RPPA: A Remote Parallel Program Performance Analysis Tool

Yunlong Xu, Zeng Zhao, Weiguo Wu*, and Yixin Zhao
School of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an, China
Email: xjtu.ylxu@gmail.com, wgwu@mail.xjtu.edu.cn

***Abstract*—Parallel program performance analysis plays an important role in exploring parallelism and improving efficiency of parallel programs. To ease the performance analysis for remote programs, this paper presents a remote parallel program performance analysis tool, RPPA (Remote Parallel Performance Analyzer), which is based on dynamic code instrumentation. A hierarchical structure is adopted by RPPA which consists of 3 parts: client, server and computing nodes. Performance analysis tasks are submitted to the server via the graphical user interface of the client, and then actual analysis processes are started on the computing nodes by the server to collect performance data for visualization in the client. The performance information gained by RPPA is comprehensive and intuitive, hence it is quite helpful for users to analyze performance, locate bottlenecks of programs, and optimize programs.**

***Index Terms*—parallel programming tools, program performance analysis, dynamic code instrumentation, performance visualization**

I. INTRODUCTION

In the past few years, parallel computers have grown very fast, while software for parallel computing has grown comparatively slow. This situation makes the efficiency of parallel systems relatively low, thus hardware performance cannot be fully utilized [1]. Therefore software becomes the bottleneck of parallel computing, and limits the widely use of parallel computers. For instance, parallel program development is challenging due to the lack of effective tools for coding, debugging, performance analyzing and optimizing.

Motivated by the preceding challenge, a remote visualization parallel program performance analysis tool, RPPA, which aims to help programmers to optimize performance of applications and make full use of parallel computing resources, is designed and developed based on the survey of existing tools and our former research of a parallel program integrated development environment (i.e., IDE). RPPA provides a solution for performance analysis of MPI [4] applications running on parallel computers with SMP nodes. To gain comprehensive performance information of programs, a dynamic instrumentation based approach, which includes modifying, deleting, and inserting code to change the execution behaviors of programs, is applied to the performance measurement. The framework of RPPA consists of three parts: client, server and computing nodes. A performance analysis task is committed to the server through the client graphical user interface (i.e., GUI); a server daemon running on the server then starts the program performance data collection processes on several computing nodes; later, the server gathers performance data files from computing nodes, and sends back to the client for visualization.

The reminder of this article is organized as follows: First, we consider related work and point out RPPA's strength through comparison with existing tools in the next section. In Section 3, we describe the overall architecture of RPPA. After that, we present the methodology applied in the client, the server and the computing nodes in detail respectively in Section 4. In Section 5, we present the evaluation of a RPPA prototype. Section 6 is a discussion. Finally, we consider future work and conclude the paper.

II. RELATED WORK

Performance analysis is crucial to parallel program development. This work combines knowledge of various fields, e.g., parallel computing, computer architecture, data mining. The related work of program performance analysis has been carried out by many organizations and agencies.

MPE [2] is an extension of MPICH [3] which is a portable implementation of MPI. It consists of a large number of programming interfaces and examples for correctness debugging, performance analysis and visualization. It supports several file formats to log performance information of programs, which can be viewed by Upshot, Jumpshot [5], and other visualization tools.

Paradyn [6], developed by University of Wisconsin, is a software package which aids in analyzing performance of large-scale parallel applications. It supports analyzing MPI and PVM [7] applications. The cause of performance problems is systematically detected by inserting and modifying code of programs automatically. And performance bottlenecks are automatically searched by means of a W^3 (i.e., When, Where, and Why) model.

TAU [8] is a parallel program performance analysis system developed by University of Oregon, Juelich

The journal article is based on the conference paper, "Research and Design of a Remote Visualization Parallel Program Performance Analysis Tool," which appeared in PAAP'10.

*Corresponding author: Weiguo Wu

doi:10.4304/jsw.6.12.2399-2406

Research Center, and Los Alamos National Laboratory together. It focuses on system robustness, flexibility, portability and integration of other related tools, and supports multiple programming languages, including C/C++, Fortran, Java, Python. A MPI wrapper library is provided to analyze performance of MPI functions, as well as a log format conversion tool.

VENUS (Visual Environment of Neocomputer Utility System) [17] is a parallel program performance visualization environment developed by Xi'an Jiaotong University. It provides solutions for monitoring, analyzing and optimizing large-scale PVM parallel programs based on profiling method. System supports real-time and post-mortem performance visualization. Measurement code is automatically inserted into source code of programs to collect performance data for data analysis and performance visualization. In this way, performance analysis helps users to optimize their programs.

Open|SpeedShop [9], a dynamic binary instrumentation based performance tool using DPCL [16]/Dyninst [12], aims to overcome a common limitation of performance analysis tools: each tool alone is often limited in scope and comes with widely varying interfaces and workflow constraints, requiring different changes in the often complex build and execution infrastructure of the target application; thus it provides efficient, easy to apply, and integrated performance analysis for parallel systems.

HPCToolkit [10], developed by Rice University, is an integrated suite of tools that supports measurement, analysis, attribution, and presentation of application performance for both sequential and parallel programs. HPCToolkit can pinpoint and quantify scalability bottlenecks in fully-optimized parallel programs and multithreaded programs at a low cost. Call path profiles for fully-optimized codes can also be collected without compiler support.

The tools enumerated above have a common defect that performance visualization and data collection are both carried out on parallel computers or clusters. It is hard for users who are not familiar with parallel systems to install and apply these tools. The preceding defect hinders the widely use of parallel program development environment and parallel computing resources to some extent. Therefore, there is an urgent need for a user-friendly remote parallel program performance analysis tool. RPPA is to meet this need. Users connect to remote parallel systems over the internet via RPPA's client GUI, which is part of a parallel program IDE built on Eclipse Plug-in [11] technology. Interactive operations and performance visualization are both carried out in the client, while actual operations are executed on remote clusters which are transparent to common users. RPPA is easy to use and portable for Windows, Linux and other operating systems.

III. OVERALL ARCHITECTURE

The RPPA tool adopts a hierarchical structure, through which users are shielded from the complexity of parallel systems. The performance information of programs is presented in RPPA's GUI in an intuitive way, which is quite helpful to locate performance bottlenecks and improve efficiency of programs. A performance analysis task is committed to the server through the graphical client; a daemon running on the server starts the program performance data collection processes, whose core step is dynamic instrumentation, on several computing nodes; the server gathers performance data files from the computing nodes, and sends back to the client for visualization.

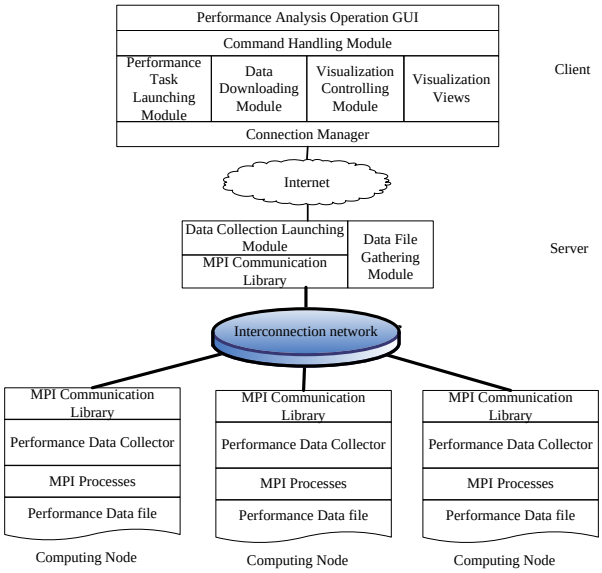

Figure 1. Overall architecture of RPPA

RPPA is customized to the cluster structure. The overall architecture of RPPA is depicted in Figure 1. The client connects with the server over the internet, and the computing nodes connect with the server, which can also be a computing node, over high-speed interconnection network. Users control the whole system through a GUI, which includes two core modules for submitting performance analysis tasks, downloading and visualizing performance data. The server-side operation details (e.g., how the run-time processes are assigned to the computing nodes) are transparent to users. As a result, operations are greatly simplified for users who are not familiar with the bottom parallel cluster structure. The server, entry node of a parallel cluster, is responsible for maintaining the connection between the client and the cluster, accepting instructions from the client and then starting performance analysis processes on computing nodes to collect performance data. Actual performance analysis tasks and program execution tasks are carried out on the computing nodes: RPPA creates user program processes, into which measurement code is inserted, controls its execution, generates performance data files, integrates and analyzes the files when analysis processes finish.

IV. DETAILED DESIGN

A. Client

Client provides an interface for users to commit

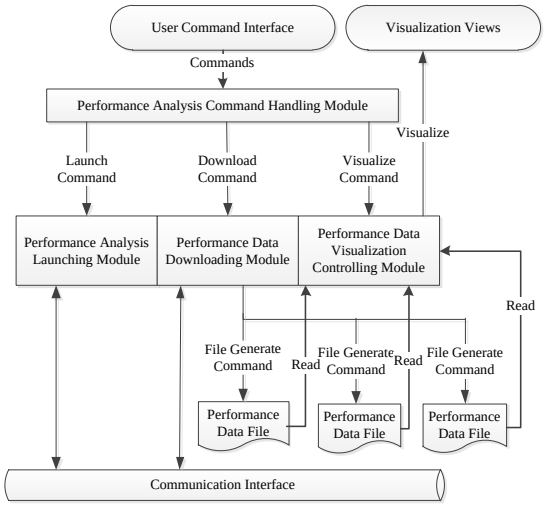

Figure 2. Structure of client

analysis tasks and visualizes performance analysis result. The GUI implemented with Eclipse Plug-in technology greatly simplifies users' operation. Client is composed of four functional modules: performance analysis command handling module, performance analysis launching module, performance data downloading module, and performance data visualization controlling module. The relationship between the four modules is shown in Figure 2.

Performance Analysis Command Handling Module

This module implemented by extending Plug-in extension points of Eclipse platform. Graphical items of performance analysis (e.g., buttons, menus, pop-up lists and editors) are added to the original GUI of Eclipse platform. Each item corresponds to a specific performance analysis operation. This module supports interactions between users and RPPA.

Performance Analysis Launching Module

This module sends commands to the server and starts remote performance analysis task. A customized Eclipse launching mode is implemented by extending the *launchModes* extension point of the original platform, and the actions carried out while launching are implemented by extending the *launchDelegates* extension point of Eclipse.

Performance Data Downloading Module

Performance data files are distributed among the computing nodes when analysis task finishes due to the C/S architecture adopted by RPPA. This module gathers and integrates the distributed performance data files, and then transfers the integrated file to the client for visualization.

Performance Data Visualization Controlling Module

The structure of this module is shown in Figure 3. Performance information can be viewed in multiple aspects through various display commands and views RPPA provides. A command deals with a specific kind of data set; and a view displays a kind of performance information. It is worth mentioning that, in accordance with the characteristics of parallel programs, a view of

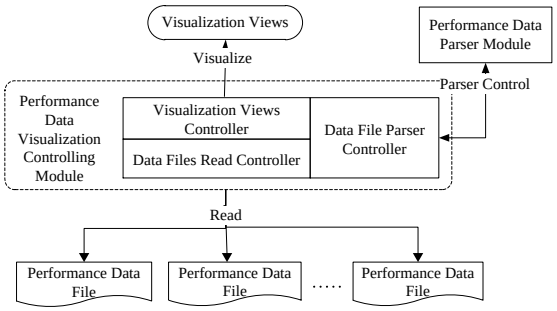

Figure 3. Performance visualization control module of client

performance comparison between multiple processes is provided.

Performance data visualization is an important means to help users analyze program performance and pinpoint performance bottlenecks. Because it is difficult to measure a program's performance based directly on large volume of performance data files of various entries, not to mention that the data volume grows rapidly as application scales. Therefore, RPPA offers vivid and intuitive graphical charts to help users visualize performance information analyze program performance and then locate bottlenecks quickly.

B. Server

Server is the communication relay between the client and the computing nodes: it serves as the entry of a cluster, which is composed of several computing nodes, and controls execution of tasks on computing nodes; it also receives commands from the client and responses to these commands. The functionalities of the server include launching performance analysis tasks, gathering performance data files from computing nodes, and maintaining communication with the client. The performance data file gathering module is the core of the server.

According to when collected data is analyzed and visualized, program performance analysis falls into two categories: real-time analysis and post-mortem analysis. Each mode has its strength: the real-time mode can adjust data collection during running time of tasks, while the post-mortem mode introduces minor perturbation into the original program. As to RPPA, if a real-time mode is adopted in the C/S structure, the server has to receive performance analysis instructions frequently, deal with these instructions, and then transmit them to the computing nodes on which they are actually executed, and finally large volume of data is transferred back to the client while execution. The whole process which is greatly affected by network would introduce major perturbation to the original program. Thus the post-mortem mode is adopted in RPPA.

Unlike the launch process of common MPI parallel program, the launch process of performance analysis task has to ensure that all MPI processes running on the computing nodes are under the control of the program performance analysis tool, specifically the control of the performance analysis launching module. In this way, measurement code is instrumented into user programs, and then performance data can be collected. The launch

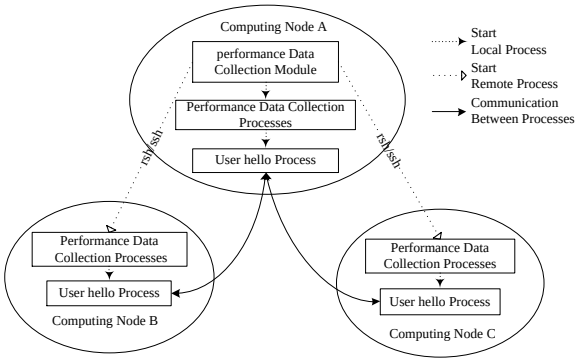

Figure 4. MPI program starting process under the control of data collection module

process of MPI parallel program under the control of launching module is shown in Figure 4. When receiving a performance analysis launch command, the launching module of the server (e.g., Node A in Figure 4) analyzes the parameters of the command, creates a performance analysis task, sets some parameters (e.g., name and full path of the user program, number of processes) for the task, and then starts data collection processes on the computing nodes. The data collection processes analyze the parameters and then create hello processes based on the name of the user processes and the parameters of the performance analysis task. After that, the hello processes is considered to be part of the whole task, and managed by MPI runtime environment.

C. Computing Nodes

The computing nodes, on which MPI runtime environment is deployed, is the substrate of RPPA. A bijection is set up between MPI processes and performance data collection processes on the computing nodes. Data collection processes start and control MPI processes. The performance data collection module of the computing nodes is launched by the performance data collection launching module of the server. MPI processes are started by the data collection module according to some parameters (e.g., the absolute path of the MPI program). And then the collection module inserts measurement code into MPI processes, and integrates generated performance data files when finishes.

The main purpose of performance analysis is to help users to locate program bottlenecks, i.e., code segments which account for large portions of running time. Besides running time, the number of function calls is also collected to estimate the average running time of each function. As to MPI parallel program, communication volume of MPI functions also concerns programmers a lot. To sum up, three kinds of data, i.e., running time, number of function calls, communication volume, need to be collected.

Performance data is collected by means of code instrumentation [12], which includes modifying, deleting, and inserting code to change the execution behaviors of programs. According to the timing of code inserting, instrumentation can be divided into two categories: static and dynamic code instrumentation. The static one inserts code before execution of programs and the dynamic one, per contra, during execution. The static instrumentation introduces little perturbation into the original programs, but the information it collects is not comprehensive; while the dynamic instrumentation is able to collect comprehensive performance information at the cost of major perturbation. Using dynamic instrumentation, programs are just awakened rather than to be recompiled, relinked and restarted after instrumentation. Apparently, the dynamic instrumentation is more complicated to implement than the static one.

In this paper, we use DynistAPI [12], a dynamic binary instrumentation interface developed by University of Maryland, to insert running processes with binary code. Instrumentation processes (i.e., Mutator) start user program processes which are to be instrumented (i.e., Mutatee), and attaches themselves to the user processes. Measurement code segments (i.e., Snippet) are inserted into user processes by instrumentation processes when user processes are suspended. And performance data is collected when instrumented code of user processes are executed.

The process of performance data collection is described as Algorithm 1. We note that, when no MPI

Algorithm 1: Performance data collection procedure

Input: *A*, Attributes of user task
Output: *PF*, an integrated performance data files

```
define: proc_array, array of MPI process pid
proc_array ← spawn(A)
suspend(proc_array)
foreach i in proc_array
    define: node, name of computing node
    define: file, path of data file
    file ← new(node,i)
    define: procImage, image of user process
    procImage ← getImage(i)
    define: usrModule, image of user process
    usrModule ← search(procImage)
    define: usrFunc_array, array of user function
    func_array ← search(usrModule)
    for j in func_array
        define: snippet, binary code segment
        define: gettimeofday, function to get current time
        define: fscanf/fprintf, function to read/write a file
        snippet ← generate(gettimeofday,fopen/fprintf)
        insert(j, snippet,file)
        if j is a MPI function
            define: commAttr_array, array of communication
                    attributes, e.g., volume, source, destination
            snippet ← generate(commAttr_array,file)
            insert(j,snippet)
        end if
    end for
    wake(proc_array)
    if i finishes
        acknowledge(server,file,i)
    end if
end foreach
while(TRUE)
    if all processes complete
        define: file_array, array of performance file
        PF ← integrate(file_array)
        return PF
    end if
end while
```

communication function is founded, the data collection process skips the step of collecting MPI communications information, and goes ahead rather than returns, because the program being analyzed could be a serial program, which can also be executed by *mpiexec* command in MPI runtime environments.

V. EVALUATION

This section describes the evaluation of a RPPA prototype in two aspects: functional evaluation and perturbation evaluation. Pros and cons of RPPA are also presented at the basis of the evaluations.

A. Test Environment and Test Case

In this paper, the evaluations are carried out on a parallel computing system, a Dawning 4000L cluster, which composed of 18 nodes. Each node is equipped with: Intel (R) Xeon (TM) CPU (3.00 GHz × 2), 2G memory, Broadcom BCM5721 1000Base-T LAN, Linux 2.4 operating system, and OpenMPI [14] 1.2.3 parallel computing environment. The test case is an MPI parallel program solving ocean circulation problem by using two-dimensional *stommel* model developed by Timothy H. Kaise.

B. Functional Evaluation of Performance Visualization

Users can check performance information in 7 different views, including: function running time of single process, function call numbers of single process, communication volume of single process, function running time comparison of multi-processes, function call number comparison of multi-processes, communication volume comparison of multi-processes, and space-time diagram of all processes. In this evaluation, the test program is composed of 5 performance data collection processes, each of which creates a *stommel* process on 2 computing nodes separately. Data files are downloaded and performance information is visualized when data collection task finishes. Function running time of one process is shown in Figure 5; function running time comparison of multi-processes is shown in Figure 6; and space-time diagram of all processes is shown in Figure 7.

Functional evaluation shows that the performance data collected by RPPA which is based on dynamic instrumentation is comprehensive, because performance data of all kinds of functions (i.e., user-defined functions, external library functions, standard MPI communication functions) can be collected through dynamic instrumentation. This evaluation also shows that, RPPA meets the requirement of program performance analysis, and brings great convenience to programmers by providing a user-friendly GUI.

C. Perturbation Evaluation

The perturbation introduced into the original programs by an analysis tool is an important metric to measure the performance of the analysis tool itself [15]. The perturbation can be reflected by the running time of programs. And the degree of perturbation can be estimated by comparing the running time of a program with measurement code inserted and the running time of the original program.

What concerns us most is the difference of running time between the original program and the program with code inserted. The running process of parallel programs is affected by many factors. And in this paper, perturbation evaluation is influenced mainly by network status and system background workload status. Thus a single node, on which there is no process running besides Linux system processes, is used to run the evaluation programs. In this way, the network and system load factors are ruled out.

The Paradyn performance analysis tool, which also uses the Dyninst tool for dynamic instrumentation, is selected to compare with RPPA. To make the evaluation data more representative, we carry out 3 kinds of tests using the *stommel* program, including running program without interference of any performance analysis tools, with interference of RPPA, and with interference of the Paradyn tool. 10 groups of these 3 kinds of tests are conducted. The test results are shown in Figure 8. Test type 1 represents running time without interference of any analysis tools, type 2 represents running time with RPPA, and type 3 represents the Paradyn tool.

It can be seen that, under the same conditions (i.e., the same running environment, and the same types of performance data to be collected), the running time of the stommel program with interference of RPPA is closer to the running time without any tools than to the running time with Paradyn. From this perspective, RPPA surpasses the Paradyn program performance analysis tool.

However, Figure 8 also shows that both RPPA and Paradyn introduce major perturbations into the original programs. The primary cause is that they both build on dynamic instrumentation which intrudes the original programs frequently by coping, modifying, transferring, and inserting code during runtime of programs. But for the same reason, the performance data collected by the analysis tools based on dynamic instrumentation is comprehensive, because performance information about all kinds of functions can be gained. Obviously, major perturbation is the price of comprehensive performance information.

VI. DISCUSSION

In this section, we discuss some new ideas and methods to improve RPPA. As mentioned in the previous section, dynamic instrumentation based performance analysis tool introduces major perturbation into the original programs. Hence we seek a novel approach to collect performance data on the server and computing nodes. A multi-level instrumentation based data collection approach is proposed in our recent research. The implementation of this approach is part of our recent and future work, and is planned to be presented in follow-up work. In this section, we first discuss why this approach would reduce the perturbation, and how it works.

As analysis in section 7 says, dynamic instrumentation collects comprehensive program performance

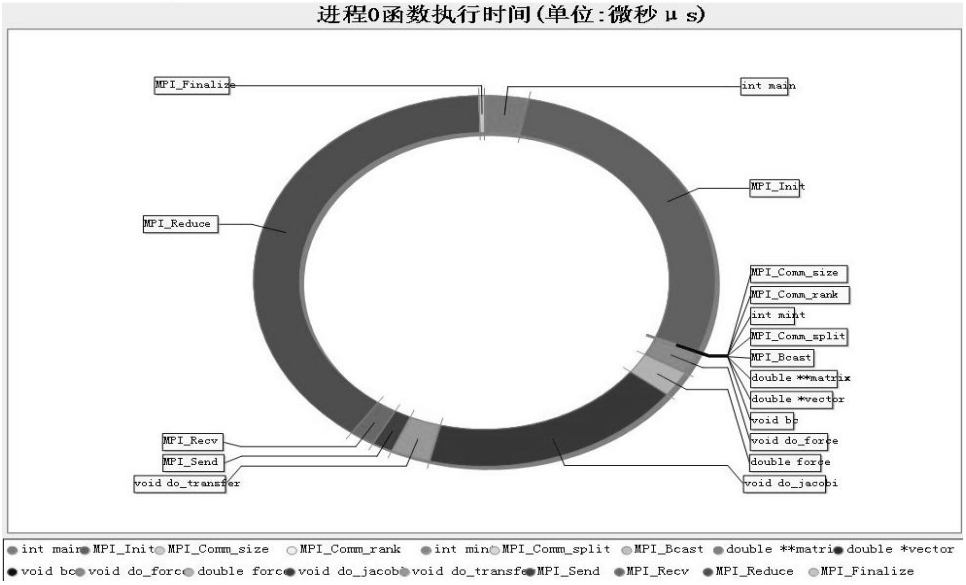

Figure 5. Function running time of one process

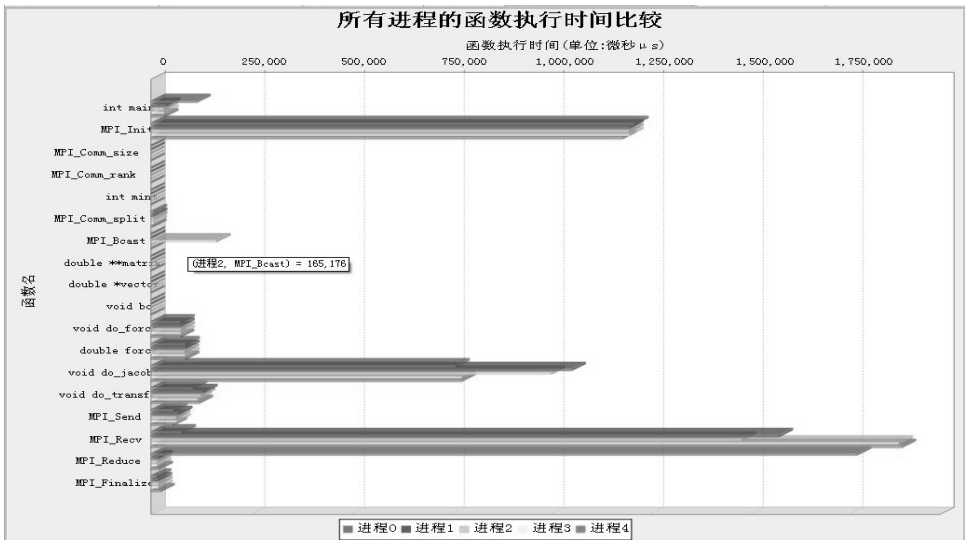

Figure 6. Function running time comparison of multi-processes

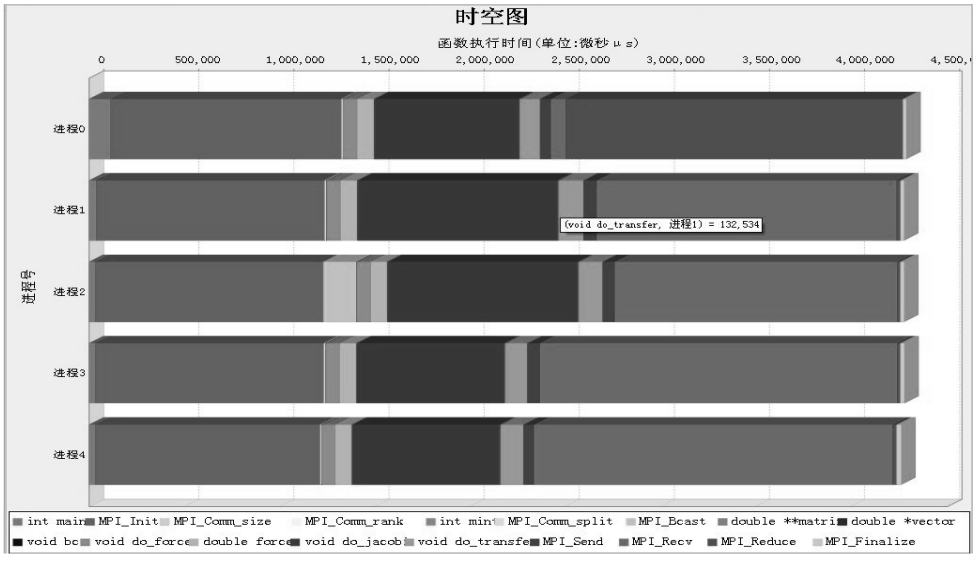

Figure 7. Space-time diagram of all processes

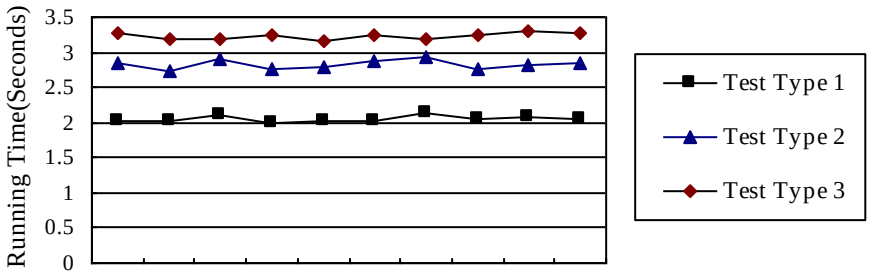

Figure 8. Running time comparison of *stommel* program interfered by different analysis tool

information at the cost of major perturbation. Therefore we propose the multi-level instrumentation based approach, through which the comprehensiveness of performance information would not be compromised. The essential of this approach is to collect different kinds of information in distinctive proper ways; and only do dynamic instrumentation when really necessary. We separate the functions, whose performance information concerns programmers most, into three categories: user-defined function, standard MPI function, external library function. Performance information of user-defined function is collected through a source-level instrumentation [8] based method. As to standard MPI functions, we adopt PMPI [4], the MPI standard profiling interface, to collect its performance information. And for external library function, to the best of our knowledge, the dynamic instrumentation is the best method to collect its performance information. We note that the source-level instrumentation could be replaced by the dynamic one when there are no other files than the binary ones for some reasons. We also note that dynamic instrumentation could be skipped if the external library functions' performance information is not a concern or little perturbation can be tolerated by users. And still there is plenty of performance information presented to programmers even if the external library functions are ignored.

By using this multi-level based approach, the perturbation introduced into the original programs would surely be at a lower level, and comprehensiveness of performance information would also not be compromised. Firstly, the source-level based instrumentation introduces little perturbation into the original programs because only a few measurement code is inserted into the source files at the entry and exit point separately when compilation. And all the information that the dynamic approach collects can also be collected by this source-level one. Secondly, when using PMPI profiling to collect standard MPI function's performance information, there is no need to copy, modify, transfer, and insert code during runtime of programs. Because all *MPI_* prefix functions are substituted by equivalent *PMPI_* ones in which some additional measurement code is added. Thus less perturbation is introduced. And also the comprehensiveness of performance information would not be compromised.

The philosophy of this tool resembles SaaS (i.e., Software-as-a-Service) in cloud computing which emerges and then receives significant attention in the media recently. The cloud conceals the complexity of the infrastructures from common users through internet. Cloud computing partly refers to the software delivered as services over the internet. Users could shift computing power away from local servers, across the network cloud, and into large clusters of machines hosted by companies such as Amazon, Google, IBM, Microsoft, Yahoo! and so on. We plan to transplant our parallel program IDE's client, which include RPPA's client, into the internet environment by rewriting its GUI with some web programming Languages. Then the Eclipse platform would be replaced by an internet explorer. And with the support of a daemon running on the server node of a remote cluster, users could develop parallel programs remotely over the internet on laptop rather than on parallel cluster. In this way, great convenience would be brought to programmers, while excellent portability of this tool would also be achieved.

VII. CONCLUSION AND FUTURE WORK

A remote parallel program performance analysis tool, RPPA, is designed based on the research of the existing program performance analysis tools, most of which are not user-friendly. RPPA meets the urgent need of parallel program performance analysis as mentioned in Section 1. The workflow of RPPA is as follows: performances analyses tasks are submitted to the server through the client; then performance data collection processes are started on the computing nodes by the server to collect performance data; at last, program performance information is visualized in the client. RPPA is user-friendly, easy to use, and the collected performance information is comprehensive, so it is quite helpful for users to analyze performance, locate bottlenecks of programs, and optimize programs.

However, there is a common shortcoming of dynamic instrumentation based performance analysis tools: major perturbation would be introduced into the original programs. So research on multi-level instrumentation based performance analysis tool which brings minor perturbation is a part of our future work. Our future work also includes providing supports for variety kinds of parallel programming models and heterogeneous high performance computing systems.

ACKNOWLEDGMENT

This work was supported by the National High-Tech Research and Development Plan of China under Grant No. 2009AA01A131 and No. 2009AA01A135; the Chinese Ministry of Science and Technology under Grant No. 2009DFA12110.

We would like to thank all the people who give helpful suggestions to our work, especially the reviewers of PAAP'10 for their comments on the original conference paper [18]. And we also would like to thank all the members of the integration and application of heterogeneous resources service project team, department of computer science and technology, Xi'an Jiaotong University, for their supports to our work.

REFERENCES

[1] C.C. Chen and M.X. Chen, "A generic parallel computing model for the distributed environment," *Proc. 2006 7th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 4-7, December 2006.

[2] A. Chan, W. Gropp, and E. Lusk, "User's guide for MPE: extensions for MPI programs," *Technical Report*, MCS-TM-ANL-98, Argonne National Laboratory, 1998.

[3] W. Gropp, "MPICH2: A new start for MPI implementations," *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, vol. 2474, pp. 37-42, 2002.

[4] Message Passing Interface Forum, "MPI: A message-passing interface standard," *Technical Report*, University of Tennessee, 1994.

[5] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *International Journal of High Performance Computing Applications*, vol. 13, pp. 277-288, 1999.

[6] B.P. Miller, M. Callaghan, G. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, et al, "The Paradyn parallel performance measurement tool," *Computer*, vol. 28, pp. 37-46, November 1995.

[7] W. Gropp and E. Lusk, "Goals guiding design: PVM and MPI," *Proc. IEEE International Conference on Cluster Computing*, pp. 257-265, 2002.

[8] S. Shende and A.D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, pp. 287-311, 2006.

[9] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya and S. Cranford, "Open|SpeedShop: An open source infrastructure for parallel performance analysis," *Scientific Programming*, vol. 16, pp. 105-121, April 2008.

[10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice & Experience*, vol. 22, pp. 1-7, April 2010.

[11] E. Clayberg and D. Rubel, "Eclipse Plug-Ins," *Addison-Wesley Professional*, 2008.

[12] B. Buck and J.K. Hollingsworth, "An API for runtime code patching," *International Journal of High Performance Computing Applications*, vol. 14, pp. 317-329, 2000.

[13] G. Ravipati, A. Bernat, N. Rosenblum, B.P. Miller, J.K. Hollingsworth, "Towards the deconstruction of Dyninst," *Technical Report*, University of Maryland, 2007.

[14] E. Gebrial, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, et al, "Open MPI: goals, concept, and design of a next generation MPI implementation," *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, vol. 3241, pp. 97-104, 2004.

[15] J. Liu, M.M. Shen, and W.M. Zheng, "Research on perturbation imposed on parallel programs by debugger," *Chinese Journal of Computer*, vol. 25, No. 2, pp. 122-127, 2002.

[16] L. DeRose, T. Hoover and J. Hollingsworth, "The dynamic probe class library — an infrastructure for developing instrumentation for performance tools," *Proc. of the 15th International Parallel and Distributed Processing Symposium*, Apr. 2001.

[17] X.H. Shi, Y.L. Zhao, S.Q. Zheng, and D.P. Qian, "VENUS: A general parallel performance visualization environment," *Mini-micro Systems*, vol. 19, pp. 1-7, 1998.

[18] Y.L. Xu, Z. Zhao, W.G. Wu, and Y.X. Zhao, " Research and Design of a Remote Visualization Parallel Program Performance Analysis Tool," *Proc. of the Third International Symposium on Parallel Architectures, Algorithms and Programming*, pp. 214-220, Dec. 2010.

Yunlong Xu, born in Sichuan, China, 1987. He is currently a Ph.D. candidate at School of Electronic and Information Engineering, Xi'an Jiao tong University, China. His research interests include parallel computing, and cloud computing.

Zeng Zhao received his master degree in School of Electronic and Information Engineering, Xi'an Jiao tong University, China, in 2010. Zhao is currently a member of Tencent, Inc., China.

Weiguo Wu, born in 1963, is Professor, Ph.D. supervisor, of School of Electronic and Information Engineering, Xi'an Jiao tong University, China. He received his master and Ph.D. degree in Xi'an Jiaotong University. His research interests include high-performance computer architecture, massive storage system, computer networks, and embedded systems.

Yixin Zhao received his master degree in School of Electronic and Information Engineering, Xi'an Jiao tong University, China, in 2009.