}\Big) + \pi \max_{\substack{1\leq i\leq k\\ 1\leq j\leq n_i}} (p_{ij}) \sum_{i=1}^{k}\sum_{j=1}^{n_i} t_{ij}$$

II. In the second scenario values of P_2 and T_2 will be respectively

$$P_2 = \max_{1\leq i\leq k}\Big(\sum_{j=1}^{n_i} p_{ij}\Big) \quad , \quad T_2 = \sum_{i=1}^{k} \max_{1\leq j\leq n_i} t_{ij}$$

and the cost of execution will be

$$C_2 = \tau(T_2) + \pi P_2 T_2 = \tau\Big(\sum_{i=1}^{k} \max_{1\leq j\leq n_i} t_{ij}\Big) + \pi \max_{1\leq i\leq k}\Big(\sum_{j=1}^{n_i} p_{ij}\Big) \sum_{i=1}^{k} \max_{1\leq j\leq n_i} t_{ij}$$

III. In this case the amount of processors used by program is variable and depends on a particular subtask that is currently executed. So, the value of P_3 will vary during the time and $P_3 \in \{p_{ij} | 1 \leq i \leq k, 1 \leq j \leq n_i\}$. The overall time, spent for completing the execution of the whole program will be

$$T_3 = \sum_{i=1}^{k}\sum_{j=1}^{n_i} t_{ij} + (\sum_{i=1}^{k} n_i - 1)\theta$$

The cost in this case will be expressed as

$$C_3 = \tau(\sum_{i=1}^{k}\sum_{j=1}^{n_i} t_{ij} + (\sum_{i=1}^{k} n_i - 1)\theta) + \pi \sum_{i=1}^{k}\sum_{j=1}^{n_i} t_{ij}p_{ij}$$

IV. In this case also the number of processors will be variable and will change through the time, taking values $P_4 \in \{\sum_{j=1}^{n_i} p_{ij} | i = 1, \ldots, k\}$. The execution time of the overall program will be

$$T_4 = \sum_{i=1}^{k} \max_{1 \leq j \leq n_i} t_{ij} + (k-1)\theta$$

The cost of execution will be

$$C_4 = \tau(\sum_{i=1}^{k} \max_{1 \leq j \leq n_i} t_{ij} + (k-1)\theta) + \pi \sum_{i=1}^{k}\sum_{j=1}^{n_i} (p_{ij} \max_{1 \leq j \leq n_i} t_{ij})$$

V. When using dynamic resource allocation, processors will be reserved by the program as they needed and will be released as they become idle, so in this case also the number of processors P_5 will change dynamically during the execution of program. Depending on cluster load, the real time of program execution will be between its maximum and minimum values, which are

$$T_5^{max} = \sum_{i=1}^{k}\sum_{j=1}^{n_i} t_{ij} + (\sum_{i=1}^{k} n_i - 1)\theta$$

$$T_5^{min} = \sum_{i=1}^{k} \max_{1 \leq j \leq n_i} t_{ij} + (k-1)\theta$$

Based on this expressions for maximal and minimal times, the maximal and minimal costs of execution can be calculated

$$C_5^{max} = \tau(\sum_{i=1}^{k}\sum_{j=1}^{n_i} t_{ij} + (\sum_{i=1}^{k} n_i - 1)\theta) + \pi \sum_{i=1}^{k}\sum_{j=1}^{n_i} p_{ij}t_{ij}$$

$$C_5^{min} = \tau(\sum_{i=1}^{k} \max_{1 \leq j \leq n_i} t_{ij} + (k-1)\theta) + \pi \sum_{i=1}^{k}\sum_{j=1}^{n_i} p_{ij}t_{ij}$$

Depending on cluster load, the cost of execution C_5 using dynamic resource allocation will vary between these two values, $C_5^{min} \leq C_5 \leq C_5^{max}$

Having the function $\theta(t)$, the values of π and θ, a parallel program and its decomposition into subtasks and having the values of p_{ij} and t_{ij} for each of the subtasks it is possible to calculate the costs of execution for the cases described above and find a case, in which the cost of execution is minimal.

It is obvious that $C_5^{min} \leq C_4$ and $C_5^{max} = C_3$ meaning that cost of execution in cases 3 and 4 can be greater or equal to that in case 5. It is also obvious that depending on value of θ and nature of function $\tau(t)$ the cost provided by the fifth case can be lower than the one provided by first or second cases. For example, if the time of spawning a parallel program through a queueing system is considered to be 0 ($\theta = 0$), then $C_5^{max} \leq C_1$ and $C_5^{min} \leq C_2$, meaning that the cost of program execution in case 5 can be lower or equal than that in cases 1 or 2.

4 Structure of System for Dynamic Resource Allocation and Process Management

To support the system architecture described above a System Dynamic Resource Allocation and Process Management has been designed. The system provides functionality for queuing system and queuing system API levels, shown on fig.1. It consists of three main pieces: main module, API library and user interface commands.

Most of the functionality of the system is encapsulated inside the main module. It is responsible for management of task queues, running tasks and collecting results, computational resources management, user management, enforcing policies, etc. These responsibilities are distributed between four major subsystems of main module: Queue Management Subsystem, Task Management Subsystem, Process Management Subsystem and Request Processing Subsystem (fig. 3). Besides the concepts, which are used when working with regular queuing systems (such as "job" or "task", "queue", "virtual processor"), dynamic resource allocation system introduces its own specific concepts: task context, context size, task rank and context attributes.

Task context is a virtual environment, in which the task is being executed. For each separate task, submitted by user, a new context will be created for task execution. When processes (task) spawn from parallel program, they are executed as a separate task in the context of spawning task. Context size is the number of tasks executing in the context. Each task has its unique identifier in its context, called task rank. On creating context, its size always has the value of 1, and the original task, for which the context was created, always has rank of 0 in that context. Tasks, executed in the same context can communicate through the context, using each other rank's for addressing. Two mechanisms of communication between tasks are possible: active communication and passive communication. In active communication send/receive mechanism is used, like in MPI's point-to-point communication. Passive communication deals with so called context attributes, using set/get mechanism. Simpler, a task can set an attribute in the context, giving it a name, and another task after some time can check on the availability of the attribute with a given name in the context and retrieve its value.

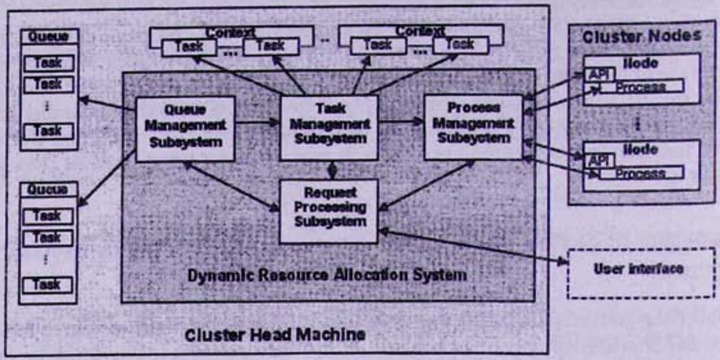
Queue
Task
Task
Task
Queue
Task
Task
Task
Context
Task
Task
Context
Task
Task
Queue Management Subsystem
Task Management Subsystem
Process Management Subsystem
Request Processing Subsystem
Dynamic Resource Allocation System
Cluster Head Machine
Cluster Nodes
Node
API
Process
Node
API
Process
User interface

Fig. 3

API library provides a set of functions, allowing the programmer to interact with "outer world" from parallel program in the framework of its context. These functions include functions for querying the queuing system, functions for active and passive communications and functions for allocation of computational resources and spawning processes as new tasks through the queuing system.

User interface provides users with a set of command line utilities, which allow manipulating tasks, viewing information about queues and task status.

The request processing subsystem provides a unified querying interface for main module. It handles the execution of all of the incoming requests to the main module, whether from the user interface or API library.

Queue management subsystem handles task queues and resource management. It also provides scheduling capabilities for mapping computational resources to the job by using preset policies. The policies used by queue management subsystem are set through configuration files and include the resources accessible from each queue, the algorithms for each queue according to which the tasks in that queues are served, etc. This subsystem can be implemented using an existing queuing system such as PBS. Task management subsystem provides task context abstraction and encapsulates a mechanism for the task to interact with the context and with the other tasks in the same context. It is also responsible for creating and destroying contexts, launching and killing tasks and collecting task resource usage statistics and the results of its execution.

Process management subsystem is responsible for immediate spawning of processes of a parallel program. It also serves as an interface to which API library sends its requests and from where it receives responses and events. The implementation of this subsystem is more likely to use a process spawning mechanism provided by the MPI package.

The description of the API library and user utilities is given in [8], where a simple implementation of such a system is also described.

5 Conclusion

The use of dynamic resource allocation in some cases can lower the cost of program execution on cluster. Depending on the program and its algorithm, that difference in costs can be really valuable. The created system for dynamic resource allocation allows to benefit from the decomposition of a parallel task into sequential groups of subtasks by providing a mechanism for spawning a new parallel program from a running one through the queue management system.

References

[1] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface (http://www.mpi-forum.org/docs/mpi-20.ps)

[2] W. Gropp, E. Lusk. Dynamic Process Management in an MPI Setting; Mathematics and Computer Science Division Argonne National Laboratory, 1995

[3] G. Burns, R. Daoud, J. Vaigl. LAM: An Open Cluster Environment for MPI; Proceedings of Supercomputing Symposium, pp. 379–386; 1994

[4] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI; Proceedings, 10th European PVM/MPI Users' Group Meeting, pp. 379–387; 2003

[5] B. Barrett, J. M. Squyres and Andrew Lumsdaine. Integration of the LAM/MPI Environment and the PBS Scheduling System; Proceedings of the 17th International Symposium on High Performance Computing Systems and Applications and OSCAR Symposium, pp. 277–283; 2003

[6] A. Bayucan, R. L. Henderson, J. P. Jones, C. Lesiak, B. Mann, B. Nitzberg, T. Proett, J. Utley. Portable Batch System, OpenPBS Release 2.3, Administrator Guide; Veridian Information Solutions, Inc., 2000

[7] В.В. Воеводин, Вл.В. Воеводин. Параллельные вычисления; "БХВ-Петербург", Санкт-Петербург, 2004.

[8] T. Grigoryan, V. Sahakyan. Dynamic Resource Manager for Clusters. Proceedings of CSIT2005, pp. 439–442; Yerevan, 2005

Հաշվողական կլաստերների դինամիկ ընթացքների ղեկավարման համակարգային ճարտարապետություն

Տ. Մ. Գրիգորյան

Ամփոփում

Կլաստերային համակարգի ծանրաբեռնվածության և օգտագործողների քանակի աճի հետ մեկտեղ առաջ է գալիս դրա հաշվողական ռեսուրսների էֆեկտիվ օգտագործման պրոբլեմը: Առաջանում են ռեսուրսների «ագահ» օգտագործման և բեռնվածության հավասարակշռման խնդիրները, որոնք պետք է լուծվեն կլաստերի օպերացիոն միջավայրի կողմից: Գոյություն ունեցող մեխանիզմները ապահովում են այդ խնդիրները լուծումը քանի դեռ զուգահեռ ծրագրերը աշխատում են ֆիքսված քանակությամբ պրոցեսորների վրա:

Հաշվողական ռեսուրսների դինամիկ զբաղեցումը և ազատումը կարող է զգալիորեն բարձրացնել ինչպես զուգահեռ ծրագրի արտադրողականությունը, այնպես էլ կլաստերի օգտագործման էֆեկտիվությունը: Սույն հոդվածում նկարագրված է հաշվողական ռեսուրսների դինամիկ զբաղեցում և դինամիկ ընդացքների սերում ապահովող կլաստերի համակարգային ճարտարապետություն, որը կարող է այլընտրանք հանդիսանալ MPI-2 ստանդարտի դինամիկ ընդացքների սերման մեխանիզմին: Ցույց է տրված նաև, թե ինչպես դինամիկ սերվող ընդացքների կիրառումը կարող է բարձրացնել զուգահեռ ծրագրի արտադրողականությունը: