A Case for Kernel Level Implementation of Inter Process Communication Mechanisms

Seyedeh Leili Mirtaheri
Department of Computer Engineering
Iran University of Science & Technology
Narmak, Tehran
Iran
mirtaheri@comp.iust.ac.ir

Ehsan Mousavi Khaneghah
Department of Computer Engineering
Iran University of Science & Technology
Narmak, Tehran
Iran
emousavi@comp.iust.ac.ir

Mohsen Sharifi
Software Engineering Laboratory
Department of Computer Engineering
Iran University of Science & Technology
Narmak, Tehran
Iran
msharifi@iust.ac.ir

Abstract

Distributed systems must provide some kind of inter process communication (IPC) mechanisms to enable communication between local and especially geographically dispersed and physically distributed processes. These mechanisms may be implemented at different levels of distributed systems namely at application level, library level, operating system interface level, or kernel level. Upper level implementations are intuitively simpler to develop but are less efficient. This paper provides hard evidence on this intuition. It considers two renowned IPC mechanisms, one implemented at library level, called MPI, and the other implemented at kernel level, called DIPC. It shows that the time taken to calculate the Pi number by a distributed system that uses MPI to program and run the calculation of *Pi* number in parallel is on average 35% slower than by the same distributed system that uses DIPC to program and run the calculation of *Pi* number in parallel. It is concluded that if distributed systems are to become an appropriate platform for high performance scientific computing of all kinds, it is necessary to try harder and implement IPC mechanisms at kernel level, even ignoring so many other factors in favor of kernel level implementations like safety, privilege, reliability, and primitiveness.

Keywords: Distributed Systems, Operating System Kernel, Inter Process Communication (IPC), Distributed Inter Process Communication (DIPC), High Performance Scientific Computing.

1 Introduction

A distributed system is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone or complete computers cooperatively working together as a single, integrated computing resource. The general focus nowadays is mostly on MIMD model, using general purpose processors or multi-computers. We only focus on multi-computers in this paper. In contrast to

multiprocessors, multi-computers lack physical shared memory. They rather rely either on message passing or on distributed shared memory. Irrespective of which one of these interfaces they provide to the programmers, they have to implement some sort of inter process communication mechanism. The crucial challenge is how efficient these mechanisms are implemented by distributed systems. Ignoring special cases where efficiency is not an issue, like in educational or research systems, efficiency of IPC is critical to real distributed systems that are intended for high performance computing, like in scientific cluster computations dealing with compute-intensive computations. Given the importance of IPC mechanism implementation efficiency, the next challenge is how best one can attain the best degree of implementation efficiency.

Generally, IPC mechanisms in distributed systems can be implemented at 4 levels, namely at program level, library level, operating system interface level, and lastly at the lowest system level at the operating system kernel level. The latter level is prone to yield the best efficiency because it runs in the privileged kernel mode of the operating system and thus uses fewer instructions for implementation of IPC, while other levels run in non-privileged user mode and consequently use more instructions for the same purpose[13, 14].

Surely, kernel level implementation of IPC has to be more efficient than in other levels, but it is much harder to implement. That is why so many implementations at other levels are common in distributed systems, like the Message Passing Interface (MPI [1]) that is used extensively in scientific cluster computations.

In this paper we provide hard evidence that kernel level implementation of IPC is more efficient and much appreciated and needed in high performance computations like scientific cluster computations. To achieve this objective, we have chosen a library level implementation, namely MPI, and a kernel level implementation, namely DIPC [2] that has been developed by ourselves and previously reported [2,3,4]. The calculation of *Pi* number, as a standard benchmark program, has been programmed and ran using these two mechanisms on the same distributed platform to show the superiority of DIPC.

The rest of paper is organized as follows. Section 2 gives a brief introduction to MPI and DIPC. Section 3 compares the results of runs of *Pi* program on MPI and DIPC. Section 4 concludes the paper.

2 Backgrounds on MPI and DIPC

2.1 Message Passing Interface - MPI

The MPI standard defines a software library used to turn serial applications into parallel ones that can run on distributed memory systems. MPI has sought to make use of the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as the standard. MPI has been strongly influenced by works at the IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, and PARMACS. Other important contributions have come from Zipcode, Chimp, PVM, Chameleon, and PICL [5].

The MPI standardization effort involved about 60 people from 40 organizations

mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry. The standardization process began with the Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing. The basic MPI defines policy and codify some standards for communication problems but do not explain how to implement these standards. Some vendors such as LAM [6], ProMPI [7] and MPICH [8] have implemented these standards [1, 9].

Several advantages are attributable to MPI [1, 10]**:** universality, expressivity, well suited to formulating parallel algorithms, acceptable performance (explicit association of data with process allows good use of cache), any parallel algorithm can be expressed in terms of the MPI paradigm, runs on both distributed and shared-memory systems (performance is generally good in either environment), allows explicit control over communication leading to high efficiency due to overlapping communication and computation, allows for static task handling, data placement problems are rarely observed, for suitable problems it scales well to very large numbers of processors, it is portable, and its current implementations are efficient and optimized.

A number of critical disadvantages have been reported too [10]: it is harder to learn than shared memory programming, it does not allow incremental parallelization, its variate implementations cannot communicate with each other, application development is difficult (re-fitting existing serial code using MPI is often a major undertaking, requiring extensive restructuring of the serial code), it is less useful with fine-grained problems where communication costs may dominate, for all-to-all type operations, the effective number of point-to-point interactions increases as the square of the number of processors resulting in rapidly increasing communication costs, dynamic load balancing is difficult to implement, and last but not lastly, variations exist in different manufacturer's implementation of the entire MPI library, where some may not implement all the calls, while others offer extensions

MPI is implemented in Library Level and provides some functions to users for parallel programming; these functions include system calls that are executed by the operating system kernel. The first MPI versions only supported parallel programming but later ones after the MPIV2 added facilities for distributed programming.

MPI does not fully support heterogeneous platforms. Most of its implementations are Linux based on i386 hardware. Some of them support windows and MAC too but only on i386 [1, 9, 11].

It is noteworthy that most of the aforementioned weaknesses of MPI are due to being implemented at the non-kernel level.

2.2 Distributed Inter-Process Communication - DIPC

DIPC [2, 3, 4] was founded by Dr. Mohsen Sharifi in 1993. DIPC provides the programmers of the Linux operating system with distributed programming facilities, including Distributed Shared Memory

(DSM). It works by making UNIX System V IPC mechanisms (shared memory, message queues and semaphores) network transparent, thus integrating neatly with the rest of the system. The underlying network protocol used is TCP/IP and it is targeted to work on WANs (Wide Area Networks) and in heterogeneous environments.

UNIX is among the platforms of choice for writing parallel and distributed programs. The AT&T UNIX provides what is known as System V IPC mechanisms, consisting of shared memories, message queues and semaphore sets, to enable programmers to exchange data and synchronize between processes running on the same computer. It should be noted that DIPC provides a set of mechanisms, and is not concerned with policies. The software designer determines how these mechanisms are used.

DIPC is based on UnixWare operating systems and support some hardware platforms such as I386, PowerPC, Motorola, and Sparc. DIPC's services are accessible via the Linux kernel, letting application programmers to use the already familiar System V IPC system calls to send and receive data. So, as far as the application programmer is concerned, there are no major changes in DIPC's programming model relative to normal System V IPC programming. There is also no need for any modified compilers or link libraries.

DIPC strengths lie in simplicity of the system (preferring simplicity of the algorithms whenever a conflict between that and the performance arises), transparency of the distributed facilities (doing distributed actions is not very different from doing the same actions in a single computer), independence from network characteristics (the programmer is not concerned with physical characteristics of the computer network, such as network topology, addresses, etc.), compatibility with legacy software (non-distributed programs using System V IPC mechanisms are able to coexist with other distributed programs, simplicity of programming (preserving the UNIX semantics helps those programmers who are already familiar with UNIX, and prevents the need to master some completely new programming models), independence from any specific programming tool or model (programmers are able to use DIPC in any language that can access operating system's functions), ability to turn legacy programs into distributed ones (it is relatively easy to change older programs, using System V IPC mechanisms and run it on multi-processors, to take advantage of DIPC, thus making them distributed programs), ability of the programmer to influence program performance (the main performance parameters such as frequency and amount of data exchange between machines is in the hands of the programmer), ability to develop programs on inexpensive hardware (programs could be developed on a single computer and later used in a computer cluster), making DIPC work on Wide Area Networks, making DIPC work in a heterogeneous environment.

It is noteworthy that most of the above good features of DIPC are due to being implemented at the kernel level.

A number of weaknesses of DIPC were also reported in 1996 [2, 3, 4] such as lacking any formal specification, suffering from SVIPC restrictions like the number of Message, Size of DSM, etc., lack of Fault Tolerance support, sole reliance on operating system for security, supporting only strict consistency model of DSM, and lack of support for process migration. Fortunately

none of these weaknesses affect our argument in this paper, except the strict consistency model that goes against DIPC in competition with MPI.

3 DIPC and MPI Implementation Levels Compared

Let us reiterate that the overall performance of applications running on distributed systems depend on the efficiency of IPC mechanisms that are implemented by the distributed systems. This is particularly true in applications or systems that need or provide high performance computing. The reason is that computations are ideally broken down to many parallel processes by programmers to be run efficiently in a distributed fashion by the distributed systems. These processes often need to communicate to perform and complete their computations and thus their efficiency depends very much on the degree of efficiency of the provided IPC mechanisms.

The implementation of every IPC mechanism, regardless of the level of its implementation, entails one or more system calls to the operating system. Each system call traps the kernel and yields a process context switch between user mode and kernel mode. The switch is costly and has lots of overhead, so it has to be avoided as far as possible if one is interested in better system *response time*.

In DIPC, UNIX System V IPC mechanisms [12], consisting of semaphores, messages and shared memories, are smartly modified to function in a network environment. The very same system calls that are used to provide local communication between processes running in the same computer are allowed to be used for communication between remote processes running on different machines. There is no new system call for the application programmers' use and programmer only use IPC system calls like *Msg-Send* , *Msg-Recv*, etc. That is to say, the use of IPC entails a single system call irrespective of whether it is used for local or remote communication between processes.

In contrast, the MPI primitives for IPC (like *MPI_Send*, and *MPI_Recv*) that are used by programmers are implemented as a set of system calls, each of which traps the kernel. For example, 7 system calls are made to the kernel when an *MPI_Send* is called by an MPI application. This entails 7 context switches, compared to 1 system call and context switch when a DIPC *Msg-Send* is called by a DIPC application. Considering a whole application program that uses so many IPC calls, one can be optimistic to get higher performance from DIPC than MPI.

To back up the above optimism, we conducted experiments with DIPC 2.1 and LAM/MPI 7.1. We deployed a homogeneous computer cluster consisting of 4 dual Intel cores with 512MB RAM each. Each client node was equipped with at least one network interface card, and clients were connected to each other in a WAN.

We chose the calculation of the *Pi* number as the benchmark. Since *Pi* is an irrational number, it cannot be written as a simple fraction or as an exact decimal with a finite number of decimal places. However, one can increase the number of digits until it reaches a number as near to *Pi* as needed. Mathematicians with computers have calculated *Pi* to millions of decimal places. *Pi* is used in several mathematical calculations like in calculating the area of circles, and the volume of spheres or cones.

We programmed the *Pi* calculation in C, once with DIPC and once with MPI. In actual fact, we chose an existing MPI program for *Pi* calculation that was reported to have good performance on MPI, but programmed the calculation in DIPC by ourselves.

Figure 1 shows the *Wall Time* of executing the two programs 20 times under the same conditions on the test bed cluster; wall time is the sum of user and system time. Figure 2 shows the average wall time of both program runs, implicating a near 35% better performance in favor of DIPC.

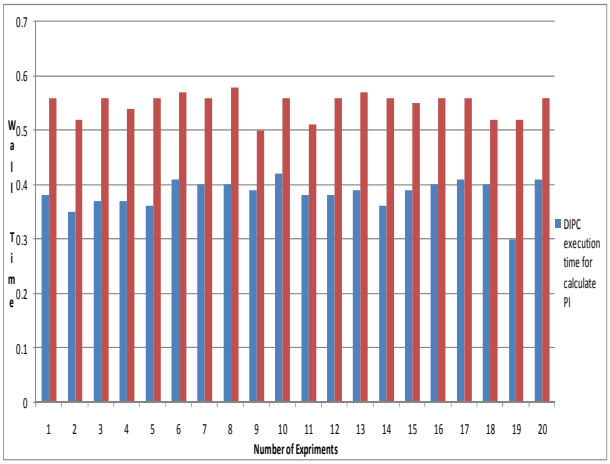

Figure 1. Wall times of *Pi* calculation using DIPC and LAM/MPI

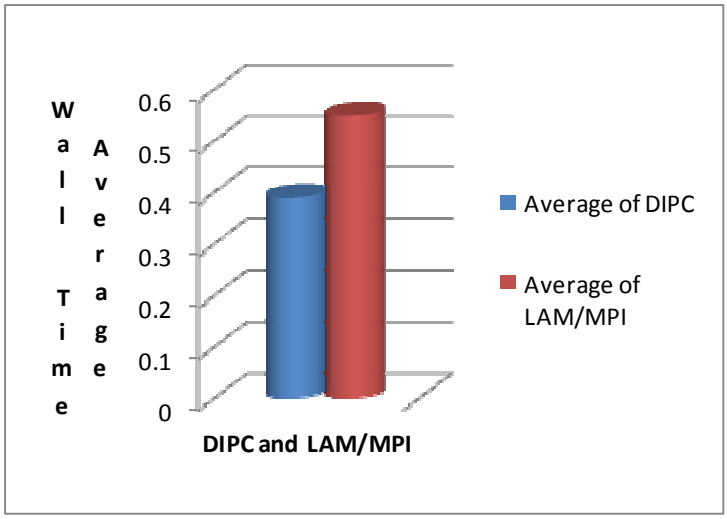

Figure 2. Average wall times of *Pi* calculation using DIPC and LAM/MPI

4 Conclusion

This paper put forward the consensus that kernel level implementations of inter process communication mechanisms are favorable to those that implement such mechanisms at other non-kernel levels. It showed that a given bench program run in parallel in a distributed cluster in WAN is more efficient when programmed and ran under a kernel level implementation of IPC, namely DIPC, than when programmed and ran under a non-kernel level implementation of IPC, namely MPI. However, although the case is for the kernel level implementation of IPC, other parameters which may be important to high performance computation apart from efficiency of IPC, like extensibility, security, ease of use, etc., need further investigation.

References

[1] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard”, Supported in Part by ARPA and NSF under Grant ASC-9310330, the National Science Foundation Science and Technology Center Cooperative, 2003.

[2] M. Sharifi, K. Karimi, “DIPC: A System Software Solution for Distributed Programming”, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 97), Georgia University, U.S.A., April 1997.

[3] M. Sharifi, K. Karimi, “DIPC: A Heterogeneous Distributed Programming System”, In Proceedings of the 3rd Annual Int. Computer Conference of the Computer Society of Iran, Iran University of Science and Technology, Tehran, 1997.

[4] M. Sharifi, K. Karimi, “DIPC: The Linux Way of Distributed Programming”, Linux Journal, Issue 57, 1999. pp. 10-17.

[5] V. Bala and S. Kipnis, ” Process Groups: a Mechanism for the Coordination of and Communication Among Processes in the Venus Collective Communication Library”, Technical Report, IBM T. J. Watson Research Center, October 1992.

[6] LAM/MPI Team, “LAM/MPI User’s Guide Version 7.1.2”, Open Systems Lab, http://www.lam-mpi.org/ , 2006.

[7] “Verari System Software Inc”, http://www.verarisoft.com/mpi_pro_2.php , 2007.

[8] William Gropp. MPICH2: A new start for MPI implementations. In Dieter Kranzlm¨uller, Peter Kacsuk, Jack Dongarra, and Jens Volkert, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Number LNCS2474 in Lecture Notes in Computer Science, 2006.

[9] J. J. Dongarra, S. W. Otto, M. Snir and D. Walker, “An Introduction to the MPI Standard” Technical Report UT-CS-95-274, University of Tennessee, 1995.

[10] G. Jost, H. Jin, “Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster”, NASA Ames Research Center, Fifth European Workshop on OpenMP (EWOMP03) in Aachen, Germany, 2003.

[11] Voltaire®HCA 400, Release Notes, “Linux InfiniBand Stack for Voltaire® HCA 400 Revisions 3.0.0”, Document No. 399Z30200, 2005.

[12] Stallings William, "Operating Systems: Internals and Design Principles" (5th Edition), Hardcover, 2004.

[13] T. Maeda, “Safe Execution of User Programs in Kernel Mode Using Typed Assembly Language”, Ms Thesis, The Graduate School of The University of Tokyo, 2002.

[14] O. Gl¨uck, J. Lamotte, A. Greiner, “The Influence of System Calls and Interrupts on The Performance of a PC Cluster Using a Remote DMA Communication Primitive”, University P. & M. Curie LIP6 Laboratory, 2002.