J Supercomput (2009) 48: 43–65
DOI 10.1007/s11227-008-0204-2

Performance-based parallel application toolkit for high-performance clusters

Kuan-Ching Li · Tien-Hsiung Weng

Published online: 9 April 2008

Abstract Advances in computer technology, encompassed with fast emerging of multicore processor technology, have made the many-core personal computers available and more affordable. The availability of network of workstations and cluster of many-core SMPs have made them an attractive solution for high performance computing by providing computational power equal or superior to supercomputers or mainframes at an affordable cost using commodity components. In order to search alternative ways to extract unused and idle computing power from these computing resources targeting to improve overall performance, as well as to fully utilize the underlying new hardware platforms, these are major topics in this field of research. In this research paper, the design rationale and implementation of an effective toolkit for performance measurement and analysis of parallel applications in cluster environments is introduced; not only generating parallel applications' timing graph representation, but also to provide application execution's performance data charts. The goal in developing this toolkit is to permit application developers have a better understanding of the application's behavior among selected computing nodes purposed for that particular execution. Additionally, multiple execution results of a given application under development can be combined and overlapped, permitting application developers to perform "what-if" analysis, i.e., to deeper understand the utilization of allocated computational resources. Experimentations using this toolkit have shown its effectiveness on the development and performance tuning of parallel applications, extending the use in teaching of message passing, and shared memory model parallel programming courses.

K.-C. Li (✉) · T.-H. Weng
Dept. of Computer Science and Information Engineering, Providence University, Taichung, Taiwan
e-mail: kuancli@pu.edu.tw

T.-H. Weng
e-mail: thweng@pu.edu.tw

Keywords Performance monitoring and visualization · OpenMP · MPI · Hybrid model parallel application · Cluster computing

1 Introduction

In recent years, the merge of rapidly advancing computer and networking technologies has resulted in a new computing infrastructure named the cluster or network of workstations, also known as the cluster of SMPs, multi-core PC-based clusters. The potential of this computing infrastructure has attracted attention from the computing industry, mainly due to their scalability, as also their ability to provide significant cost effective computing, since they rely solely on commodity technology, and efficiently support both single processor interactive processing and large batch parallel processing.

Personal computers (or multi-core/many-core multiprocessors PC) are typically interconnected through a high-speed network, such as Gigabit Ethernet, SCI, Myrinet or Infiniband, and they run commodity operating systems, such as FreeBSD, SUSE, Fedora or any other UNIX-like operating systems. A number of software tools have been developed to support concurrent and distributed computing in network of workstations, including popular tools such Pthreads [25], OpenMP [24], Parallel Virtual Machine (PVM) [11], and Message Passing Interface (MPI) [1, 12, 19]. MPI and PVM are explicit message passing libraries where tasks communicate through message exchange, while Pthreads explores the degree of concurrency through a multithreaded programming model, and OpenMP is used to explore concurrencies in a shared memory programming model.

Cluster computing is an area of growing interest to support parallel and distributed applications, and a cost effective and convenient platform for high-performance computing, widely used to improve the performance of many complex computational problems and applications with intensive demands for computational power. Problems, such as the grand challenges, are fundamental in science and engineering with broad scientific and economic impact, which solution can definitely be advanced with high-performance computing [2]. Many of these applications are I/O intensive and the limited bandwidth of the I/O subsystem of the cluster computing systems is an important bottleneck usually ignored [4]. Therefore, the performance of parallel I/O primitives is critical for the overall computing system performance.

Parallel applications can behave in a number of unexpected ways, due to their complex structure, the parallel system on which they run, the number of computing nodes used to execute the application in a cluster environment, the dataset used by the parallel code, the regularity of applications, the complexity of algorithms, the variability in programming environments, the heterogeneity of software and hardware platforms, among others [13]. Additionally, effective partitioning, allocation and scheduling of application programs on network of workstations are crucial to obtain good performance; the performance is very sensitive to the strategy used to distribute data among computing nodes or processors [17].

With the continuous increase of workstation computing power and communication speed, cluster computing has become an inexpensive way to execute scientific

applications. Though, in the same proportion, there is an increase need for performance tools that support these platforms, since it is a difficult task to allocate the entire work in an optimal manner. It is hard to know the exact optimal proportion of hardware resources, even if the code to be executed is in its best way, and thus, the optimization of the distributed application is still an open problem.

The ability of performance technology to keep pace with the growing complexity of parallel and distributed systems depends on robust performance frameworks that can at once provide system-specific performance capabilities and support high-level performance problem solving. Flexibility and portability in empirical methods and processes are influenced primarily by the strategies available for instrumentation and measurement, and how effectively they are integrated and composed. For many parallel applications, efficient use of a fast interconnection network is essential for good performance. Finding the right trade-off between parallel application type and construction and ideal number of computing nodes to execute the application is difficult in general. The way to tailor parallel application toward improving its overall performance in a given cluster system is another issue to be investigated.

This paper presents a software toolkit for performance visualization and analysis of MPI/OpenMP parallel applications, describing how it addresses diverse requirements for performance observation and analysis, as also an integrated graphical toolkit for creating, compiling, and executing parallel programs. During the development of parallel applications, we investigate the effects of alternative data-transfer methods using a number of shared memory and message passing constructions available, where we also evaluate the influence of heterogeneity in systems and network.

As any performance evaluation and analysis methodology, high-level abstraction of application plays an important role. Based on the message passing MPI programming paradigm, a class of timing graphs named *DP*Graph* has been proposed in [17]. The main objective in proposing such novel representation is to describe concisely program parallelism for parallel programs with parallel constructions, i.e., simultaneous executions of tasks, as well as the communication and synchronization relationships among these parallel computations. In particular, this representation is defined only by the program and is independent of runtime input values or computational results.

The remainder of this paper is structured as follows: Sect. 2 brings motivation for the design and implementation of the proposed toolkit and background for this research are discussed, while in Sect. 3, we briefly discuss the implementation of this toolkit and its major features, and Sect. 4 presents experimental results using this toolkit. Finally, conclusions and future works for this research are discussed in Sect. 5.

2 Background and motivation

The design of the proposed toolkit involves cluster performance monitoring, parallel timing graph representation, and application performance evaluation approaches. These approaches are discussed separately in subsections that follow next.

2.1 OpenMP, MPI and hybrid model MPI/OpenMP

MPI is a message passing standard defined by vendors and academic research centers and designed for distributed memory systems, that is, involving explicit data parallelism and implicit parallelism. OpenMP [24] is an industrial standard for shared memory parallel programming agreed on by a consortium of software and hardware vendors. It consists of a collection of compiler directives, library routines, and environment variables that can be easily inserted into a sequential program to create a portable program that will run in parallel on shared-memory architectures. It is easier for a nonexpert programmer to develop a parallel application under OpenMP than in the de facto message passing standard MPI [14]. OpenMP also permits the incremental development of parallel code. Thus, it is not surprising that OpenMP has quickly become widely accepted for shared-memory parallel programming; though most current compilers support it to run only within a node of shared-memory address space machine.

Furthermore, in order to run the application on cluster of SMPs or cluster of many-core PCs, a hybrid model of MPI/OpenMP is one of the required models. In a hybrid model of MPI/OpenMP, a node is mapped to an MPI process, which realized by MPI communication functions for internode communication and processors within the node are mapped to OpenMP threads, which communication can be accessed directly to the shared-memory address space by all the processors in a same node. The potential benefits of the hybrid model MPI/OpenMP have been discussed and found in [29].

2.2 Performance visualization

In the last few years, a number of powerful performance visualization tools have emerged and are available for graphical visualization of parallel applications. They are essentially "discrete event monitoring" tools, which are able to display time-line information of individual parallel processes and show graphically active communication events during the execution. Research groups have concentrated efforts in the development of tools to help application developers and users in finding and correcting performance anomalies and inefficiencies in parallel programs under development. Some of the major advantages to visualize parallel applications using such tools are to perform qualitative latency/bandwidth model comparisons, and also the amount of overlapping computations and communication.

Examples of such performance visualization tools include VAMPIR [31] and DIMEMAS [8]. These tools display execution performance results in high resolution images, based on data acquired during execution and stored in a specific database, e.g., RRD [28], and later display through HTML pages. Multi Router Traffic Grapher (MRTG) [21], based on scripting language Perl and C, is widely used to graph all sorts of data for network attached devices. Basically, it generates HTML pages containing PNG images that provide a real-time visual representation of the network traffic. Perl scripts read the traffic counters in a stored file, while a fast C program logs the traffic data file. Round Robin Database (RRD) is a system that stores and displays time-series data (e.g., network bandwidth and average load) in a compact

way; in parallel, the major feature is that it does not expand over time. Several well-known tools, such as Ganglia Cluster Toolkit [10], CACTI [5], and NMS [23], are implemented by independent teams around the world using the RRD tool.

As listed, tools such as Ganglia and CACTI can only show performance data over time, as each computer node of a cluster system, and unfortunately, not possible to show particular time periods, such as the start and the end of execution of parallel applications in a cluster system. In addition, VAMPIR and DIMEMAS are available commercially only, with quite a high license fee, as also not flexible to display performance data for evaluations and performing "what-if" analysis, according to application developers' needs.

2.3 Parallel graph representation

Writing parallel applications is quite a difficult task, particularly if the application developer plans to efficiently describe his parallel algorithm. This act implies the understanding of message passing and cycle distribution techniques, as also association of statements for best matching on particular situations' needs. Moreover, the execution of a parallel application depends on several factors, in which tasks may interact in complex ways. Basically, a distributed algorithm can be defined as a sequence of local computations, interleaved with stores/loads, device I/O operations, and network communication steps, that is, all types of operations are allowed to be executed concurrently among themselves.

To identify and understand which would be the best structure to be used when writing a parallel application, it is also important to understand what factors affect the performance of this parallel application. Therefore, a representation model is required, where computation, network communication, and local I/O times are evaluated in orthogonal way.

Sequential task graph-based representation, also known as the computation of maximum task execution time, had been comprehensively discussed in [20, 26]. Karypis proposed multi-level graph partitioning in [15], Cain introduced call-graph based search strategy in [6], and Kwok and El-Rewini discussed task graph scheduling in [9, 16]. Unfortunately, no discussions have been done for the parallel version of such timing graph representation.

New representation terminologies have been added to this sequential version of representation, producing a novel version of timing graph capable of representing both sequential and parallel programs. *DP*Graph* [17] is a novel representation design of timing graphs that can represent not only serial programs, but also parallel programs instrumented with communication and synchronization operations, showing execution flow of this parallel application in each computational node of cluster system, where there is present local computations interleaved with communications.

2.3.1 Terminology

In parallel and distributed processing, synchronization and communication operations among computing nodes are fundamental operations. By using message passing concepts to build a parallel program, these message passing constructions can

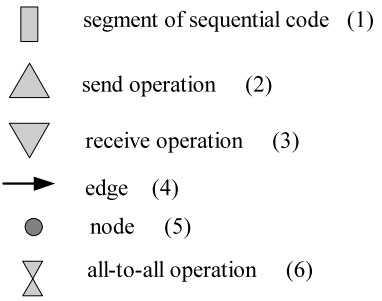

Fig. 1 Elements of DP*Graph representation

```
if (rank==0){ /* Node0 */
#pragma omp parallel for
for (x=0;x,1000;x++)
{
  buf1[x]=x+99;
}
MPI_Send(&buf1,1000,MPI_INT,1,
Msg_tag1,MPI_COMM_WORLD);
}
if (rank==1){ /* Node1 */
#pragma omp parallel for
for (x=0;x,1000;x++)
{
  buf2[x]=x+7;
}
MPI_Recv(&buf1,1000,MPI_INT,0,
Msg_tag1,MPI_COMM_WORLD,&status);
}
```

```
if (rank==2){ /* Node2 */
#pragma omp parallel for
for (x=0;x,1000;x++)
{
  buf3[x]=x+89;
}
MPI_Recv(&buf4,1000,MPI_INT,3,
Msg_tag2,MPI_COMM_WORLD,&status);
}
if (rank==3){ /* Node3 */
#pragma omp parallel for
for (x=0;x,1000;x++)
{
  buf4[x]=x+99;
}
MPI_Send(&buf4,1000,MPI_INT,2,
Msg_tag2,MPI_COMM_WORLD);
}
```

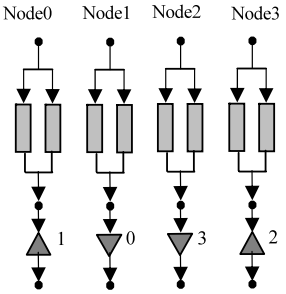

Fig. 2 Code listing and its representation

be performed explicitly using MPI functions such as send, receive, broadcast, reduce, scatter, and others, while parallel programs built using OpenMP constructions merely involves parallel execution of tasks and synchronization. In Fig. 1, we show *DP*Graph* graph elements used to represent parallel programs.

2.3.2 Program representation

The representation of sequential and parallel programs with MPI/OpenMP works out as follows. A sample parallel code and its graph representation using the *DP*Graph* is depicted in Fig. 2.

▽ n means that the computing node is receiving a message sent by computing node n, while △ n means that computing node sends a message to processing node n.

Assuming that the OpenMP number of threads in this parallel code is set to two, it performs the following: all nodes calculate a simple vector sum in a loop of size 1,000, which works are shared among two threads, and then computing node 0 sends the data to computing node 1, while computing node 3 send data to computing node 2. We can observe that although nodes 1 and 2 perform calculations, they actually were not used at any moment during the parallel program's execution.

3 Implementation

The merit of designing and development for this proposed toolkit for visualization and analysis of parallel applications is twofold. One is to develop a practical toolkit to provide performance tuning and analysis of parallel applications, a source where application programmers and users may work on their MPI/OpenMP parallel program, while the other is to handle the representation and performance visualization of experimental results, supported with dynamic multigraphical visualization of application execution information and graph representation, through a number of performance monitoring templates and displays, as depicted in Fig. 3.

The proposed performance toolkit is composed of three main components. The first one is Application Development Manager (ADM), a module where programmers develop their parallel programs, and generate graph representations for performance tuning issues. The second component is Application Visualization Manager (AVM), not only providing the developers a dynamic multi-view graphical charts of application execution information, but also comparative performance charts among version during development stage of a parallel application [17, 18]. The third component is the System Visualization Manager (SVM), responsible to bring dynamic and real-time graphical view of system resources' performance and information customized and displayed through the Ganglia monitoring system [10].

Different from other existing monitoring tools, it is an environment where it is possible to provide performance data analysis and application investigation for parallel programs under development. Additionally, it is designed to make the performance analysis process qualitatively, supporting systems built with a small number of machines as well as one with a large number of computing nodes, noting that the amount of performance information gathered between both systems are tremendously different. During the process of investigation, it makes use of data obtained from successive executions of sequential, MPI, OpenMP, or MPI/OpenMP applications in the performance tuning, in order to observe the difference and variances in performance of this selected application.

3.1 Application development manager

In order to generate parallel representation of parallel programs, this component is implemented using the SUN Java Standard Edition (J2SE) [30]. J2SE is a rapid development environment and toolset for Java-based applications. It provides compiler, tools, and Application Programming Interfaces (APIs) for writing, deploying, and running applets/applications in the Java programming language. Steps to obtain such representation are shown in Fig. 4.

As a first step, the component reads as an input either a sequential or parallel application. Then the program is parsed and the number of computing nodes for the computation of this application program involved is acquired. Sequential and parallel constructions as well as their types are identified and saved into a data file. Next is to construct the *DP*Graph* representation chart based on execution of data obtained from the previous step. Later, the visualization module generates a JPEG format picture for the chart using the Java 2 Platform API Packages. The output chart can be

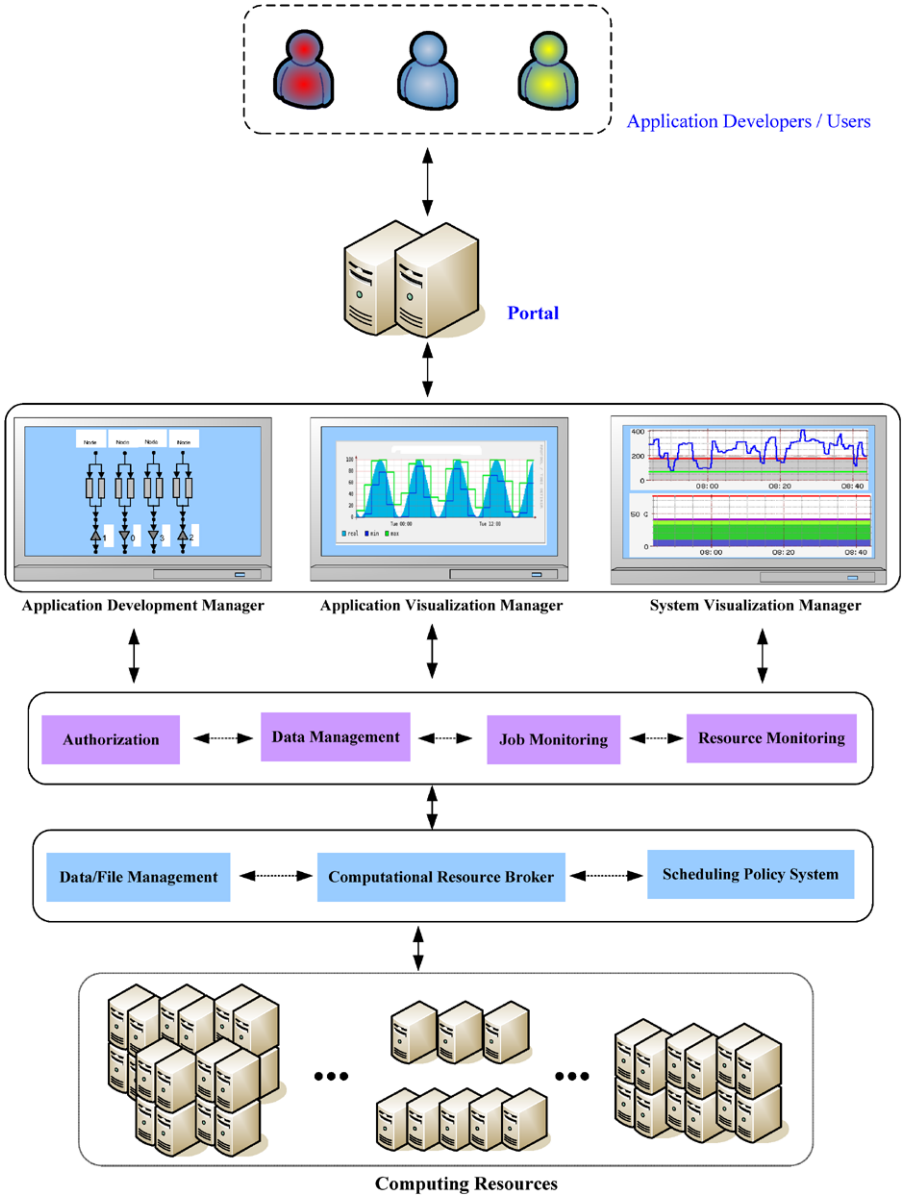

Fig. 3 The proposed toolkit

enlarged and viewed for more detail segments of computing node for a given executed parallel program, and can be combined into the developer's screen to display the complete representation of the parallel program under development. Furthermore, the developer can select any one of executed parallel programs to display on webpage, and compare the difference after the modification of their parallel program.

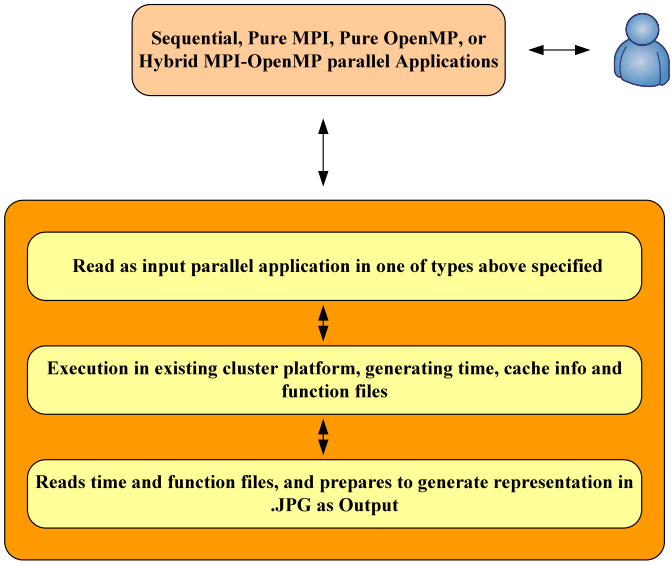

Fig. 4 Steps to obtain representation of a given application

3.2 Application visualization manager

This component is created and implemented using Round Robin Database (RRD) system [RRD, 04]. One of the main reasons to start working on this particular component is that we need a module to visualize the performance data of an application generated during its execution from the start until the end of the execution. Additionally, we need an interface that can process "what-if" analysis, that is, performance tuning of a parallel application during its development phase.

Performance analysis is done by experiments with the program being analyzed and studied to investigate its efficiency. Its output is in both graphical and tabular form, showing both profiling and system data, and such analysis is done on a per-process basis. Graphical interfaces help application developers and users in finding and correcting performance anomalies, inefficiencies, and load imbalances in application programs under development, due to high barrier wait or message passing times, hot spots, poor use of memory hierarchy (mainly cache, local, and remote references).

The proposed component turns possible to visualize both processing and communication activities and to analyze the efficiency of each task in a parallel application. This component enables programmers and developers to identify the area of code being displayed.

3.3 System visualization manager

This component is a web interface to display detailed text and graphical resource information, designed to provide performance data of available resources in cluster systems. Utilization of resources such as CPU, memory, network, I/O is displayed graphically, showing their utilization over time. In addition, this component provides

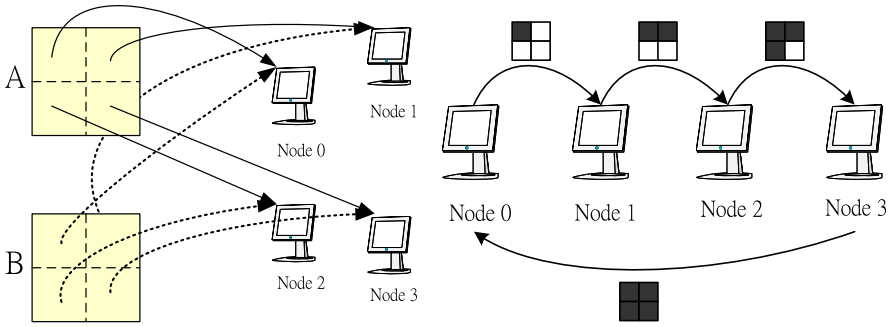

Fig. 5 Parallel version of matrix multiplication—as steps 2 and 3

a process monitoring window, in which we can monitor all ongoing processes in the cluster platform. It can display and update sorted information about processes of each computing node, such as process ID of each task (PID), the user name of the task (User), the priority of the task (PRI), the nice value of the task, negative nice values are higher priority (NI), the size of the task (SIZE), the total amount of physical memory used by the task (RSS), and the amount of shared memory used by the task (SHARE). Each process's state is also shown in this same window, as: S for sleeping, D for uninterruptible sleep, R for running, Z for zombies, or T for stopped or traced. For the purpose of updating the information in real-time basis, we added a button in the window that can retrieve the latest data and refresh the window.

4 Experiments

To illustrate the use of this toolkit, we present the experiment using MPI/OpenMP parallel version of matrix multiplication program and OpenMP SPICE Circuit Simulator Program in the next sections.

4.1 MPI+OpenMP standard matrix multiplication

Given matrices A and B, we compute $C = A * B$ using a standard matrix multiplication method. Steps described below are depicted in Fig. 5.

The computation using four computing nodes was performed in the following way:

1. Values of matrices A and B are initialized in node 0;
2. Corresponding portion of matrices A and B are mapped to corresponding computing nodes, depend on number of cores and processors in each node, the work is further distributed among threads within each node by OpenMP work sharing directives;
3. As computing node 0 concludes its computation, this one fourth of result matrix C is passed to computing node 1. As computing node 1 finishes its computation of one fourth of result matrix C, together with previous one fourth, it will send

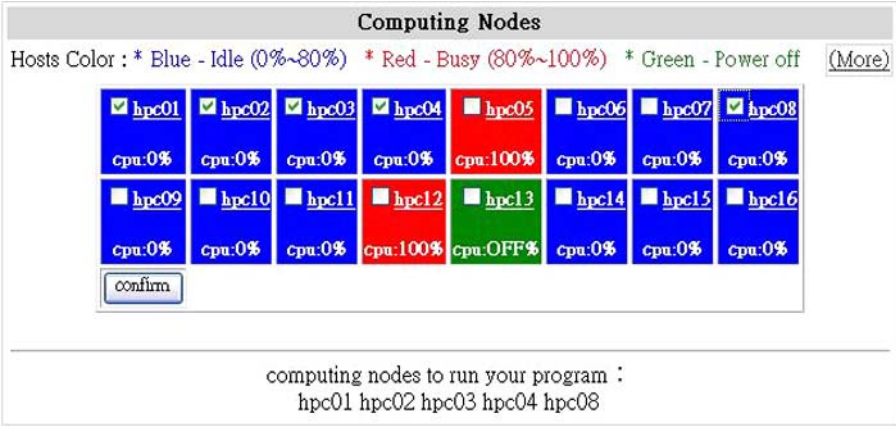

Fig. 6 Computing nodes selection/utilization table

one half of result matrix C to computing node 2. The calculation stops when computing node 3 finishes its calculation, and finally, it passes the result matrix C.

The process of selecting computing nodes is performed by looking at the computing nodes table available, in its simplified view, as shown in Fig. 6. The square boxes in red mean the CPU is busy, with utilization between 80–100%, blue boxes mean that the CPUs utilization is 0–80%, and finally, green boxes mean the computing node is off. This real-time colored interface in computing node table assists the user with the computing node selection. The table in Fig. 7 shows all information of all computing nodes available in our cluster platform, in the full view mode. Again, the CPU utilization is indicated by its color. Still in this table, information including computing node's memory total size and percentage available, number of CPU in each computing node, its speed and OS kernel version are also available.

The file manager window shows us available source and compiled files, performance visualizations, and visualization comparisons already built during successive executions of the developers' parallel programs, as shown in Fig. 8. That is, the developers upload their programs or configurations via upload function, they can handle their files between compile or delete them by checking the corresponding box and clicking the respective command button.

A result of execution of either sequential or parallel application program is shown in Fig. 9 and its corresponding graphical performance data charts in Fig. 10. The developer can either delete or select many previous executions of an application program to combine into one single visualization chart (see Fig. 11), as a matter of comparison. After selecting the combine button, the comparison charts are shown, as in Fig. 12, and this will be stored in the comparison selection list. Moreover, this chart can be used by the developer to perform as much "what-if" analysis he needs during the development of his sequential or parallel code; in other words, code improvement can be assured by comparing the recently tuned application to the previous versions of the same application under investigation. Note that only those computing nodes that are involved in the computation have their performance data chart displayed.

PDPC Cluster	Processor			Memory		OS	
	CPU load	Speed (MHz)	Number	Memory Used	Size (MBytes)	Type	kernel
hpc01	0%	1991	1	72%	991	Linux	2.4.22-1.2115.nptl
hpc02	0%	1991	1	81%	991	Linux	2.4.22-1.2115.nptl
hpc03	0%	1991	1	47%	991	Linux	2.4.22-1.2115.nptl
hpc04	0%	1991	1	48%	991	Linux	2.4.22-1.2115.nptl
hpc05	100%	1991	1	51%	991	Linux	2.4.22-1.2115.nptl
hpc06	0%	1991	1	47%	991	Linux	2.4.22-1.2115.nptl
hpc07	0%	1991	1	97%	991	Linux	2.4.22-1.2115.nptl
hpc08	0%	1991	1	48%	991	Linux	2.4.22-1.2115.nptl
hpc09	0%	1991	1	54%	991	Linux	2.4.22-1.2115.nptl
hpc10	0%	1991	1	41%	991	Linux	2.4.22-1.2115.nptl
hpc11	0%	1991	1	48%	991	Linux	2.4.22-1.2115.nptl
hpc12	100%	1991	2	51%	991	Linux	2.4.22-1.2115.nptl
hpc13	OFF%	1991	1	-1%	991	Linux	2.4.22-1.2115.nptl
hpc14	0%	1991	1	48%	991	Linux	2.4.22-1.2115.nptl
hpc15	0%	1991	1	47%	991	Linux	2.4.22-1.2115.nptl
hpc16	0%	1991	2	47%	991	Linux	2.4.22-1.2115.nptl

confirm

Fig. 7 Real-time display of all computing nodes status information in our cluster environment

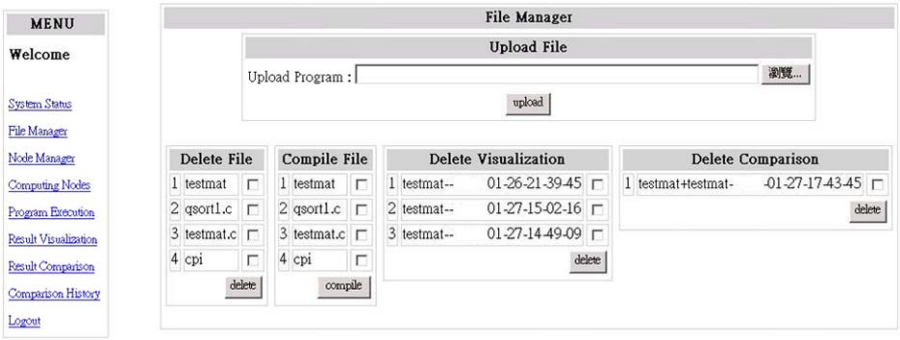

Fig. 8 View of file manager window

Figure 13 shows the MPI parallel program timing graph representation generated by our toolkit as an example of matrix multiplication. The chart shows 4 horizontal bars/lines, where each one corresponds to the execution of one specific computing node. The triangles in red correspond to the communication points in our MPI parallel program (send-receive), while the bars in blue correspond to the execution of sequential codes. Observe that although there are present parallel computations in a

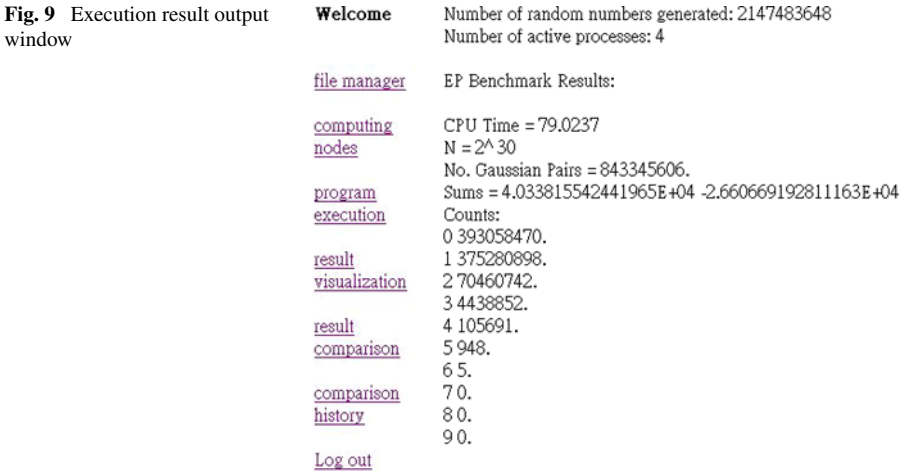

Fig. 9 Execution result output window

Fig. 10 Performance data of computing nodes involved in the computation

computing node, the only longest computation is displayed as one single horizontal blue bar, since these computations are performed inside the same computing node.

4.2 SPICE programs

SPICE is a general purpose circuit simulation program for DC, transient, linear AC, pole-zero, sensitivity, and noise analyses developed by UC Berkeley [22, 27] and written in C. Several commercial codes are based on SPICE. It is used to simulate circuits for various applications from switching power supplies to SRAM cells and sense amplifiers. By doing so, it required the simultaneous solution of a number of equations that capture the behavior of electrical/electronic circuits. The number of equations can be quite large for a modern electronic circuit with transistor counts from several hundred thousands to millions, and thus the simulation of circuits has

Fig. 11 Selections of past execution for performance comparison

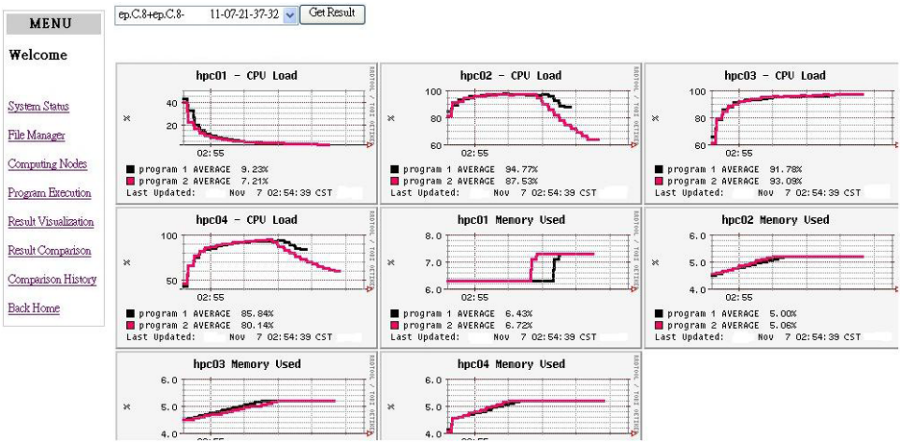

Fig. 12 Performance comparison of 2 execution results, overlapped node by node, showing its CPU load and memory usage

become complex and quite time consuming. Thus, the parallel SPICE program is needed to achieve cost-effective performance. In this paper, we use our toolkit to measure the performance of several versions of SPICE programs. They are sequential, parallel versions using standard OpenMP worksharing directive, and versions using Intel OpenMP *taskq*. We give a basic introduction of each version in the following subsections.

4.2.1 Sequential version

The sequential version of the SPICE program is downloaded and compiled from the original version 3 of UC Berkeley. It provides several types of circuit simulations (or analyses) for modern VLSI design. Among these simulations, the transient simulation

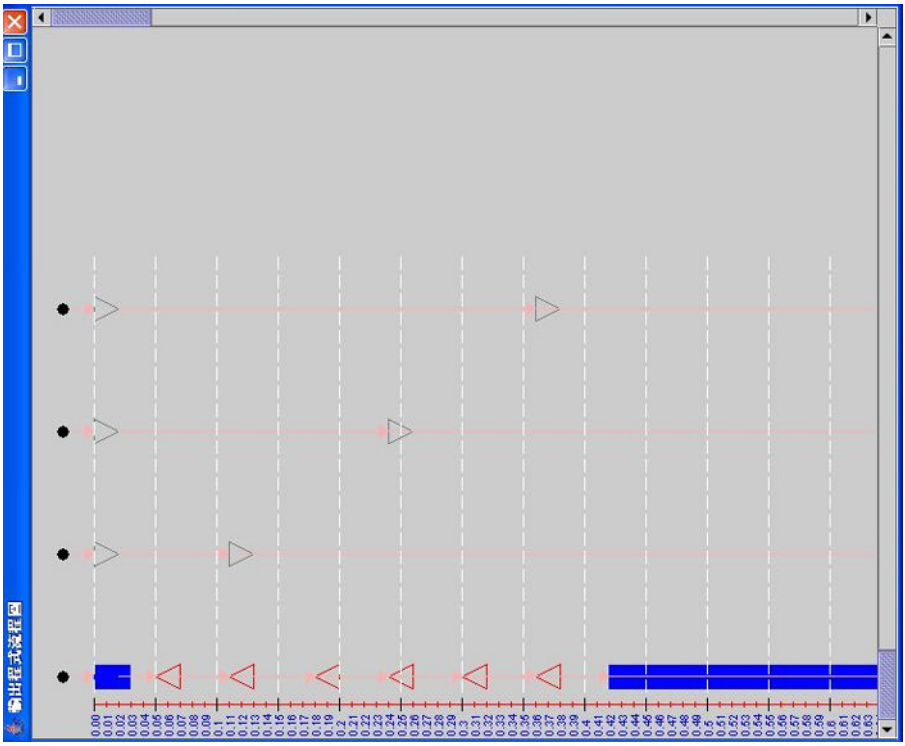

Fig. 13 MPI parallel program representation

is the most frequently used simulation. The circuit netlist describing the connection of the electronic devices in the circuit simulated is first parsed by SPICE and the appropriate data structures are generated. Then the matrix representing the circuit is created and the data structures related to the matrix are set up. Actual transient analysis occurs next. For each time point in the transient analysis, the model calculations for each device, such as MOSFET, resistor, or capacitor device are performed. The electrical parameters such as conductance and current for each instance instantiated from the corresponding device model are computed and put into the matrix elements. After the device model and instance calculations, all elements in the matrix for the linear system in transient analysis are ready for the sparse matrix solver in SPICE. Then the matrix calculations for the linear system, such as the LU decomposition and forward/backward elimination in each iteration, are carried out until the convergence is obtained. This process will continue until the final transient time is reached. Finally, the simulation results for all the time points simulated are displayed on the screen or stored in an output file.

4.2.2 SPICE using standard OpenMP

The parallel SPICE program using the standard OpenMP program is exclusively done on parallelizing the model and the instance calculation part. We refer to it as the device loading routine, because all the model parameters related to the device, and

```
int MOS3load(inModel,ckt)
GENmodel *inModel;
register CKTcircuit *ckt;
{  ......
  register MOS3model *model = (MOS3model *) inModel;
  register MOS3instance *here;
  ......
  MOS3instance **MOS3instanceArray;
  MOS3instanceCount = model->MOS3instanceCount;
  MOS3instanceArray = model->MOS3instanceArray;
#pragma omp parallel default(none) shared(ckt,
  CONSTKoverQ,MOS3instanceCount,MOS3instanceArray)
#pragma omp for private(vt,Check, SenCond,EffectiveLength,DrainSatCur,
  SourceSatCur,GateSourceOverlapCap,GateDrainOverlapCap,\
  GateBulkOverlapCap,Beta,OxideCap,vgs,vds,vbs, vbd,vgb, vgd,xfact, \
  vgdo,delvbs,delvbd,delvgs,delvds,delvgd,cbhat,cdhat,tempv, cdrain, \
capgs,capgd,capgb,von,evbs,evbd,vdsat,cdreq,xrev,xnrm, ceqbd,ceqbs,ceqgb, \
ceq,geq,vgs1,vgd1,vgb1,arg,sarg,sargsw,error,gcgs,ceqgs,gcgd,ceqgd,gcgb,mod
el,here)
for( i = 0; i < MOS3instanceCount; i++) {
here = MOS3instanceArray[i];
model = here->MOS3modPtr;
......
#pragma omp critical(lockA)
{ // Right hand side of Ax = b
*(ckt->CKTrhs + here->MOS3gNode) -= (model->MOS3type * (ceqgs + ceqgb +
ceqgd));
......
// Sum of contributions for the element of matrix A
*(here->MOS3DdPtr) += (here->MOS3drainConductance);
*(here->MOS3GgPtr) += ((gcgd+gcgs+gcgb));
......
} /* end critical */
} /* end of for loop */
return(OK);
} /* end of MOS3load() */
```

Fig. 14 MOS3load function which is part of SPICE3 OpenMP source code

the parameters for the instantiations of the device are computed and loaded into the corresponding matrix elements. There are many devices, such as MOSFET, resistor, capacitor, diode, and bipolar transistor, supported by SPICE. For each device, SPICE provides at least one model for the instances corresponding to this device used in the circuit simulated. For example, MOS3 is one of the models for the instances of MOSFET device. The parameters such as the conductance and current are calculated according to the model equations built into the device loading routines. The conductance calculated will contribute to the elements of the matrix used in the linear system for simulation, while the calculated current will be entered into the right-hand side of the linear system.

The input data we use is SRAM circuit with different size of memories. The SRAM circuit consists of many instances of the MOSFET device with MOS3

model. Therefore, the time-consuming part of the original sequential routine was the *MOS3load* function, which is the device-loading routine in SPICE. It contains a nested pointer loop traversing an orthogonal linked-list. The actual size of the source code of the loop is approximately 1.3 K LOC Line of Code (LOC) and Fig. 14 reproduces the compact example code of *MOS3load* function. The size of iteration is dependent on the size of the circuit, the number of devices such as transistor, capacitor, etc. simulated may vary widely.

There are a few steps to parallelize the sequential nested loop that is shown in Fig. 14. First, at the level of the circuit matrix setup, we introduce a data structure to store the address of each linked-list element of an instance in an array of pointers, *MOS3instanceArray*[*i*], as well as to keep track of the total number of elements in the lists in a variable *model->MOS3instanceCount*. Second, we perform loop coalescing to reduce the number and nesting level of loops, as well as to generate loops with a larger loop iteration count. This loop now involves an array of pointers and integer index instead of pointers. Finally, we may now directly add a parallel omp for directive to the loop since the loop iterations are independent except for first, shared pointers that point to the variables that are used to update the right-hand side of the linear system, $Ax = b$, and second, shared pointers that point to the elements of matrix A, which is used to sum the contributions for those elements. The omp critical synchronization directive is used to resolve this conflict.

Another version is the model device-loading Capacitor load (*CAPload*) function parallelized using OpenMP is shown in Fig. 15. Other model device-loading functions such as diode load (*DIOload*), voltage-source load (*VSRCload*), and many more have a very similar program structure as MOS3load, so they are parallelized in the same way. There is a parallel region within the *MOS3load* routine, which involves a considerable amount of fork-join overheads in addition to the cost of the synchronization overheads. This routine may be called so many times that the number of barrier and critical sections can be large and unavoidable.

4.2.3 SPICE using Intel taskq

In this section, we introduce the version of SPICE implementation using Intel omp taskq. This implementation has been parallelized to perform device instances calculation per thread without modification to the original program. In order to parallelize it without any modification, we have to parallelize at a coarser grain, that is, at the level of device instances of a model (each model consists of many device instances) per task. With this level of granularity (model per thread), the load imbalance problem can occur, since each model may consists of different number of device instances, but then the OpenMP directives can be inserted directly to original program.

Another version that parallelizes CAPload() function using Intel OpenMP task queue is shown in Fig. 17. Both versions of the source code shown in Figs. 16 and 17 have been successfully compiled by an Intel OpenMP compiler on Linux cluster of 2-CPUs machines.

4.2.4 Performance measurement results

We use our toolkit to measure several version of the SPICE3 in its OpenMP implementations. The SRAM circuit consists of many instances of the MOSFET device

```
    ......
/* Count the number of instances */
for(model_temp = model; model_temp != NULL; model_temp = model_temp-
>CAPnextModel )
 for(here_temp = model_temp->CAPinstances; here_temp != NULL ;
here_temp=here_temp->CAPnextInstance)
    instanceCount++;
/* allocate the memory */
CAPinstanceArray = (CAPinstance **) calloc(instanceCount,
sizeof(CAPinstance *));

#pragma omp parallel shared(inModel,ckt)
#pragma intel omp taskq private(model,here,vcap,geq,ceq,cond1,error)
        for(model=inModel ; model != NULL; model = model->CAPnextModel )
#pragma intel omp task
{       /* loop through all the instances of the model */
        for (here = model->CAPinstances; here != NULL ; here=here-
>CAPnextInstance) {
          .....
              for(iparmno=1;iparmno<=info->SENparms;iparmno++){
                 Osxp = tag0 * *(ckt->CKTstate1 + here->CAPsensxp
                             + 2*(iparmno - 1))
                      + tag1 * *(ckt->CKTstate1 + here->CAPsensxp
                             + 2*(iparmno - 1) + 1);
              .....
                 if(iparmno == here->CAPsenParmNo)
                     Osxp = Osxp - tag0 * vcap;
              .....
                 *(info->SEN_RHS[here->CAPposNode] + iparmno) += Osxp;
                 *(info->SEN_RHS[here->CAPnegNode] + iparmno) -= Osxp;
            } } }
    return(OK); }
```

Fig. 15 OpenMP implementation of CAP3load in SPICE

with a MOS3 model. Therefore, the time-consuming part of the original sequential routine was the MOS3load function, which is the device-loading routine in SPICE3. The actual size of the source code of the loop is approximately 1.3 K Line of Code (LOC). The size of iteration is dependent on the size of the circuit, the number of devices such as transistor, capacitor, etc., and simulated may vary widely.

We ran the experiment using our tool, based on different versions of SPICE3 circuit simulator programs simultaneously instead of running sequentially one by one on cluster of 2 CPUs inside computing nodes. The results of execution of all nodes are collected, computed, and plotted. Each version of the programs label "Source," "Capload," "MOS3load," "Capload+MOS3load," and three versions labeled with the beginning word "Taskq." They correspond to an original sequential program of SPICE3, parallelized Capload() function alone, parallelized MOS3load() function alone, parallelized both Capload() and MOS3load() functions using omp parallel for and synchronization directives, three versions each parallelized using omp task queue, respectively. Each version takes input of simulation data of 1, 4, 8, and 16 K SRAM based on Meta Oxide Semiconductor (MOS) 3 level model simulation running on (each node) a DELL PowerEdge SC1420 Intel Xeon Processor 3 GHz Em64T 800 MHz FSB*2 and 1 GB DDR400 memory. Our preliminary experimental results

```
.......
#pragma omp parallel default(none) shared(ckt,
CONSTKoverQ,inModel)
 #pragma intel omp taskq private(vt,Check,
SenCond,EffectiveLength, \
DrainSatCur,SourceSatCur, ........... ,model,here)
    for(model=inModel; model!=NULL;model=model->MOS3nextModel) {
#pragma intel omp task
          for(here=model->MOS3instances; here!=NULL; here=here-
>MOS3nextInstance){
            ...
          #pragma omp critical(lockA)
{     // Right hand side of Ax = b
                  //   Sum of contributions for the element of
matrix A
           }  /* end critical  */
}}
    return(OK);
```

Fig. 16 Parallelized loop of MOS3load() function using Intel omp taskq

```
....
#pragma omp parallel shared(inModel,ckt)
#pragma intel omp taskq private(model,here,vcap,geq,ceq,cond1,error)
       for(model=inModel ; model != NULL; model = model->CAPnextModel )
#pragma intel omp task
{      /* loop through all the instances of the model */
       for (here = model->CAPinstances; here != NULL ; here=here->CAPnextInstance) {
.....
           vcap = *(ckt->CKTrhsOld+here->CAPposNode)
                - *(ckt->CKTrhsOld+here->CAPnegNode) ;
           for(iparmno=1;iparmno<=info->SENparms;iparmno++){
               Osxp = tag0 * *(ckt->CKTstate1 + here->CAPsensxp + 2*(iparmno - 1))
                    + tag1 * *(ckt->CKTstate1 + here->CAPsensxp + 2*(iparmno - 1) + 1);
.....
               if(iparmno == here->CAPsenParmNo)
                   Osxp = Osxp - tag0 * vcap;
.....
               *(info->SEN_RHS[here->CAPposNode] + iparmno) += Osxp;
               *(info->SEN_RHS[here->CAPnegNode] + iparmno) -= Osxp;
           } } }
    return(OK);
```

Fig. 17 Parallelized loop of CAPload() function using Intel omp taskq

using the toolkit in Fig. 18a, b shows the L2 cache read miss and the performance speed up, respectively.

5 Conclusions and future work

To achieve high performance on next generation many-core multiprocessor cluster computing systems, applications developers still face a large number of application performance problems, e.g., load imbalance. These problems make application tuning complex and most of time counter-intuitive. Additionally, similar problems are tough to be detected without the help of performance tools that is able to correlate dynamic performance data sourced from both software and hardware.

We have shown in this paper the viability of implementing a toolkit that brings to application developers representation of his parallel program, easing the hand-made

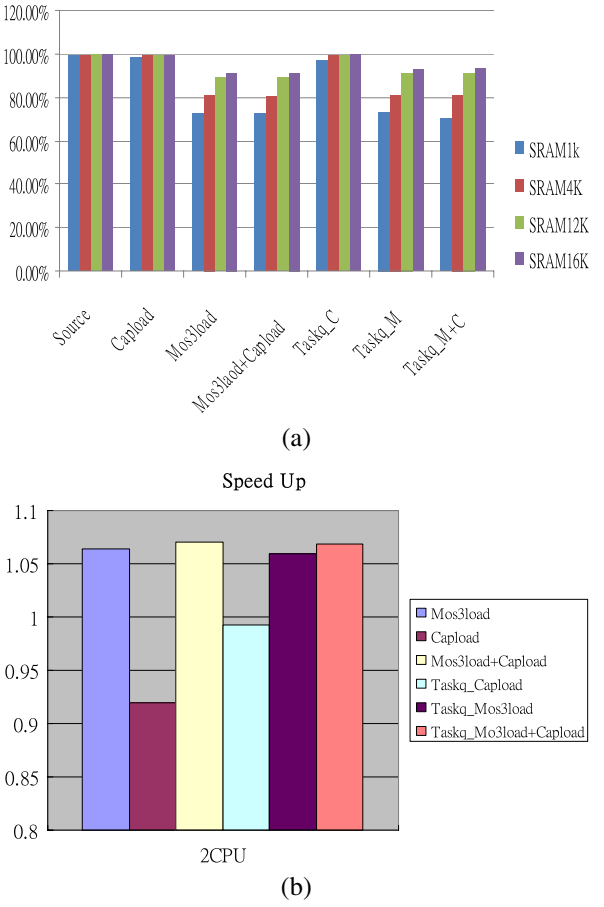

Fig. 18 (**a**) Cache read miss information, and (**b**) performance of SPICE3 application

load balancing issues. In addition, the developer can perform "what-if" analysis on his parallel program by tuning his parallel application to achieve higher performance. This toolkit brings to developers performance data executions of this parallel program under development, by displaying in real-time basis data available of each one computing node, to provide easier execution process of the developers parallel program. Moreover, it provides a friendly interface to developers.

The implementation includes the performance measurement and analysis of hybrid models of MPI/OpenMP in cluster environments. MPI/OpenMP matrix multiplication application examples have been represented by a program timing graph, which may be used for analysis such as load imbalance and communication overheads among nodes. Even though the representation chart in the timing graph only displayed the longest computation among threads within a node, the execution time of each thread can be easily obtained by selecting the bar chart, so that the load imbalance information within a node can be analyzed to locate the region of code where it spent much time waiting while others are doing useful computations. We also present the OpenMP SPICE application for a performance evaluation purpose, showing that

cache behavior for a selected area of code can be obtained from sequential or parallel programs. This information is crucial not only because to its cost-effectiveness in developing parallel applications using MPI/OpenMP, but also to benefit the application developers for performance tuning of their applications.

Overall, not only can application developers and programmers benefit in pursuing a toolkit for performance analysis and tuning their applications, but also its practicality and high aggregated value in using it in research centers, universities that offer courses and training in parallel programming, and other application domains as well.

5.1 Future researches

The proposed research is still in early stages of development and ongoing work, being full-featured to execute sequential and parallel applications. We expect significant improvements for the future process of its development and implementation.

Several directions important to the development of the toolkit can be listed as:

- Computing node selection: possibility to select a range of most suitable computing nodes for a given computation, as discussed in [32]. To create automatic computing node selection function, requesting that web-based toolkit select most suitable computing nodes for a given computation by considering network and CPU computing power factors. Selecting computing nodes for a given computation in heterogeneous environments is not an easy task, with the goal in mind of solving an application in shortest execution time.
- MPI communication primitives: many collective all-to-all constructions have not yet been included in the proposed toolkit yet, the need to also include such symbol in the representation chart when parsed.
- Scheduling policy: to distribute computations among computing nodes in a computing platform efficiently, i.e., to detect the presence of SMPs or multi-core CPUs in computer nodes of a cluster platform. One efficient scheduling method that may be considered is parallel loop self-scheduling for cluster and grid environments [33], or general task scheduling for distributed systems [3, 7, 9, 16].
- Parallelization: to semi-automatically transform existing MPI, OpenMP, or hybrid MPI/OpenMP applications into multithreaded applications, in order to explore higher degree of parallelism and explore many-core SMP multiprocessor cluster systems.

Acknowledgements This paper is based upon work supported in part by the National Science Council (NSC), Taiwan, under grants NSC95-2221-E-126-006-MY3, NSC96-2221-E-126-004-MY3 and NSC96-2218-E-007-007. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSC.

References

1. ANL—Argonne National Laboratory. MPICH: a portable implementation of MPI. Available via http://www-unix.mcs.anl.gov/mpi/mpich1/
2. Anderson TE, Culler DE, Patterson DA (1995) A case for NOW (Network of Workstations). IEEE Micro 15(1):54–64

3. Anik S, Hwu W-W (1992) Executing nested parallel loops on shared-memory multiprocessors. In: Proceedings of the 21st annual international conference on parallel processing (ICPP'92), USA
4. Beguelin A, Dongarra J (1991) Solving computational Grand Challenges using a network of heterogeneous supercomputers. In: Proc of 5th SIAM conference on parallel processing
5. CACTI Tool webpage. Available via http://www.cacti.net/
6. Cain HW, Miller BP, Wylie BJ (2000) A Callgraph-based search strategy for automated performance diagnosis. In: Proc of Euro-Par 2000, Munich, Germany
7. Chan F, Cao J, Chan ATS, Zhang K (2005) Visual programming support for graph-oriented parallel/distributed processing. Softw Pract Exp 35:1409–1439
8. DIMEMAS Tool webpage. Available via http://www.cepba.upc.es/dimemas/
9. El-Rewini H, Lewis TG, Ali HH (1994) Task scheduling in parallel and distributed systems. Prentice Hall, New York
10. Ganglia Cluster Toolkit. Available via http://ganglia.sourceforge.net/
11. Geist G, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V (1994) PVM: Parallel Virtual Machine—a user's guide and tutorial for networked parallel computing. MIT Press, Cambridge
12. Gropp W, Lusk E, Skjellum A (1994) Using MPI: portable parallel programming with the Message Passing Interface. MIT Press, Cambridge
13. Gropp W, Lusk E, Doss N, Skjellum A (1996) A high-performance, portable implementation of the MPI message passing interface standard. Available in Argonne National Laboratory's site at http://www.mcs.anl.gov/mpi/mpicharticle/paper.html
14. Hoeflinger J et al (2001) An integrated performance visualizer for MPI/OpenMP programs. In: Proceedings of WOMPAT'2001 international workshop on OpenMP applications and tools. Lecture notes in computer science, vol 2104. Springer, Berlin, pp 40–52
15. Karypis G, Kumar V (1998) Analysis of multilevel graphs partitioning. Technical Report 98-037, University of Minnesota
16. Kwok YK, Ahmad I (1999) Benchmarking and comparison of the task graph scheduling algorithms. J Parallel Distrib Comput 59:381–422
17. Li K-C, Gaudiot J-L, Sato LM (2002) Performance prediction methodology for parallel programs with MPI in NOW environments. In: Das SK, Bhattacharya S (eds) IWDC'2002 international workshop on distributed computing, Kolkata, India. Lecture notes in computer science, vol 2571. Springer, Heidelberg
18. Li K-C, Chang H-C, Yang C-T, Chang L-J, Cheng H-Y, Lee L-T (2005) Implementation of visual MPI parallel program performance analysis tool for cluster environments. In: AINA'2005 The 19th IEEE international conference on advanced information networking and applications, Taiwan
19. Liu LT, Culler D, Yoshikawa C (1996) Benchmarking message passing performance using MPI. In: Proceedings of ICPP'1996 international conference on parallel processing. IEEE Comput. Soc., Los Alamitos, pp 101–110
20. Lisper B (2003) Fully automatic, parametric worst-case execution time analysis. In: Gustafsson J (ed) Proceedings of the third international workshop on worst-case execution time (WCET) analysis, Porto, Portugal, pp 77–80
21. MRTG webpage (2007) Available via http://www.mrtg.org
22. Nagel LW (1975) SPICE2—A Computer program to simulate semiconductor circuits. Memo ERL-M520, University of California, Berkeley, ERL
23. NWS Information Service (2007) Available at http://nws.cs.ucsb.edu/ewiki/
24. OpenMP webpage (2007) Available via http://www.openmp.org
25. Pthreads tutorial webpage (2008) Available via https://computing.llnl.gov/?set=training&page=index
26. Puschner P, Schedl A (1997) Computing maximum task execution times—a graph-based approach. J Real-Time Syst 13(1):67–91
27. Quarles TL (1989) Analysis of performance and convergence issues for circuit simulation. Memo ERL-M89, University of California, Berkeley, ERL
28. RRD tool webpage. Available via http://www.rrdtool.org
29. Smith L, Bull M (2000) Development of Mixed Mode MPI/OpenMP Applications. In: Proc of the workshop on OpenMP applications and tools (WOMPAT2000)
30. SUN Microsystems. JAVA2 Second Edition (J2SE). http://java.sun.com
31. VAMPIR tool. Pallas Products webpage. Available via http://www.vampir.eu
32. Xu Q, Subhlok J (2005) Automatic clustering of grid nodes. In: Proceedings of the 6th IEEE/ACM international workshop on grid computing, USA

33. Yang C-T, Cheng K-W, Li K-C (2004) An efficient parallel loop self-scheduling on grid computing environments. In: Jin H, Gao G, Xu Z, Chen H (eds) NPC'2004 IFIP international conference on network and parallel computing. Lecture notes in computer science, vol 3222. Springer, Berlin

Kuan-Ching Li received the Ph.D. and M.S. degrees in Electrical Engineering and Licenciatura in Mathematics from the University of São Paulo, Brazil in 2001, 1996 and 1994, respectively. After he received his Ph.D., he had been a post-doc scholar in the Univ. of California—Irvine (UCI) and Univ. of Southern California (USC), USA. He is currently Associate Professor in the Department of Computer Science and Information Engineering, Providence University, Taiwan. He has served on the steering, organizing, and program committees of several conferences and symposiums. His main research interests include cluster and grid computing, parallel software design, and life science applications. He is a senior member of the IEEE.

Tien-Hsiung Weng is an Associate Professor at the Department of Computer Science and Information Engineering at Providence University, Taiwan. He received his Ph.D. in Computer Science from University of Houston, Texas, USA. His current research focuses on parallel programming model, performance measurement, and compiler analysis for code improvement and parallel programming.