Hybrid CPU and GPGPU Volunteer Computing Framework over the Extensible Messaging and Presence Protocol for Parallel Branch and Bound Optimization of Truss Structures

Mikolaj Mackowiak
Krzysztof Kurowski

Poznan Supercomputing and Networking Center, ul. Noskowskiego 10, 61-704 Poznan, Poland

Aleksandr Igumenov
Julius Zilinskas

Vilnius University, Institute of Mathematics and Informatics, Akademijos 4, LT-08663 Vilnius, Lithuania

Abstract

A new fault-tolerant message passing interface framework for a hybrid parallel computing in distributed volunteer-based networks is presented in this paper. It is a new programming core library being able to link together many CPUs and promising General Purpose Graphic Processing Unit (GPGPU) devices over the Internet. We successfully developed a high-level programming interface on the top of well-known XMPP communication protocol to exchange and synchronize controlling messages among computing processes executed remotely via the Internet on distributed computing clusters. In this short paper, we briefly present the architecture of our fault-tolerant parallel programming framework called XMPP-MPI, together with preliminary results obtained. We conclude this paper with best practices and improvements that will be needed to solve computational problems like presented parallel branch and bound algorithm for topology optimization of truss structures using larger amount of volunteer computing resources.

Key words: Computational grid, GPGPU, Distributed computing, Global optimization, Volunteer computing, Branch and bound

Email addresses: mikolaj.mackowiak@man.poznan.pl (Mikolaj Mackowiak), krzysztof.kurowski@man.poznan.pl (Krzysztof Kurowski), igumenov@gmail.com

Preprint submitted to Elsevier

June 21, 2011
Introduction

Today, linking together many personal computers distributed over the Internet involves highly efficient many-core CPUs together with powerful general purpose graphic processing units. New computing facilities can be used by end users to speed up their legacy parallel applications. Unfortunately, well known parallel programming and execution environments, e.g. Message Passing Interface, have been developed and then optimized for systems typically deployed on homogeneous supercomputers and computing clusters, separated from the Internet by firewalls.

There is a large number of parallel algorithms and legacy applications, where parallel processes have to exchange messages and must be synchronized in an MPI like manner. Moreover, managing parallel processes onto heterogeneous computing resources via the network requires new protocols that support an easier and secure way to establish reliable communication links among personal computers. Even the commonly used parallel programming environments, such as OpenMPI [1], are practically impossible to setup due to firewalls often blocking of outgoing and incoming network traffic in operating systems on PCs. Consequently, parallel programming environments are difficult, or often impossible to deploy using volunteer computers available in the Internet or any type of network. Therefore, one of the main motivations for our research was to create a standard based parallel environment where communication messages can be easily distributed not only in the master-worker model, but also synchronously and asynchronously among parallel processes. Consequently, we are able to engage ordinary desktop computers to build an alternative to existing solutions which gives developers more programming features and flexibility to implement parallel applications and their execution on distributed computers.

Related Work

In fact, the concept engaging end-users personal computers over the Internet is not new. Often called as volunteer computing, it has gained the momentum many years ago when SETI@Home [2] system engaged idle CPUs to help scientists in processing large amount of data received from radio telescopes.

The Berkeley Open Infrastructure for Network Computing [3], which evolved from the SETI project, successfully reached recently the level of Petascale computing using ordinary desktops to process data [4]. The BOINC approach fits

(Alksandr Igumenov), julius.zilinskas@mii.vu.lt (Julius Zilinskas).
perfectly to many computing models which can be split into many independent computing pieces. Those parts of data are typically spread out over the Internet and then synchronized by a central application server. The end-user has to download a BOINC application and register to a certain project via the Internet. His program obtains input data, then processes them while computer is idle, and uploads to the central server once it is finished. The usage of Internet connections is limited to the basic server-client interactions, without a possibility to exchange or synchronize controlling messages among personal computers involved in the computational experiment.

OpenMPI [1] is a free implementation of standardized API for passing messages in parallel and/or distributed computing. It is used in powerful supercomputers all over the world [5]. OpenMPI utilizes standard network connections to communicate with participating nodes, therefore due safety reasons clusters operates in local area networks and are accessible from the Internet via dedicated machines only. This solution is highly effective and depending on connection type data can be passed with speed varying from 100Mbps to 100Gbps or even higher. On the other hand, cluster solutions require large input in hardware infrastructure and have high exploration costs.

**XMPP-MPI Framework**

In this section we briefly describe a new parallel programming environment together with a set of well-defined communication APIs and core library - XMPP-MPI developed on the top of the Extensible Messaging and Presence Protocol (XMPP) [6, 7]. The XMPP protocol, also called Jabber, is a well-known instant messaging protocol optimized for nearly real-time and extensible instant messaging over the Internet. The key XMPP feature is the HTTP binding for users, so they personal computer behind restricted firewalls can easily communicate by fetching and posting messages without any disruption. Moreover, there are many free access Jabber servers available in the Internet.

Thanks to decentralized and federated XMPP architecture any massage can be forwarded to a desired destination via a chain of XMPP servers, what in turn naturally improves the reliability and protocol scalability. Technically speaking, the XMPP-MPI client is executed in a demon mode on a user personal machine. Once the machine is connected to the Internet, the client has to establish an XMPP-based connection to the nearest XMPP server available. It is worth to emphasize at this point that the XMPP-MPI core library was implemented in ANSI C what allows end users to executed on different operating systems, including Windows, Linux or Mac OS. More importantly, the XMPP-MPI core library was tested with a new parallel computing architecture developed by NVIDIA called Compute Unified Device Architecture [8].
### Table 1
The comparison of the basic MPI vs. XMPP-MPI communication and synchronization functions

<table>
<thead>
<tr>
<th>OpenMPI function</th>
<th>XMPP-MPI function</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>xm_send</td>
<td></td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>xm_receive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xm_connect</td>
<td>connect client to jabber server</td>
</tr>
<tr>
<td></td>
<td>xm_disconnect</td>
<td>disconnect client from jabber server</td>
</tr>
<tr>
<td>MPI_Comm_create</td>
<td>xm_group_create</td>
<td></td>
</tr>
<tr>
<td>MPI_Comm_join</td>
<td>xm_group_join</td>
<td></td>
</tr>
<tr>
<td>MPI_Comm_disconnect</td>
<td>xm_group_destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xm_group_leave</td>
<td>allows user to leave comm. group</td>
</tr>
<tr>
<td>MPI_spawn</td>
<td>-</td>
<td>cannot spawn processes in XMPP-MPI</td>
</tr>
</tbody>
</table>

Therefore, the XMPP-MPI based application can be executed locally on both CPU and GPU devices. The XMPP-MPI server also establishes a connection to his XMPP server. His unique jabber address called JabberID or shortly JID is known a’priori to participating nodes. This application awaits for client nodes to join a created computing group. After joining group, client acquires from server an ID number, unique within computing group, which is used in message passing functions. In general the XMPP-MPI API is similar to OpenMPI, however due to different concept it does not support MPI standards. The comparsion between OpenMPI and XMPP-MPI functions can be found in table 1.

One of the core feature of XMPP is presence notification. We distinguish several types of presence status, e.g. available, free for chat, away, etc., however XMPP-MPI connects with away status only. After a client establishes the connection to his XMPP server, in order to send messages, it must send initial presence to server. Next, the client’s server broadcast received message to all resources that are subscribed to receive user’s presence notifications. When client disconnects from server, on his behalf server sends unavailable presence notification to all client subscribers. [9] This way, XMPP-MPI clients are almost immediately notified of nodes unavailability and can update their node list dynamically.
Most of the XMPP servers are open-source projects. However, XMPP standards were presented in 2004, some of them still have errors. In order to deal with presence malfunctions, a fallback mechanism was implemented. It bases on a ping-pong mechanism. If client doesn’t respond on several requests, group manager assumes that client is unavailable and removes him from his group by sending notification to remaining clients.

As mentioned before, XMPP-MPI based programs need to communicate with server to receive presence notifications. It is managed by a separate background thread called XMMonitor. By creating this structure, another message type could be distinguished. The XMPP-MPI sends message in three ways:

- **XMSendSync** - sending client awaits for recipient’s notification when message is received using `xm_receive` function
- **XMSendAsync** - an asynchronous send. Client send message and does not wait for any response
- **XMSendNormal** - the XMPP-MPI specific send type. Sending client receives confirmation from recipient’s XMMonitor thread. The message is delivered, but sender has no information, that the recipient had read his message.

Thanks to this three types, programmer can send messages in more efficient way. In cluster environments, where messages are passed with very high speed, synchronous processing is well seen. Volunteer networks characterizes asynchronous and message driven processing. Moreover, the connection between participating nodes are significantly slower, therefore message size must be limited.

In conclusion XMPP-MPI works best with long computing and small input data problems like branch and bound algorithms. By partitioning given problem into large amount of small pieces, the volunteer based network can be used. Even though some of nodes can become unavailable during computing, their packages can be computed by remaining nodes. Thanks to XMPP-MPI ability to send messages between nodes directly, using scalable, distributed XMPP servers’ network the bottleneck of one central synchronizing server is avoided.

**Preliminary test and results**

We established a real testbed connecting two computing clusters located in completely different geographical locations, in Poznan (Poland) and Vilnius (Lithuania) respectively as it is shown in Figure 1. Personal computers available in those locations are connected internally over two completely separated networks. Moreover, those two networks were completely separated due to the
deployment of Network Address Translation (NAT) technique. In a nutshell, NAT is a practical and common solution to hide many private network IP addresses behind a single IP address. Thus, even large networks can be connected to the Internet with a limited number of public IP addresses, often just one IP address. However, even the NAT-based isolation of personal computers allowed our clients located in two local networks to establish the communication channel to other nodes via a chain of two XMPP servers. XMPP servers were playing a role of proxy managing the communication among all clients independently from the underlying network complexity. In our tests we repeated many experiments to measure and estimate both the scalability and performance of our solution.

Parallel branch and bound algorithm for optimization of truss structures [10] was chosen as computing problem for usage in XMPP-MPI framework. The number of XMPP-MPI workers varied between 6 and 14, while in Open-MPI was constant. In presented results number of XMPP-MPI workers had been averaged. Computers used in tests had similar configuration, thus this same computing power.

The results of performed computational experiments show that the new XMPP-MPI library performs similar to MPI library, see 2. Moreover, we observed that adding more XMPP servers does not impact significantly on the overall application performance. Therefore, using our approach we can establish dynamically large-scale parallel computing environments, more scalable than the BOINC type of applications, see Figure 2.
<table>
<thead>
<tr>
<th></th>
<th>OpenMPI</th>
<th>XMPP-MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workers</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>task parts</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Time [s]</td>
<td>375</td>
<td>146</td>
</tr>
</tbody>
</table>

Table 2
Execution times of optimization of truss structures comparsion between OpenMPI and XMPP-MPI

Figure 2. The generic comparison of BOINC and XMPP-MPI architectures

Conclusions and further directions

Preliminary tests encouraged us for further extensions and optimization of our framework using more volunteer personal computers in the Internet with many-core CPUs and GPUs. As it was presented in the previous section, we identified some relevant bottlenecks that have to be taken into account to improve our parallel solving environment. Our XMPP-MPI core library is generic and not limited to the mentioned branch and bound problem. We expect to release more parallel application templates, e.g. using a higher level and object-oriented parallel environments, e.g. Charm++[11], to simplified the development of new applications. Consequently, even today we are able to offer both developers and end-users a more scalable alternative to a centralized BOINC type approach to take advantage of hundreds or thousands personal computers located in campus networks or in the Internet.
Acknowledgments

The work was partially supported by the COST Action “Open European Network for High Performance Computing on Complex Environments” IC0805.

References


