J Supercomput (2012) 59:548–567
DOI 10.1007/s11227-010-0452-9

A platform independent distributed IPC mechanism in support of programming heterogeneous distributed systems

Mohsen Sharifi · Ehsan Mousavi Khaneghah · Morteza Kashyian · Seyedeh Leili Mirtaheri

Published online: 25 June 2010

Abstract Interprocess communication (IPC) is a well-known technique commonly used by programs running on homogeneous distributed systems. However, it cannot be used readily and efficiently by programs running on heterogeneous distributed systems. This is because it must be given a uniform interface either by a set of middleware or more efficiently properly ported to the kernel of all varieties of open source and closed source proprietary operating systems running on heterogeneous nodes of distributed systems. This is particularly problematic to achieve when the kernel code of closed source operating systems are inaccessible to third parties. We propose an alternative nonproprietary approach to enable the use of IPC in heterogeneous distributed systems by wrapping IPC calls from the kernel of closed source operating systems, and converting them into equivalent IPC calls that are efficiently implemented inside the kernel code of open source operating systems. To show the superiority of our approach, we developed a wrapper for converting MS-Windows IPC calls into equivalent Linux IPC calls and benched our approach on a hybrid computer cluster running both types of operating systems.

Keywords Hybrid cluster · Wrapper · Open source · Closed source · Interprocess communication (IPC) · Kernel level · User level · Message · Remote procedure call

M. Sharifi · E. Mousavi Khaneghah · M. Kashyian · S.L. Mirtaheri (✉)
School of Computer Engineering, Iran University of Science & Technology, Tehran, Iran
e-mail: mirtaheri@comp.iust.ac.ir

M. Sharifi
e-mail: msharifi@iust.ac.ir

E. Mousavi Khaneghah
e-mail: emousavi@comp.iust.ac.ir

M. Kashyian
e-mail: kashyian@comp.iust.ac.ir

1 Introduction

Computer clusters are known as a type of distributed system comprising of homogeneous computers networked in a restricted area. This has not stopped researchers though to offer varieties of clusters for different requirements such as for high availability (HA) and high performance computing (HPC) on heterogeneous platforms networked in wider areas [1]. Some researchers have further studied and proposed hybrid clusters comprising of as many heterogeneous clusters networked in a restricted or wider areas, and used different techniques to enable communication between the disparate and heterogeneous members of a cluster or members of different clusters. A well-known technique used is the distributed interprocess communication (IPC) technique that has been implemented at different levels, namely at the user level as a set of library routines, at the operating system interface level as a transparent middleware, or at the operating system kernel level transparent to users and applications.

User level implementation of distributed IPC is the least efficient and easiest to do, though hardest to be used by programmers. Kernel level implementation is the most efficient and easiest to be used by programmers but it has the hardest implementation. Middleware level implementations are somewhat in between these two implementation levels considering their efficiencies and their ease of use [2, 3].

Kernel level implementations of distributed IPC become challenging particularly when heterogeneous platforms run under a combination of open source and closed source operating systems. Though the implementation of a distributed IPC mechanism within the kernel of an open source operating system is in itself very difficult and challenging, but a similar implementation inside the kernel of a closed source operating system may be impractical by many. So one may raise the question of why bother at all to do such an implementation inside the kernel of a closed source operating system in the first place. The answer is that most clusters use commercial off the shelf (COTS) operating systems whose closed source code run on homogeneous platforms and have no problem as far as they run independently. Problem arises when several clusters whose underlying platforms and operating systems are heterogeneous try to provide a single system image to users and applications running on such hybrid clusters.

Two solutions to the problem of implementing a distributed IPC mechanism on heterogeneous hybrid clusters can be envisaged. The first solution is that the vendors of closed source operating systems agree on a common protocol for communicating processes on heterogeneous platforms and (closed and/or open source) operating systems. An alternative solution is proposed in this paper that does not need such an agreement and can thus be provided by any third party, though it may sacrifice some efficiency and implicate lower efficiency compared to the first solution.

We propose a platform independent distributed IPC mechanism in support of programming heterogeneous distributed systems in general and specially hybrid heterogeneous clusters. This is achieved by wrapping IPC calls from the kernel of closed source operating systems and then converting them into equivalent IPC calls that are efficiently implemented inside the kernel code of open source operating systems. To be more specific, we have developed a wrapper for converting IPC calls in Microsoft Windows to equivalent IPC calls in Linux and benched our proposed mechanism on

a hybrid computer cluster comprising of a Windows cluster and a Linux cluster based on DIPC 2006 [2]. The wrapper provides two, one for each operating system that enables developers to develop programs on the hybrid cluster using our proposed wrapper programming model.

2 Motivation and background

Computer clusters have long since been introduced and deployed to solve scientific problems with reasonable cost effective performance. The need for affordable higher performance has been risen ever since too with the requirements and wishes of more scientists to develop and solve more complex models of their concerned problems. Avionics is an exemplar field whose scientists have been active in extending their developed avionics models with more and more affective measurable parameters, requiring higher performance. There is thus no end to the requirement for growing higher performance of computer clusters in the foreseeable future.

On the other hand, HPC clusters are mostly comprised of commercially off the shelf (COTS) computers [4] that run under commercially off the shelf network operating systems, ranging from closed source ones like MS-Windows and Unix to more commonly deployed open source ones like Linux. MS-Windows based clusters are more recent and fewer than Unix and Linux based clusters, on which most of HPC applications have been mapped. In fact, most HPC applications have long been developed according to the Unix and Linux programming model in scientific programming languages like Fortran. Linux based clusters have become the de facto clusters now in scientific fields due to the open source nature of the Linux operating system.

Given the need for higher performance clusters, would it be possible to cluster different types of HPC cluster computers running MS-Windows and Linux? Though logically feasible, this is not readily available. This is because of differences in the programming models of MS-Windows and Linux, especially in their communication models, that makes HPC applications developed under these two models not interoperable. The availability of so many MS-Windows based commercially off the shelf computers makes the provision of a reasonable solution to interoperability of MS-Windows and Linux clusters very tempting for providing even higher expected performances. This is exactly what has initiated our current research reported in this paper.

One of the challenges of developing an HPC cluster out of ordinary computers is the implementation level of the system software for the establishment and the management of the cluster. Library level implementation levels such as in MPI yield a more stable standard for HPC application developers, but lead to lower performance compared to implementation levels closer to the kernel of the cluster underlying network operating systems such as in the DIPC2006 solution. The latter approach relieves HPC application developers from many intricacies of cluster programming. Furthermore, the very mechanisms that are embedded in the kernel of the operating system in support of cluster computing are also available to all network programs running on that operating system. This is in fact one of the main strengths of this level of implementation that enables the interoperability of network and cluster programs with comparatively less changes to the source code of programs. That is why

the proposed wrapper mechanism is enabled to map the base RPC mechanism of MS-Windows to equivalent Linux embedded DIPC2006 system V mechanisms, making MS-Windows and Linux clusters interoperable.

It is important to note that we have been faced with two operating systems in the development of our proposed wrapper mechanism that have been designed only in support of network programs and no specific support for cluster computing. We neither wanted to modify these operating systems nor wanted to impose changes to HPC applications already developed base on differing communication mechanisms of these two operating systems. Instead, the proposed wrapper mechanism tries to enable the transparent communication of Linux clusters in search of acquiring higher performance using MS-Windows clusters with these clusters, without changing Linux cluster programs or the MS-Windows RPC communication mechanism.

The wrapper approach can be thought of as an Inter HPC Cluster Resource Discovery mechanism wherein a Linux cluster as a header cluster machine may utilize as many Linux or MS-Windows clusters at their disposal to run Linux cluster application programs with higher cumulative performance without entangling these programs with how IPC is handled by MS-Windows clusters. We have successfully used the proposed wrapper approach in distributed discovery of resources in the integrated peer to peer distributed system framework whose principal features have been presented in [5]. Using this mechanism, all four main types of resources (processes, input/output, memory, and file) are discovered using Unix System V interprocess communication mechanisms even when resources are owned by MS-Windows peers.

The rest of paper is organized as follows. Section 2 presents some notable related works. Section 3 describes our solution by presenting the architecture of the wrapper and how the primitives in Linux and MS-Windows operating systems are mapped. Section 4 compares the wrapper mechanism with other distributed mechanisms using the time taken to finish remote primitive calls as the benchmark criteria. Section 5 presents a comparative study of the proposed wrapper with respect to DIPC2006 and highlights the strengths of the wrapper solution. Section 6 concludes the paper.

3 Related work

Wrapper is a software pattern to leverage the capabilities of a component, an object or even a library without involving the internal complexities of the used component [6]. We present and use a wrapper in this paper to implement, the compatibility between IPC mechanisms in two different operating systems on the one hand and, the simplicity of the programming model for developers on the other hand.

Wrapper can be implemented at different levels, namely the kernel, the application or the user level. A kernel-level implementation needs modification to kernel that is not our concern here because we cannot change the kernel of closed source operating systems. An application-level implementation depends on the types and requirements of specific applications leading to a solution that will not be reusable. A user-level implementation of wrapper, which is adopted by us in this paper, provides the transparency and at the same time can be tailored to any known or unpredictable require-

ments. Having said the chosen level of implementation of wrapper, let us continue with a brief review of related works.

Condor [7], IPC layer [8], and LAM [9] are amongst the works that are implemented at the user level, without modifying the operating system kernels. This is why this category of distributed IPC mechanisms can be used with commercially available closed source operating systems. These systems translate system calls between the application and the kernel and there is a limitation on porting if there are no equivalent system calls in the two communicating operating systems.

Condor is a particular type of heterogeneous distributed environment that supports high throughput computing. Performance is not the main objective of Condor; it tries to perform long lasting computations by assigning each job to an idle system. It uses a process migration mechanism that is implemented outside the kernel creating low transparency and performance but achieving high level of portability [10]. Condor is not capable to create new processes if multi-programming is needed. It does not support inter-process communication either. Transparency is virtualized by shadow processes, one for each process. System calls are performed by Condor on behalf of remote systems [11, 12].

IPC layer has the required mappings to provide a uniform interface to application processes. Therefore, each different operating system in a hybrid cluster needs a different interface implementation. The IPC layer communicates with the process on one hand and the underlying interprocess communication mechanisms of the system on the other hand. Because the IPC layer interposes between the application layer and the system layer, system performance is low.

LAM is capable of changing a dedicated computing cluster to a large parallel computer for solving computationally extensive problems. It uses an MPI (Message Passing Interface) based message passing mechanism (Burger, 2006). It is in fact an MPI based programming environment for heterogeneous networked systems. LAM provides a single daemon as a nano-kernel that is hand-threaded for each system; daemons are linked together by UDP [9]. MPI based programs comprise of a set of autonomous processes running their own code in MIMD or SIMD styles. Processes are executed in their own address spaces but can share memory. Communications between processes are point to point and are established by MPI-API. Although MPI is known as a standard for inter-process communication on distributed memory systems but it does not support platform interoperability and the number of processes is fixed implying that no process is created or removed [13]. Windows Compute Cluster Server uses MS-MPI (i.e., Microsoft's implementation of MPI) in support of communication between nodes [14].

Although Condor, IPC layer, and LAM provide the means for interprocess communication in their perceived heterogeneous distributed environments, they put restrictions on facilities like multiprogramming and dynamic process creation. They are not concerned with high performance either, which is in contrast to our concern for high performance in our proposed wrapper approach.

The second category includes solutions whose distributed IPC mechanisms are implemented at the socket level. These implementations include TIPC [15] and Striding IPC based on TCP/IP [16]. This category uses sockets popularized by Berkeley Software Distribution (BSD). They use the communication capabilities of their

underlying protocols. Therefore, they provide a protocol-independent interface such that an application that is using an implemented form of socket, just like Winsock, can communicate with other applications using different socket implementations on other systems. This is very helpful in implementing communication between heterogeneous operation systems. However, different byte sequences of different operating systems should be converted into a uniform format.

TIPC (Transparent Inter-Process Communication) protocol allows developers to design distributed applications capable of communicating with each other regardless of their locations on a clustered network. Various versions of TIPC for different operating systems like Linux®, Solaris, VxWorks®, and MS-Windows® are available. Applications that are written in C/C++ language and use TIPC can communicate with each other by a set of addresses defined in AF_TIPC through sockets [15]. In addition to inter-process communication, it enables kernel-to-process and kernel-to-kernel communication. The TIPC protocol is interposed between the application layer and the transport layer like Ethernet, TCP, or ATM. It focuses on transparency rather than performance [17].

Striding platforms of inter-process communication with TCP/IP protocol is an experimental solution for power supply control in particular locations [13]. The implemented system follows two objectives: (1) using QNX6.20 to provide real-time features, and (2) to provide better user interface to WIN2000 for setting control parameters. WIN2000 is a center for transferring and gathering data to or from control devices. Communication between two operating systems is based on the socket (TCP/IP) mechanism. Because of differences in operating systems, variable-host byte sequences have to be converted into uniform-network byte sequences for data and control transfer. Although the system implements a central server and multiple clients, it can be adjusted to become a hybrid cluster. It is designed to transfer data toward process rather than requesting remote processing on a remote data [16].

The solutions in the above second category have portability problems like byte sequence conversion for different operating systems and tight bounding to socket addresses to provide location transparency. In contrast, our wrapper approach tries to remove such like restrictions by using a remote procedure call mechanism for process communication and a socket mechanism for communication between processes.

There are also some component-based implementations of distributed IPC mechanism that provide a higher level of abstraction compared to other types of implementations. An example of this third category is [18].

Microsoft has developed a communication foundation called WCF (Windows Communication Foundation) for establishing secure and reliable transacted Web services with the objective of reducing the complexity of applications by unifying Enterprise Services, Messaging, .NET Remoting, Web Services, and WSE. It uses and combines the features and abilities of Microsoft .NET Remoting, Web Services, Distributed Systems, and Web Services Enhancements [19].

Microsoft has also DCOM for distributed programming. DCOM is an extension to COM that provides distributed features using the Object Remote Procedure Call (ORPC) mechanism. However, it does not support multiplatform requirements and only works well on Microsoft platforms. In contrast, multiplatform distributed components like XPCOM do not have this DCOM limitation and it is supported by Windows NT, MacOS, and Linux. The COM limitation still exists wherein the interface of

one component should be passed to another one on a remote server upon connection [18].

The above third category of distributed IPC mechanisms have lower communication complexities and are thus simpler to use because the run-time environment of components is responsible for handling the communications. However, legacy applications must be rewritten from scratch to use the features of components and component-based communication mechanisms. This is quite contrary to the objective of our wrapper approach that requires the least or even no modification to applications.

There is another type of distributed IPC mechanism implementation for wider area networks like World Wide Web that is based on standard document request format [8]. This type of implementation uses an abstraction level at the URL level that makes it suitable for communication between Web servers. An applet is a sample of code migration in a distributed environment that uses the Internet features like Web servers and the HTTP request format to obtain its required documents. The instructions of an applet are executed within an applet viewer in isolation; disallowing the applet to communicate with other servers except the server that applet comes from. This security policy limits the functionality of applets in distributed environments. This restriction can be obviated by encoding RPC calls in a standard document request format like URL by using HTTP, and sending the URL to an RPC process. The RPC process can then parse the URL to find the requested services. Finally, the results can be sent back to the applet by placing them in a document [8]. Additional handshaking is needed to send the services to an applet by RPC process before forming the requests by the applet. As a result, the establishment of an IPC in loosely-coupled heterogeneous distributed environments is simple for developers but has low system performance because of extra efforts needed to prepare the communication based on the HTTP protocol. Our proposed wrapper approach uses the same mechanism but at lower levels of operating system architecture as much as possible to achieve higher performance.

4 The proposed wrapper

In this paper, we propose a wrapper approach to solve the problem of communications between processes that run on heterogeneous platforms and operating systems. We have implemented our wrapper for a cluster running under two operating systems, namely, MS-Windows that is a closed-source operating system and Linux that is an open-source operating system. The wrapper provides a library for both operating systems and enables program developers to design programs using our proposed wrapper programming model. The Linux members of cluster form a kernel-level high performance cluster [20]. The wrapper somehow ports processes on these different operating systems as required by detecting the interprocess communication between processes, and processes need not be rewritten for communicating with processes running on a different operating system.

It is almost for the first time that such a wrapper library is included in a high performance computing (HPC) hybrid cluster whose members run on both open-source

and closed-source operating systems. The distributed IPC mechanism is implemented at the kernel level of Linux using DIPC2006 [20] that provides a high performance cluster, while the wrapper gives a simple to use cluster wherein processes running on MS-Windows can transparently communicate with processes running under Linux without knowing the differences in the communication protocols.

4.1 Wrapper structure

Our proposed wrapper has two modules, namely, the MS-Windows Wrapper Manager (WWP) and the Linux Wrapper Manager (LWP). We refer to both of them as Wrapper Manager (WM) unless it is stated otherwise explicitly. Figures 1 and 2 show the block diagrams of WM modules for MS-Windows and Linux, respectively. WM consists of three major modules, *Converter*, *Transferring*, and *Executer*. In addition, it has an *Identification Manager* that is responsible for tracking the address of the server to receive the clients' requests. The Converter module takes an MS-Windows RPC-based source and converts it to an IPC System V message-based one or vice versa (we refer to RPC and message for MS-Windows RPC and IPC System V message, respectively, unless it is stated otherwise explicitly). The Transfer module receives the converted program and sends it to the Executer module for running. Communication between the two WMs is done by the Transferring module.

Suppose a distributed program is implemented on a MS-Windows based network and uses the RPC mechanism for communication between the programs, namely clients and servers. In normal conditions, a client sends its requests to a server. In this scenario, we need some criteria to trigger the wrapper module in abnormal conditions and redirect client's requests to the same server but on another system. The module that detects the criteria is called the *Performance Detector* (PD). If PD detects any deficiency that is completely stated in the configuration of PD, it triggers the WM module, which is described shortly afterward.

WM tries to port the server program into Linux by establishing a communication between two WMs on two operating systems. The communication is based on socket mechanism and it is defined as a function for transferring a module. On the other part, the destination WM on the destination operating system, i.e., LWM, receives the converted server program. Because the distributed program on the Linux-based network has been written using the message mechanism, WWM must convert the program so that it can run on the Linux system based on the message mechanism. LWM loads and runs the message-based daemon, produces results, and sends back the results to the requesting system. Results are taken from the message daemon by WWM on behalf of the RPC client. Finally, WWM converts the results to RPC-based understandable data for the RPC client.

4.2 Wrapper Dynamics

In this section, we describe how a request is serviced by an RPC client on MS-Windows step by step as it is shown in Figs. 1 and 2. Figure 3 shows the whole scenario of our wrapper at a glance. As it was mentioned earlier, the Performance Detector (in Fig. 3) watches the status of the system based on some defined criteria,

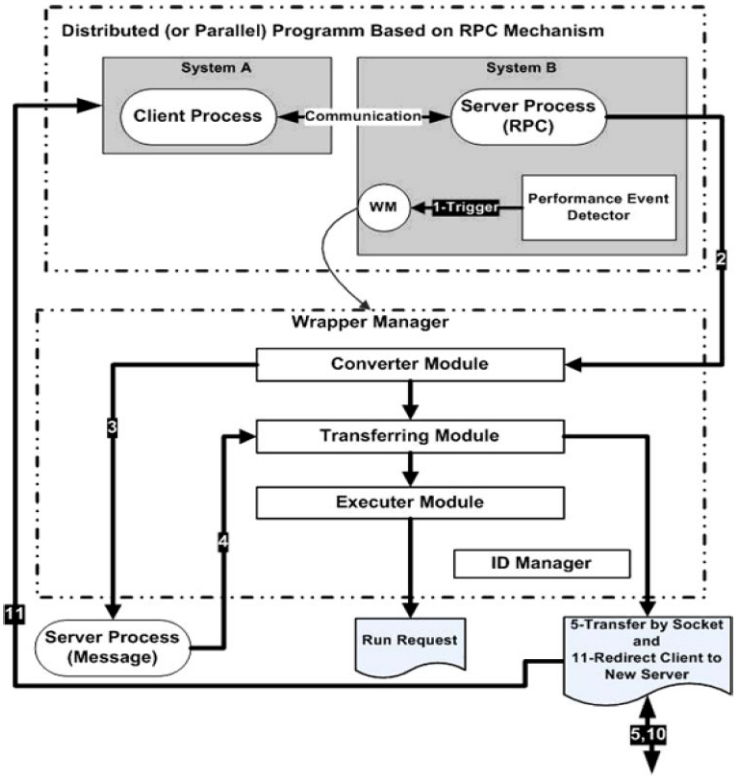

Fig. 1 The MS-Windows wrapper manager module

e.g., by measuring the CPU load and the available memory space, and enforces the cluster load balancing rules based on these criteria. If conditions do not meet, it triggers an event in MS-Windows (step 1 in Fig. 1) or sets a flag in Linux to inform the server (or daemon) program to initialize an appropriate WM, else regular communication between programs proceeds (Figs. 2 and 3). WM then reads the flag or catches the event through the listener and retrieves the required information to choose the server process for porting; it also estimates the amount of required resources to run the program on the destination system.

In the second step (Fig. 1), the Converter module reads the server process and converts it to a message-based program using the conversion algorithm (presented in Sect. 4.3). Step 3 (Fig. 1) shows the converted server source from RPC to message. In the next step, step 4 in Fig. 1, the Transferring module takes the converted program and establishes a communication link to one of the Linux-based systems on the network using a socket mechanism if such a link has not been established before. The Transferring module takes the ID of the destination system from its ID manager and stores it on the local ID manager. The destination ID on the local ID manager is used by the client program (request initiator) for communication to identify results. In step 5, the Transferring module sets the new destination address of the server pro-

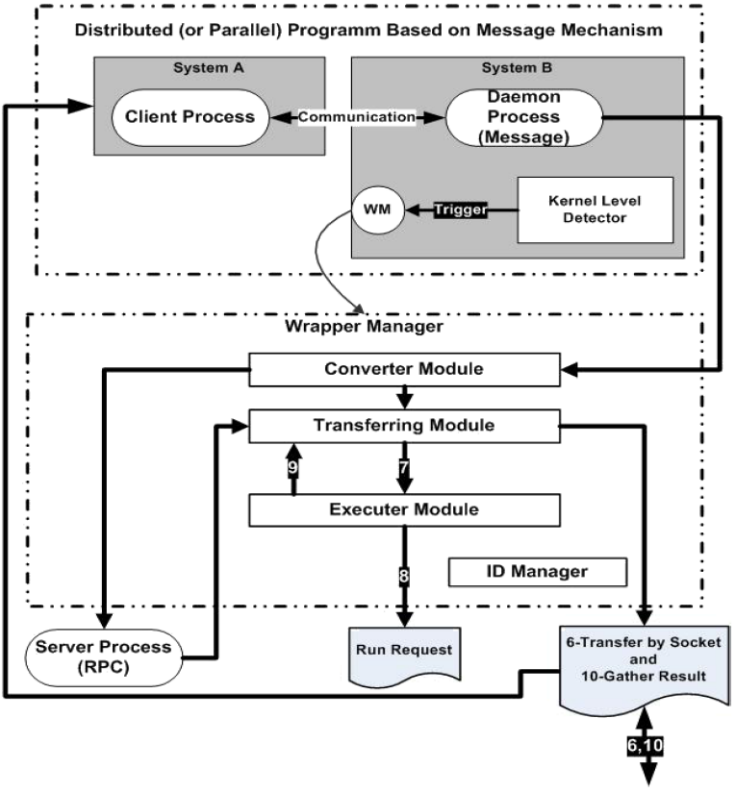

Fig. 2 The Linux wrapper manager module

gram in the naming service address database so that the client program can continue to send requests.

The Transferring module on the destination Linux-based system receives the message version of the ported program (step 6 in Fig. 2) and in step 7 sends it to the Executer module. The Executer module runs the daemon and gets the results from it and sends the results back to the Transferring module (steps 8 and 9 in Fig. 2). The results are sent to the WWM Transferring module (step 10 in Fig. 1). Finally, the Transferring module in WWM converts the results to the RPC-based data module and sends it to the RPC client (step 11 in Fig. 1). The reverse flow starting from the Linux platform and initiated by a client's request, is the same as the flow from MS-Windows to Linux with a difference in the conversion of the daemon program from message-base program to RPC-based program and conversion of the retrieved results from the WWM on MS-Windows platform from RPC to a message understandable to the client (Fig. 2).

4.3 Primitive mapping

Clusters that use open source operating systems like Linux often provide some message-passing primitives at the kernel level so that application developers can pro-

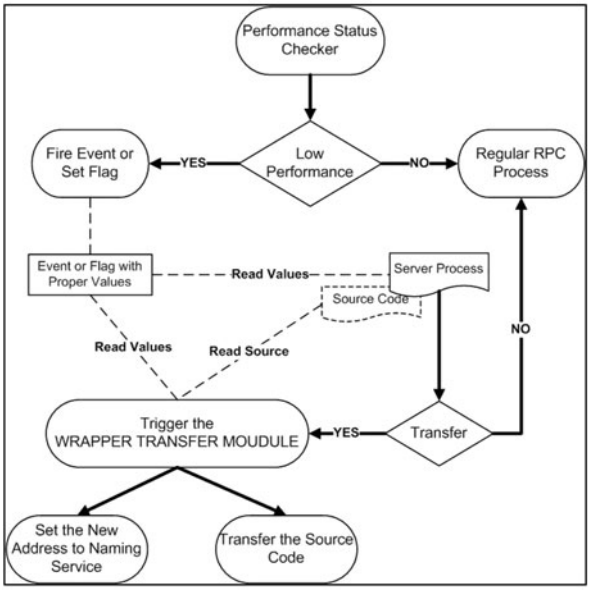

Fig. 3 The wrapper manager algorithm

gram their communication needs by system calls to these primitives manually. The necessary details about communication for distributed programming are thus made available [21] and developers can implement their required high performance clusters [22].

Clusters that use closed-source operating systems like MS-Windows are different. Communication primitives are provided at the operating system or middleware level as APIs and developers do system calls using these APIs. Since we consider MS-Windows for developing a wrapper as a closed source operating system, we choose the RPC mechanism to develop distributed programs because the RPC mechanism itself is the building block of the MS-Windows internal communication structure and MS-Windows supports it very well. Although there are some limitations in this RPC programming model, like having interface description for both remote procedure call and procedure itself, and the restriction on the data types of arguments, developers are not involved in the details and complexities that the message system entails in the RPC programming model. In spite of these constraints, some distributed programming features, like RPC programming system and RPC run-time system, are provided by programming language. This is why the RPC programming model has become a good candidate for distributed programming on closed source clusters.

To ease communication in a heterogeneous environment by remote inter process communication using header wrapper files, we need to convert the communication primitives of the above two different programming models, namely RPC on the MS-Windows platform and message on the Linux platform, to each other. A portable inter process communication is thus needed that is provided by the Wrapper modules. The Wrapper library defines instructions in its headers that allow the WM converter to map a required system call to a corresponding one on another system [23]. Two cases arise in this mapping. System calls that have one equivalent are converted easily with

Table 1 RPC functions in MS-Windows operating system

RpcBindingFromStringBinding	Returns a binding handle from a string representation of a binding handle
RpcNsBindingImportBegin	Creates an import context for importing client-compatible binding handles
RpcNsBindingImportNext	Looks up an and returns a binding handle
RpcStringBindingCompose	Creates a string binding handle
RpcStringFree	Frees a character string allocated by the RPC run-time library

Table 2 Message system calls in Linux operating system

msgsnd	Delivers a message to a queue
msgctl	Performs control operations on a message queue
msgrcv	Receives a message from a queue

Table 3 Mapping between RPC functions in MS-Windows and message system calls in Linux

MS-Windows	Linux
RpcBindingFromStringBinding	msgget
RpcNsBindingImportBegin	
RpcNsBindingImportNext	
RpcStringBindingCompose	
	msgsnd
	msgrcv
RpcStringFree	msgctl

some considerations for preparing arguments and taking back the returned results. Butsystem calls that either have more than one equivalent or do not have any equivalent, just like combination of system calls or deciding on a suitable equivalent, need more complicated mapping [24]. Tables 1, 2, and 3 show sample system calls of both systems and their mappings.

As Table 3 shows, the receive (msgrcv) and send (msgsnd) primitives in the message mechanism have no explicit equivalents in the RPC mechanism. They are performed by the RPC run-time system. In the RPC mechanism, client calls a remote procedure regardless of its location and developer is not directly involved in data send and receive. So, we need to wrap the send and receive system calls in the message mechanism and simulate them in the RPC mechanism.

Conversion from message to RPC and vice versa is not always done completely. In our experiments on 100 chosen program codes, conversion from message to RPC was more successful (85%) than from RPC to message (72%). The mismatch was mostly due to Win-RPC based program code that used thread facilities. This was in turn due to the mismatch between different levels of scheduling of threads in the two operating systems; threads are scheduled at the kernel level in Ms-Windows operating system but at the user level in Linux using thread libraries like Linux Threads, NGPT, and NPTL [24].

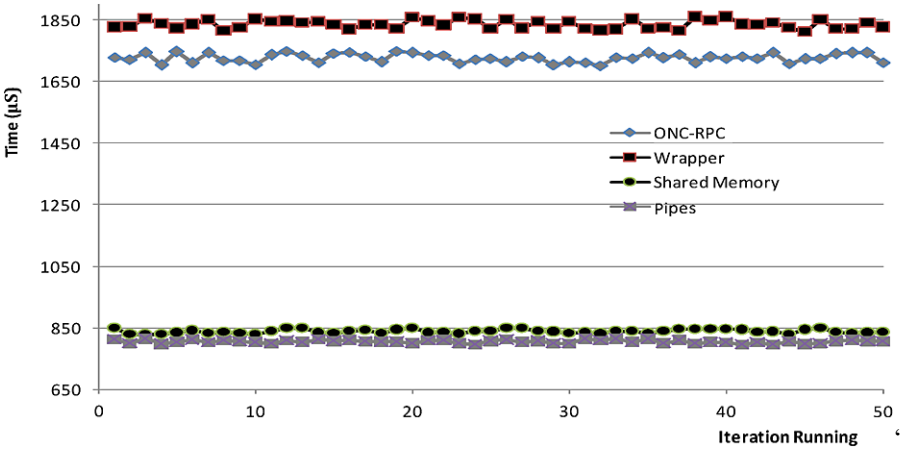

Fig. 4 Average times taken for creation and destroying of Wrapper modules

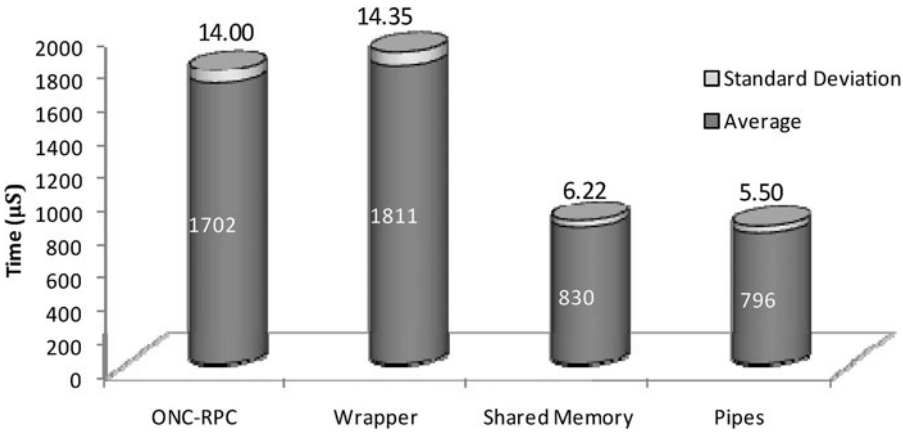

Fig. 5 Average times taken for creation, initializing and destroying of Communication modules

5 Evaluation

To provide an evaluation of our proposed wrapper mechanism with respect to other types of notable IPC mechanisms, we measured the time taken to completion of a remote call/request under these mechanisms. We divided these mechanisms into two groups, namely those that support heterogeneous environments (e.g., ONC-RPC and wrapper) and those that support only homogeneous environments (e.g., pipes and shared memory). Experiments were done in two steps.

In the first step, we measured the times taken for creation, initialization and destroying of communication modules (Figs. 4 and 5). Communication modules consisted of all modules that were needed to start data transfer.

It should be mentioned that the creation, initialization, and destroying of the communication modules occurred once in the life cycle of the wrapper. Two distributed

programs were developed, one running under Linux (representing a homogeneous platform), and one running under a combination of Linux and MS-Windows platforms. Both programs were executed 50 times with each IPC mechanism and the values were measured based on the average and Standard Deviation of time spent to finish a remote call/request. Creation, initialization, and destroying of communication modules took about 0.5 to 2 milliseconds in different mechanisms. As Fig. 5 shows, the average time taken for creation, initialization and destroying of communication modules for ONC-RPC and Wrapper mechanisms were close. The large difference between the times taken under each group (homogeneous and heterogeneous) represents the different characteristics of the two groups. However, the low value of standard deviation shows the stability of the mechanisms.

In the second step, we measured the times taken for marshalling and unmarshalling operations in addition to data transfer time; 50 KB for small size arguments and 1 MB for medium size arguments were considered; big size arguments needed more complicated data transfer models that we considered out of scope of the current paper. Figures 6 and 7 show the average times in case of small size and medium size arguments, respectively. Data transfer between clients and servers were through parameter passing. Data was organized as integer arrays and sent to the server side and an integer value was returned by the server to the client side as the return value.

The times taken for marshaling and unmarshalling under all mechanisms in case of small size data were nearly the same and equal to 5000 μs, but the times taken for data transfer were quite different and equal to 4100–4500 μs for ONC-RPC and wrapper mechanisms, and 1300–1500 μs for message and pipe mechanisms. This large difference in data transfer times (about 3000 μs) in the two groups is because the data transfer in the wrapper and ONC-RPC mechanisms is performed in a distributed environment requiring more time to completion. A similar pattern was recorded for 1 MB medium size data (Fig. 7), wherein the marshalling and unmarshalling under all mechanisms took nearly 100 ms, but data transfer times were 70 ms for mechanisms in the heterogeneous group and 17 ms for mechanisms in the homogeneous group.

6 Discussion

In this section, we present the limitations of current MS-Windows-based cluster solutions in utilizing facilities of Linux-based cluster solutions specially in supporting dynamic processes by focusing on inability of DIPC2006 as a Linux cluster solution in serving and communicating with MS-Windows-based cluster solutions, and discuss the importance of wrapper solution in establishing this communication.

As mentioned before, DIPC2006 enables developers to use existing instructions to establish communication between processes through local IPC and write programs that can run on distributed systems. It does this by extending the local IPC mechanism for distributed environment. In other words, it is a mechanism that provides IPC based distributed programming facilities for developers. Wrapper and DIPC2006 can be viewed similar in provision of distributed programming facilities on clusters, though they are different in their functionalities. DIPC2006 is designed for distributed programming on Linux clusters that are comprised of member machines powered by

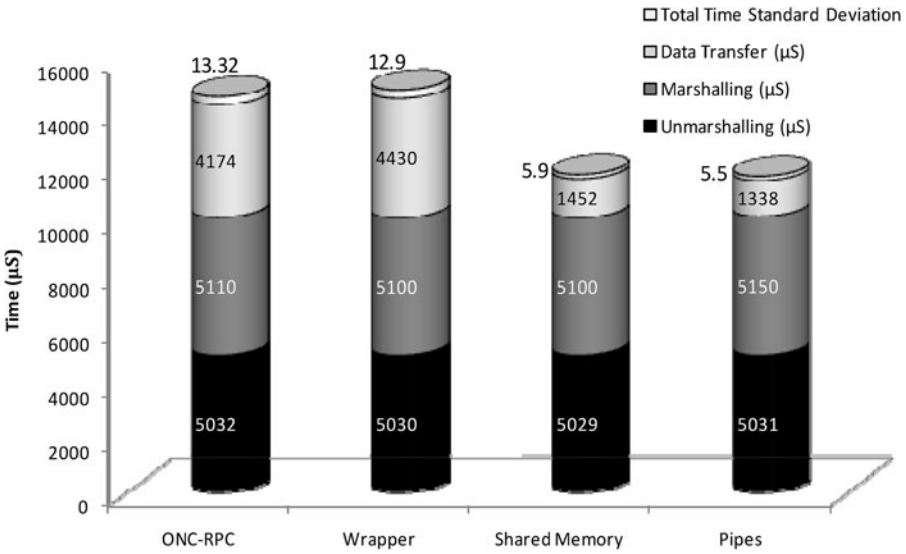

Fig. 6 Average times taken for marshalling, transfer and unmarshalling of small size data (50 KB)

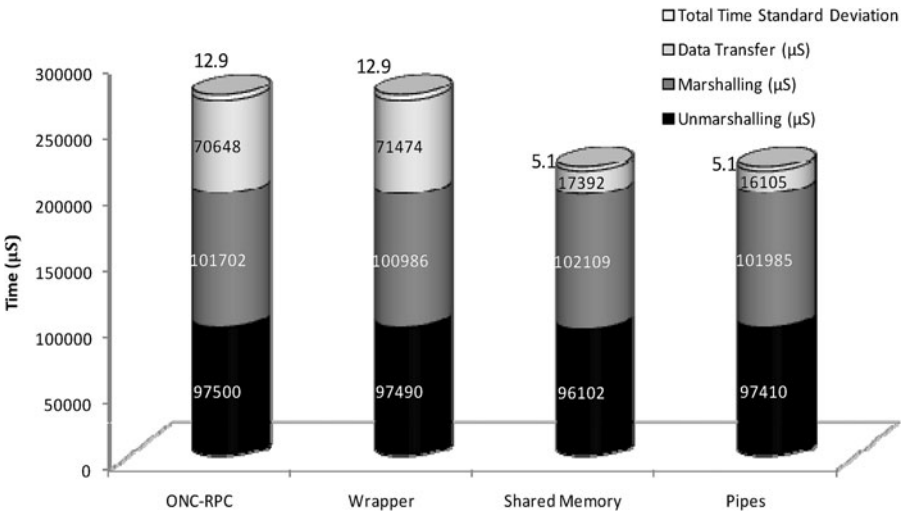

Fig. 7 Average times taken for marshalling, transfer and unmarshalling of medium size data (1 MB)

Linux. If a cluster wants to use the DIPC2006 features, it has to use Linux with DIPC2006 installed in the kernel of its operating system.

DIPC2006uses extended IPC programming instructions to create cluster-based programs on a Linux cluster, while wrapper uses RPC programming instructions to create cluster based programs running on a hybrid cluster wherein some member machines run under Linux and some others run under MS-Windows. The main difference between DIPC2006 and wrapper is in this respect. Developers who use

DIPC2006 can run their programs only on DIPC2006 based homogeneous Linux clusters. In contrast, developers who use the wrapper mechanism can run their programs on both homogeneous clusters (running under either MS-Windows or Linux) and heterogeneous clusters (running under a mixture of MS-Windows or Linux). We can thus conclude that the wrapper mechanism can be used to attain a more scalable cluster that can deploy more resources at its disposal while the DIPC2006 does not allow such deployment even in case such resources are available only because they do not run under Linux.

DIPC2006-based clusters are closer to the rather traditional definition of closed-world clusters for two reasons. Firstly, a DIPC2006-based cluster consists of machines that collaborate to run a dedicated designed DIPC2006 based program. Secondly, the DIPC2006 software is installed on all member machines to enable the cluster to run the program that is written by using the DIPC2006 programming language. These conditions force developers to narrow their required resources for developing a DIPC2006 based program to available and existing resources in the cluster without regard for any disposable external resources. In other words, if A and B are processes which is written by the DIPC2006 programming language and it is supposed to run them on the DIPC2006 cluster, then we will have:

$$\text{Resources of (Process A and Process B)} \in \text{Resource Set of Cluster}_{\text{DIPC}} \quad (1)$$

A and B processes in (1) are only allowed to use resources in the designed cluster which is targeted by a DIPC2006-based program. Clusters like this one that are designed to run only written programs which use DIPC2006 programming language are called a Closed Cluster. These are closed because they have to follow two tight conditions to be able to run a DIPC2006-based program. Firstly, cluster members need to use the Linux as a local manager. Secondly, the DIPC2006 software should be installed at the kernel level of the Linux.

Generally speaking, any cluster programming language should have two attributes to become a favorable language for developers of distributed programs: (1) provide efficient facilities for establishing communication between processes in the cluster and (2) have effective resource usage model. In our experiments, DIPC2006-based programs exhibited an acceptable performance implying that DIPC2006 programming language has an acceptable performance. This was due to the specific implementation of DIPC2006 programming language and fewer number of required instruction translations to run instructions of a program. But it fell behind of wrapper with respect to effective use of resources external to the current cluster but disposable to it. This feature is particularly needed for computationally intensive scientific applications running on clusters. The proposition of Cluster Runtime Library (CRL) that tries to provide runtime resources required by scientific and engineering programs is an evidence for such a need [25].

The above discussion becomes more critical when it is combined with the dynamic process concept. Let us provide an example. Consider a scientific or engineering program that consists of only two processes A and B. Developer encodes the general requirements for hardware and software resources in support of A and B on an assumed cluster at design time. Upon creation of a third process called C at run-time

by either A or B, we have:

$$\exists \text{Resource} \in \text{Resourceset(ProcessC)} | \text{Resource} \ni \text{Resourceset(Prosecc A and B)} \quad (2)$$

Equation (2) is a necessary condition and it means there is a resource member of resources set of process C as its not member of resource set A and B. If this condition holds, the cluster manager should provide the required resources for C inside the cluster; if it cannot provide them, the whole program will stop executing. Providing resources for C and possibility of responding to this process are a function of cluster size and cluster type, which is:

$$\text{F(Response Possibility)} \propto (\text{Cluster}_{\text{Size}})\text{R}(\text{Type}_{\text{Cluster}}) \quad (3)$$

R: Relation; $\propto$: Iscommensuratewith

Solving (3) in general case is very difficult or even impossible in some cases, but one can solve it for special cases. The Wrapper software is a inductive solution for (3) in a special case. In the wrapper solution, the type of cluster is MS-Windows cluster. Given the higher number of existing programs developed for Linux clusters, the number of Linux clusters themselves and the number of Linux cluster members in comparison with the number of MS-Windows clusters and the number of MS-Windows-based cluster programs [26], resource density in MS-Windows clusters is less than Linux clusters.

We can thus expect that Linux clusters are more amenable to deploy extra software resources for processes using the vast number of existing programs and projects on Linux clusters and also extra hardware resources by allowing extra machines to be added to the list of cluster members. This is to say that the wrapper software enables developers of MS-Windows-based programs for MS-Windows clusters to use features and resources of Linux clusters, and to create dynamic scientific and engineering programs without exclusively resorting to traditional programming concepts of the MS-Windows operating system like RPC mechanism.

7 Conclusion

Limitations of homogeneous distributed systems especially clusters in providing resources to fulfill almost unpredictable resource requirements had lead researchers to provide communication means between two or more homogeneous clusters or heterogeneous clusters. Undoubtedly, hybrid heterogeneous clusters have more flexible facilities to implement distributed programs. But the complexity and difficulty of implementing such communication means depended on their level of implementation. Kernel-level implementations had shown higher performance compared to user-level implementations but sacrificed ease of programming that is most favored by users and supported by user-level implementations.

In this paper, we focused on the problem of providing the means of communication between those types of heterogeneous clusters whose communication means

are implemented at the kernel level, but some belong to open source operating systems and others belong to commercial closed source operating systems. We chose a high performance Linux cluster called DIPC2006, representing an open source cluster, alongside an MS-Windows based cluster, representing a closed source cluster, whose communication means had been implemented through IPC at the kernel level of their operating systems. The challenge was to compose these two heterogeneous clusters to build a hybrid cluster whose element clusters could communicate with each other, given our inaccessibility to MS-Windows' closed kernel source. We presented a wrapper solution to this challenge.

Since IPC is the communication building block in almost all operating systems, we selected IPC as a suitable communication mechanism in our wrapper solution. To achieve closed source heterogeneity in hybrid clusters, we implemented the Wrapper as a platform independent distributed IPC mechanism in support of programming heterogeneous distributed systems.

The IPC mechanisms we used for distributed programming were RPC for developing programs on MS-Windows and message mechanism for Linux-based programs. In our proposed approach, IPC calls from MS-Windows are wrapped and converted into IPC calls for Linux using a conversion table to map the IPC calls of the two mechanisms. Because DIPC2006 had been used on Linux members of the cluster, IPC calls were efficiently implemented inside the kernel code of Linux and hence, the cluster had high performance feature.

To present a comparison of the Wrapper with other notable IPC mechanisms, especially those that support heterogeneous environments, we experimentally measured the completion time of a remote call/request using each of these mechanisms. The RPC mechanisms in our experiment included both heterogeneous (like ONC-RPC) and homogeneous (like pipes) mechanisms. Two types of distributed programs, one for a Linux cluster using homogeneous mechanisms in communication and one for a hybrid cluster using heterogeneous one, were developed and tested in 50 runs. We calculated the average time and standard deviation of marshalling, unmarshalling, and data transfer activities for small and medium sized arguments.

The results showed that there was a large difference in completion time values between two groups of mechanisms and that was predictable because of the difference of characterizes of the two groups. A noticeable result was the closeness of the measured values in the Wrapper and ONC-RPC. Completion times of marshaling and unmarshalling in two groups of mechanisms were nearly the same, but different for data transfer. The large difference in data transfer times in the two groups was because the data was transferred in a distributed environment using the wrapper and ONC-RPC mechanisms, requiring more time to convert the primitives in two different operating systems to complete a request.

Our proposed Wrapper solution is not restricted to the mentioned IPC mechanisms we noted in this paper, and it can be extended to other types of mechanisms. Also, if programming environments use different mechanisms simultaneously, the Wrapper solution can be extended to have some detection and decision modules to switch between them dynamically. The Wrapper solution can be used in other process migration systems too in order to port sections of an application that uses IPC for its communication between its different internal processes.

References

1. Sterling T (2001) Beowulf cluster computing with Linux, 1 edn. MIT Press, Cambridge
2. Tannenbaum AS Steen MV (2002) Distributed systems: principles and paradigms. Prentice Hall, New York. US edn
3. Mirtaheri SL, Mousavi Khaneghah E, Sharifi M (2008) A case for kernel-level implementation of inter-process communication mechanisms. In: The 3rd IEEE international conference on information and communication technologies: from theory to applications (ICTTA'08), Damascus, Syria, 7–11 April 2008
4. Sterling T, Becker D, Savarese D, Dorband JE, Ranawake UA, Packer CV (1995) BEOWULF: a parallel workstation for scientific computation. In: Proceedings of the international conference on parallel processing (ICPP), August 1995
5. Sharifi M, Mirtaheri SL, Mousavi Khaneghah E (2009) A dynamic framework for integrated management of all types of resources in P2P systems. J Supercomput 52:149–170
6. Hart JM (2004) Windows system programming, 3 edn. Addison-Wesley, Reading
7. Thain D, Tannenbaum T, Livny M (2005) Distributed computing in practice: the condor experience. Concurr Comput Pract Exp 17(2–4):323–356
8. Dean DF (2007) Interprocess communication mechanism for heterogeneous computer processes. United State Patent No: US 6,874,151, Mar. 29, 2007
9. Gloger W (2007) LAM/MPI user guide. Pervasive Technology Laboratory at Indiana University, Version 7.1.4., 2007
10. Gropp W, Lusk E, Sterling T (2003) Beowulf cluster computing with Linux, 2nd edn. MIT Press, Cambridge. Chap. 15
11. Basney J, Livny M (1999) Deploying a high throughput computing cluster. In: High performance cluster computing, Rajkumar Buyya, vol. 1. Prentice Hall, New York. Chap. 5
12. Milojicic DS, Douglis F, Paindaveine Y, Wheeler R, Zhou S (2000) Process migration. Technical report. HP Labs, AT&T Labs-Research, TOG Research Institute, EMC, University of Toronto, Platform Computing
13. Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS (2004) Open MPI: goals, concept, and design of a next generation MPI implementation. In: Euro PVM/MPI 2004, Budapest, Hungary, September, 2004
14. Lantz E (2008) Using Microsoft message passing interface (MS MPI). Windows HPC Server 2008, Microsoft Corporation
15. TIPC programmers guide: version 1.2.2. Available: http://tipc.sourceforge.net/. Accessed 2010
16. Yi L, Haichun C, Jing Q, Zhuomin C (2004) Striding network of inter-process communication based on TCP/IP protocol. Plasma Sci Technol 6(4). doi:10.1088/1009-063016141008
17. Maloy J, Stephens A (2010) Specification for version 2 of the TIPC protocol. Multicore Association. Accessed: http://tipc.sourceforge.net/
18. Orvill F, Therning M (2000) Evaluation of Interprocess communication methods in a component based environment. Master thesis, Linkoping University
19. McMurtry C, Mercuri M, Watling N (2006) Microsoft windows communication foundation, 1st edn. Sams, Indianapolis
20. Mousavi Khanegah E, Mirtaheri SL, Sharifi M (2008) Evaluating the effect of inter process communication efficiency on high performance distributed scientific computing. In: The 2008 IEEE international conference on embedded and ubiquitous computing (EUC 2008), sponsored by Shanghai Jiao Tong University and Shanghai Computer Association, Shanghai, China, December 17–20, 2008
21. Kato K, Ohori A, Murakami T, Masuda T (1993) Distributed C language base on higher-order RPC technique. Academic Press, San Diego
22. Mirtaheri SL, Mousavi Kaneghah E, Sharifi M, Abdollahi Azgomi M (2008) The influence of efficient message passing mechanisms on high performance distributed scientific computing. In: The 2008 IEEE international symposium on advances in parallel and distributed computing techniques (APDCT2008), in conjunction with the 2008 IEEE international symposium on parallel and distributed processing with applications (ISPA2008), Sydney, Australia, December 10–12, 2008
23. Kashyian M, Mousavi Kaneghah E, Mirtaheri SL (2008) Portable inter process communication programming. In: Advanced engineering computing and applications in sciences (ADVCOMP '08), International Conference, Valencia, Spain, September 29–October 4, 2008. IEEE Computer Society, Los Alamitos, pp 181–186 ISBN:978-0-7695-3369-8

24. Muthuswamy SS, Varadarajan K (2005) Port Windows IPC apps to Linux. IBM, Armonk
25. Broquedis F, Furmento N, Goglin B, Namyst R, Wacrenier P (2009) Dynamic task and data placement over NUMA architectures: an OpenMP runtime perspective. In: Fifth international workshop on OpenMP IWOMP 2009: runtime environments. Lecture notes in computer science, vol 5568, pp 79—92
26. Top 500, 'Supercomputer site'. http://www.top500.org/. Accessed 2010