Load Balancing MPI Algorithm for High Throughput Applications

Igor Grudenić, Stjepan Groš, Nikola Bogunović
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
{igor.grudenic, stjepan.gros, nikola.bogunovic}@fer.hr

Abstract. *MPI (Message Passing Interface) is a standard API (Application Programming Interface) used to create distributed applications. Development of distributed MPI software is a challenging task in which many difficulties may arise. Common problems include load balancing, deadlocks and livelocks.*
In order to tackle complexity of parallel programming various methods are developed. In this paper we present a load balancing algorithm (LBA) that can be used instead of developing custom parallel application. The algorithm uses MPI transport mechanism for internal signaling and synchronization. The LBA is most suitable for the tasks that involve high and continuous data throughput. It performance will significantly degrade if used in complex computation topologies.

Keywords. Distributed computing, MPI, load balancing, scheduling.

1. Introduction

Distributed computing became more popular in the last decade due to advances in hardware and communication networks. The increase in computing power enabled very complex computations to be accomplished within reasonable time.

In order to gain performance in distributed environment, traditional sequential algorithms must be redesigned. Some algorithms are inherently parallel or can be parallelized without compromising performance [2][10], while some other cannot benefit from distributed environment.

There are two basic programming paradigms used in the development of distributed software: shared memory and message passing. The message passing model is much more scalable, but its exploitation raises issues such as deadlock and livelock occurrence, data corruption, and performance problems.

There are a number of methods proposed to ease development of message passing applications. These approaches include parallelizing compilers [1], modeling and verification tools [11][5], performance analyzers [9], run-time checkers/analyzers [6][7] and parallel debuggers [13].

In this paper we provide a message passing algorithm (LBA) that can be generally used to schedule multiple small independent jobs with high data throughput. Typical applications include high volume data compression, video editing, encoding and decoding, 3d scene rendering, etc.

Cluster schedulers dedicated for cluster resource management [4][8] are not suitable for this task because they operate on a large scale and are optimized for long jobs with moderate size of input and output data. Additionally, network utilization is not optimized by cluster schedulers and availability of a job input data is not optimized to match the availability of the computation resources.

There is [12] scheduler designed for batch queuing targeted at high throughput of short batch jobs which is designed to work with shared high performance storage so it disregards the data availability issue.

In section 2 we provide description of the LBA architecture, as well as possibilities for its expansion. LBA scheduling algorithm that implements part of the LBA architecture is presented in section 3. In section 4 we describe test scenario for the implemented algorithm together with performance results.

2. LBA Architecture

LBA Architecture is designed to provide an easily extendable basis for parallel computation. It consists of a single element which enables parallel computation and pipe and filter architecture that can be used to chain the computation of multiple single elements in

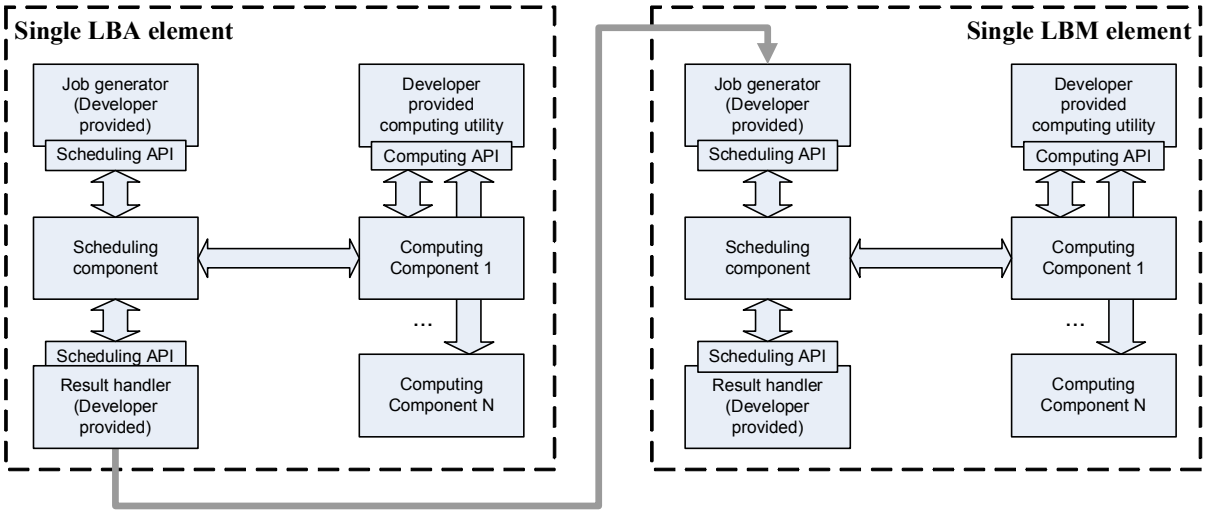

Figure 1. LBA Architecture

complex data processing scenarios. Single LBA element is described in section 2.1 and composition of elements is outlined in section 2.2.

2.1. Single LBA element

Single LBA element (Fig. 1) contains scheduling component and computation components. Scheduling component is used to schedule jobs and dispatch data to computing components. Computing components are designed to perform application logic to receive input data and to return computation results to the scheduling component. Various statistical data are returned together with the results to the scheduling component in order to support scheduler decision making process. Scheduling component and each of the compute nodes are executed at the separate MPI nodes.

In order to make LBA architecture generic and easily extendable two APIs are defined: scheduling API and computing API.

The scheduling API is used to provide support for complex job dependencies as well as to enable custom scheduling policies to be used. Complex job dependencies are managed by the developer provided job generator connected to the scheduler using the scheduling API. The scheduler gets a list of available jobs from the job generator and assigns them together with accompanying data to the computing components. Computing components execute the assigned job through the provided computing utility and send the result to the scheduling component. The scheduling component forwards the received result to the provided result handler.

The result handler is developer provided and can be connected with the job generator within the same single LBA element in order to furnish new data for computation or to enable new jobs for execution. This enables developer to create and enforce arbitrarily complex job dependencies that are needed to complete high level distributed task.

2.2. Pipes and Filters

Single elements can be composed into complex topologies using well-known pipe and filter architecture [3]. Single LBA element is used as a filter and underlying data transfer (pipe) can be implemented by technology available on the given computer cluster.

Data flow between two LBA elements is accomplished by connecting result handler in the source LBA element and job generator in the sink LBA element.

Startup and data transfer between LBA elements are two main issues in the proposed pipe and filter architecture. In the MPI environment all filters (and the corresponding scheduling and computing components) must be assigned to MPI compute nodes. The component that determines the layout of the entire system should be provided by the developer. Data transfer between LBA components can be done using MPI or other provided protocol. MPI transfer is potentially unsafe because there are threading issues in MPI implementations and it is assumed that job generator, scheduling component and result handler are running in separate execution threads.

3. LBA Scheduling Algorithm

While implementing LBA element we have provided a simple scheduling algorithm designed to perform well based on the following assumptions:

1. jobs are independent
2. input data is much larger then the output data
3. all the compute nodes are able to execute any of the jobs
4. compute nodes do not have the same processing power (heterogenous environment)

Signaling between scheduling component and computing component is done using MPI. This includes information about status of the compute nodes, together with the information on the jobs pending. The transfer of input and output job data is accomplished using operating system provided resources and is configurable.

The algorithm for scheduling component is given in section 3.1, while the algorithm for computing component and computing utility are presented in section 3.2.

3.1. Scheduling component algorithm

Scheduling component (Fig. 2) is designed to run while there are jobs available. After the job is acquired from the job generator, a target compute node (*targetNode*) must be determined. Before the decision, the scheduling component updates its information with the statistical data that are queued up from the compute nodes. After the update of the *info* object, the best compute component for the job is picked up (*findBestNode*). *Info* object contains all the information about the status of the compute components.

Finding the best compute node for the job is completed in two phases. In the first phase computing components not suitable for the job are filtered out. The criteria for elimination of compute nodes include inadequate storage size for input data, number of jobs pending for execution on compute nodes and amount of input data that should be transferred for pending jobs. In the second phase the best node is picked using similar policy. In the case of all compute components filtered out in the first phase, the algorithm continues to gather statistics until at least one compute component becomes eligible for the job.

```
Scheduling component{
  while(jobsAvailable()){
    job=nextjob();
    targetNode=none;
    while(targetNode==none){
      info+=getStatistics (compute_nodes);
      targetNode=findBestNode(job,info);
    }
    initiateDataStagingIN(job,targetNode);
    initiateDataStagingOUT(completedJobs);
    while(existsFinishedStaging()){
      (job,target_node)=getStaged();
      Notify(job,target_node);
    }
  }
  while(!allJobsCompleted(info))
    while(existsFinishedStaging()){
        (job,target_node)=getStaged();
        Notify(job,target_node);
    }
    info+=getStatistics (compute_nodes);
  }
}
```

Figure 2. Scheduling component algorithm

When the job is matched with a compute component, data staging is initiated (*initiateDataStagingIN*) for that job. Data staging is a process of preloading job input data to the node that is scheduled to execute the job. Initiation denotes that data should be moved, but transfer of data is performed only when number of concurrent transfers in progress is below given treshold and when the data for the earlier starting jobs are already transferred. All of the data staging is performed asynchronously so the scheduling component can continue its execution.

Output data for completed jobs are also retrieved in the background when similar conditions are met (*initiateDataStagingOUT*). These are then available for the developer provided result handler.

In the end of its cycle the scheduler component checks if there is any finished data staging (*existsFinishedStaging*). For every finished data staging the appropriate computing component is notified (*Notify*), and the job execution can take place.

When all the jobs are scheduled, the scheduler component waits for all jobs to complete. While waiting, it takes care of all data staging in progress.

3.2. Computing component algorithm

Computing component (Fig. 3) is designed to check for availability of new jobs and to queue jobs for the computing utility. Multiple computing utilities are allowed to be attached to the computing component and to consume jobs from the same queue.

The computing utility gets jobs from the queue, executes these jobs, updates internal statistics and forwards the updated information to the scheduling component. This information is later used by the scheduling component to determine the availability and performance of every computing component.

Communication calls implemented in the computing component and computing utility are exclusively locked and forced to be performed only one at a time. Locking in the computing component is performed in the *waitForJob* procedure that is designed not to block on MPI calls. This is necessary due to threading issues in MPI implementations, but also adds up to the processing requirements for the algorithm.

4. Experimental results

In order to create a job pattern that includes heavy computation and high data throughput we decided to use data compression as the main test scenario. As the input for the system a number of files are provided, and the task is to perform extensive data compression on the files. The provided files are suitable for compression and achieve high compression ratios. The size of each file is 1MB. An appropriate job generator

```
Computing component{
  while(true){
    job=waitForJob();
    enqueueToCompute(job,preload_data);
  }
}

Computing Utility()
{
  while(true){
    job=dequeueJob();
    execute(job);
    lock(communication)
    sendStatisticsToSchedulingComponent();
    unlock(communication)
  }
}
```

Figure 3. Computing component algorithm

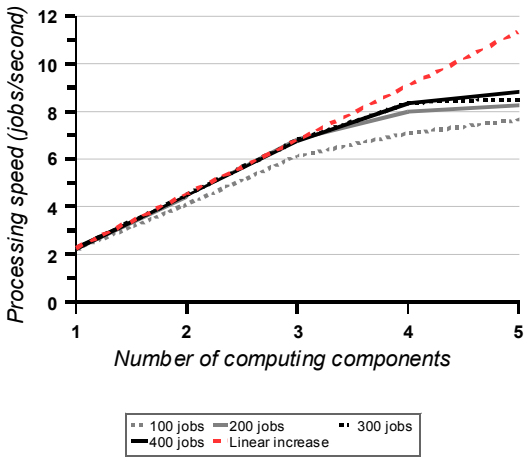

Figure 4. Experimental results

for the specified job model is provided.

The measurements are performed on 5 workstations that are connected using 100Mbps LAN. Each workstation is a 2800Mhz Pentium 4 machine, equipped with 1GB RAM and a 10krpm SCSI hard drive.

The results obtained from 20 experiments are illustrated in Fig. 4. Job sets of different size are varied together with several cluster sizes in order to determine performance of the scheduling system.

It can be observed that the increase in size of the job set is followed by the increase in the processing speed of the entire cluster. This is because the initial setup time of the application significantly affects performance results on small job sets. The initial setup time includes the time necessary for all the components in distributed environment to bootstrap together with the data staging time that is spent before the first compression takes place.

The performance increase is linear in scenarios with two or three workstations employed in data compression. The addition of fourth and fifth workstation results in performance gain that asymptotes to the value of 9 jobs/second. This is equivalent to 9MB/s (1MB per job) outgoing network traffic on the workstation hosting the scheduling component. Since the network is theoretically limited to 100Mbps (12MB/s) the performance of the distributed system is bounded by the network interface limitation of the scheduling workstation. In case of network upgrade we expect the next bottleneck to arise in hard drive performance on the same workstation, which

could be accompanied by the limited processing power.

A workaround for this problem would be to distribute scheduling component and the job generators to several workstations which would increase outgoing network performance. This would require all data (uncompressed files) to be available for all schedulers, which is a separate optimization goal. Schedulers should be modified in order to synchronize among themselves to prevent duplication of jobs in the system.

5. Conclusion

In this paper we have defined extendable architecture that can be used to develop parallel applications. Several interfaces are defined that can be used by developers to provide custom job generators, result handlers and to define custom scheduling policies.

Straightforward algorithm for job scheduling that is targeted for high throughput applications is given to demonstrate the architecture implementation. We focused our attention to network throttling by timing data staging process in order to preload data for the jobs before the compute node becomes available.

An experiment that includes independent job set with high throughput data transfer to the computing components was completed. The results indicate that the system bottleneck is in the only component in the system that is not distributed, i.e. the scheduling workstation.

The next logical step in extending the proposed architecture is to distribute scheduling. We also plan to extend this framework to achieve continuous data flow from the job generator through the compute nodes to the result handler targeting near real-time data processing.

6. References

[1] Chavarría-Miranda DG, Mellor-Crummey JM. An Evaluation of Data-Parallel Compiler Support for Line-Sweep Applications. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques; 2002 Sep 22-25; Charlottesville, Virginia, USA; p. 7-17

[2] Folding@home – Main http://folding.stanford.edu

[3] Garlan D, Shaw M. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall; 1996.

[4] Grid Engine http://gridengine.sunsource.net/

[5] Grudenic I, Bogunovic N. Modeling and Verification of MPI Based Distributed Software. LNCS 2006; 4192; 123-132

[6] Krammer B, Müller MS, Resch MM. Runtime Checking of MPI Applications with MARMOT. In: Joubert GR, Nagel WE, Peters FJ, Plata O, Tirado P, Zapata E, editors. Proceedings of the International Conference on Parallel Computing; 2005 Sep 13-16; Malaga, Spain. John von Neumann Institute for Computing, Jülich; 2006. p. 893-900

[7] Luecke G, Chen H, Coyle J, Hoekstra J, Kraeva M, Zou Y. MPI-CHECK: a tool for checking Fortran 90 MPI programs. Concurrency and Computation: Practice and Experience 2003; 15(2) : 93-100.

[8] Moab Cluster Software Suite http://www.clusterresources.com/pages/products/moab-cluster-suite.php

[9] Pllana S., Fahringer T. Performance Prophet: A Performance Modeling and Prediction Tool for Parallel and Distributed Programs. Proceedings of the International Conference on Parallel Processing; 2005 Jun; Oslo, Norway. Performance Evaluation of Networks for Parallel, Cluster and Grid Computing Systems; 2005 June 14-17; Oslo, Norway: IEEE Computer Society; 2005. p. 509-516.

[10] SETI@home: Search for Extraterrestrial Intelligence at Home http://setiathome.berkeley.edu

[11] Siegel FS. Verifying Parallel Programs with MPI-Spin. LNCS 2007; 4757; 13-14

[12] The Open Source Distributed Render Queue (DrQueue) http://drqueue.org/ [04/25/2008]

[13] The TotalView Debugger (TVD) http://www.totalviewtech.com/index.htm [04/25/2008]