International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:1, 2009

A Survey on Performance Tools for OpenMP

Mubrak S. Mohsen, Rosni Abdullah, and Yong M. Teo

Abstract**—**Advances in processors architecture, such as multi-core, increase the size of complexity of parallel computer systems. With multi-core architecture there are different parallel languages that can be used to run parallel programs. One of these languages is OpenMP which embedded in C/C++ or FORTRAN. Because of this new architecture and the complexity, it is very important to evaluate the performance of OpenMP constructs, kernels, and application program on multi-core systems. Performance is the activity of collecting the information about the execution characteristics of a program. Performance tools consists of at least three interfacing software layers, including instrumentation, measurement, and analysis. The instrumentation layer defines the measured performance events. The measurement layer determines what performance event is actually captured and how it is measured by the tool. The analysis layer processes the performance data and summarizes it into a form that can be displayed in performance tools. In this paper, a number of OpenMP performance tools are surveyed, explaining how each is used to collect, analyse, and display data collection.

Keywords**—**Parallel performance tools, OpenMP, multi-core.

I. Introduction

THE development in processors architecture, such as multi-core, contributed to broaden the complexity of parallel computer systems[1]. Within the multi-core architecture there are varieties of parallel languages that can be utilized to run the parallel programs. The OpenMP is one of the most important libraries that could be applied successfully to run the parallel applications. The OpenMP is embedded either in C/C++ or FORTRAN. Comparing with other parallel languages, OpenMP shows efficacy when it has been used with multi-core architecture processors.

For the last few years, multi-core architectures become most globally distributed. In coincidence with development of such processors generation, there is a necessity for further understanding about the OpenMP nature, performance and environment. That will lead to find the proper clues for certain problems, such as load balancing, bottleneck, tuning, debugging, and performance. Unfortunately no standardized performance analysis interface exists for OpenMP yet, making OpenMP performance analysis dependant on platform and compiler specific mechanism. There are many tools that have been recommended to evaluate the OpenMP performance; for example, TAU[2], KOJAK[3], ompP[4],…, etc. With these tools questions occur regarding how performance instrumentation and measurement are collected and how performance data is analyzed and mapped to the language-level parallel abstractions.

This paper is a survey for OpenMP performance tools that have been used. The reminder of this paper is organized as follows: section II contains a description of the OpenMP. Section III classifies the performance tools evaluation. Section IV presents a detail explanation of different performance tools and how they are used to measure the performance of OpenMP. Finally, section V presents our discussion and conclusion.

II. OPENMP

OpenMP[5] is An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. The OpenMP Architecture Review Board (ARB) published OpenMP 1.0 for FORTRAN in October, 1997. They released OpenMP for the C/C++ standard in 1998. Version 2.0 of the FORTRAN specifications is released in 2000. Version 2.0 of the C/C++ specifications is being released in 2002. Version 2.5 is a combined C/C++/FORTRAN specification released in 2005. Version 3.0 is released in May, 2008.

OpenMP uses the fork-and-join parallelism model. In fork-and-join, parallel threads are created and branched out from a master thread to execute an operation and will only remain until the operation has finished, then all the threads are destroyed, thus leaving only one master thread. Because of this parallelism OpenMP has became useful for multi-core programs. OpenMP compiler is not required to check for data dependencies, data conflicts, race conditions, or deadlocks, which may occur in programs. The user is responsible to check that the programs are free from those runtime errors.

III. Performance Tools Classification

Performance tools can be categorized based on the run time behavior into two main categories; on-line and off-line (Fig. 1). However, based on the capture and analysis approaches the tools can be classified into three categories; profiling, tracing, and profiling & tracing (Fig. 2).

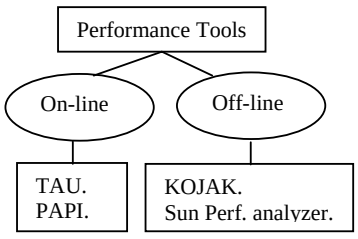

Fig. 1 On-line and off-line performance analysis tools

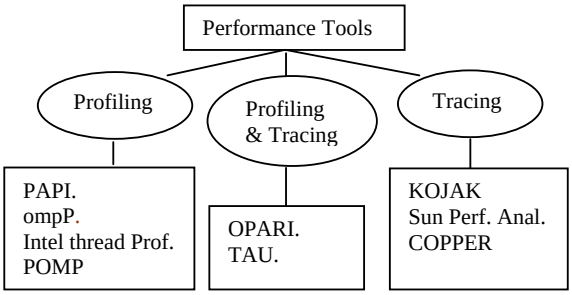

Fig. 2 Profiling and tracing performance tools

This section describes the performance tools classification with details about how it is used to collect, measure, analyse, and visualize the performance data. This section is organized as follows: subsection 1 describes the on-line and off-line performance evaluation. subsection 2 describes the profiling and tracing performance evaluation.

1. On-line and off-line Performance Evaluation

The main difference between on-line and off-line analysis approach is how the data are collected, analyzed, and displayed. The On-line analysis tool displays an on-going analysis while the program is executing, without producing trace files. It consists of two phases. The first phase instruments the program to collect information during the execution. This information is not collected for later use, but it is analyzed as execution progresses. The performance data are reduced or filtered in the memory. The performance data can be updated and accessed quickly during execution. It also can be discarded as they become obsolete. The only purpose of filtering/reduction is to reduce the space overhead;

However, off-line analysis tool display the performance data after the execution of captured data has finished. This process includes three phases. The first phase instruments the program to record the performance data during execution for each event. In the second phase, the instrumented version of the program is executed, writing this information to trace files. After the execution is completed, the final phase is to analyze and display the performance data.

In terms of comparison, the On-line method has an advantage over the off-line method that large trace files are not generated. Meanwhile, the disadvantage of the on-line method is incurring more space overhead during execution, not detecting some events data, and discarding some information.

The motivation for on-line performance monitoring are many: it is suitable for long-running programs or server processes that are not meant to terminate; it does not have to contend with a huge volume of performance data as in event-tracing as performance analysis occurs concurrently with program execution; and it highlights the performance data for the current execution state of the application which can be related to other system parameters (e.g., disk activity, CPU and memory utilization, load on the system, and network contention). Examples of on-line performance evaluation tools include the program monitor used by TAU [2] for on-line-monitoring of profile data.

Off-line tools do not analyze or display the performance data while it is collected. They store the data for later analysis. Off-line tools help the user characterize the behavior of the application by analyzing the results of performance data after the application terminates.

2. Profiling and Tracing Performance Evaluation

Based on the measurement approach, the performance tools for parallel programs fall into three categories: profiling, event-tracing and mixed profiling and tracing.

Tracing approach has the advantage that they can capture all the relevant events describing the runtime behavior of the application, but the overhead of tracing can be substantial. Traces scale poorly with increasing runtime, with no obvious mechanism of throttling the data volume, and without capturing all function call and return points, the dynamic program graph can not be generated. Many implementations of tracing methodologies have been developed, along with a rich set of data-reduction and visualization tools for example, KOJAK[3], CATCH[6], TAU[2], and VAMPIR[7]. One tracing methodology, POMP, was proposed as a standard for OpenMP performance measurement, but it was not adopted.

On the other hand, statistical profiles have the advantage that they can scale very nicely to long-running programs, simply by throttling the profiling rate. By capturing complete callstacks with each event, the dynamic callgraph behavior becomes obvious. Capturing callstacks for user-model OpenMP programs is a challenge. Statistical profiling has one major disadvantage: it is statistical in nature, and may miss some rare events; furthermore, if the application behavior is correlated with the profiling clock, the measurements may be ambiguous.

Finally, profiling and tracing tools are tried to combine the advantages of tracing and profiling together.

2.1 Profiling Approach

Profiling shows the summary statistics of performance metrics that characterize application performance behavior. Examples of performance metrics are the CPU time associated with a routine, the count of the secondary data cache misses associated with a group of statements, the number of times a routine executes, etc.

Profiling are tried to characterize the behavior of an application in terms of aggregate performance metrics. Profiles are typically represented as a list of various metrics (such as wall-clock time) that are associated with program-level semantic entities (such as routines or statements in the program). Time is a common metric used but any monotonically increasing resource function can be used. The two main approaches to profiling include sampling-based profiling and instrumentation-based profiling.

2.1.1 Sampling-based Profiling Approach

Sampling-based profiling periodically records the program state and, based on measurements made on those states, estimates the overall performance. Profiling tools use a

hardware interval timer that generates a periodic interrupt after a certain amount of time elapses. At that interrupt, the process state is sampled and recorded. Based on the total number of samples recorded, the interval between the interrupts and the number of samples recorded when a particular code segment is executing; statistical techniques are employed to generate an estimate of the relative distribution of a quantity over the entire program. Since events that occur between the interrupts are not seen by the performance tool, the accuracy of this approach depends on the length of the interval which typically varies from 1 to 30 milliseconds. Increasing the interval reduces the fixed measurement overhead of profiling, but decreases its resolution. As processor speeds continue to increase, however, sampling based on time can lead to sampling errors and inaccuracies. An alternative, as implemented in the unix profiling tool, is sampling based on the number of elapsed instructions between two interrupts. This removes the dependency on the clock speed.
Some advantages of the sampling are:

- Profiling is to find hotspots, the module, functions, lines of source code and assembly instructions that are consuming the most time
- Low overhead. Overhead incurred by sampling is typically about one percent
- No need to instrument code. There is no need to make any changes to code in order to profile with sampling.

In sampling-based profiling, either the program counter (thus the currently executing routine) is sampled or the callstack is sampled. In the first case, the program counter is translated to the currently executing block of code and the number of samples associated with that block of code is incremented. In the second case, the callstack is sampled, and the time for the block of executing code is incremented. This allows the computation of both exclusive and inclusive time. Exclusive time is the time spent executing the given routine not including time spent on other routines that it called. Inclusive time includes the contributions from callees computed by traversing the callstack. Based on the number of samples taken in a given block, the time spent in the block is estimated using the sampling frequency and the processor clock frequency.

Some tools show the caller-callee relationship using call graphs instead of flat profiles. Based on the number of times a routine (parent) invokes another routine (child), it divides the time spent in a routine among its callers in proportion to the number of times the routine was called by them. It reports this call-graph information to one level. While this may be misleading in cases where the different invocations of the routines perform unequal work, it highlights the importance of callgraphs.

Other tools use call paths, or sets of stack traces (sequences of functions that denote a thread's calling order at a given instant of time) to present performance in terms of call sequences between routines.

Although, sampling-based schemes suffer from incomplete coverage of the application (especially for applications that have a short life-span), they have a distinct advantage of fixed, low instrumentation overhead and consequently reduced measurement perturbation in the program.

Another alternative approach is to use the hardware performance monitors provided by most modern microprocessors. Hardware performance monitors can be used both as interval markers and as sampled data. They are implemented as on-chip registers that can transparently count quantities, such as the number of instructions issued, cycles completed, floating point operations performed, number of data and instruction cache misses seen. Most microprocessors provide two or three registers (which implies that two or three counters can be used simultaneously), a mechanism to read and reset the registers, and an interrupt capability when a counter overflows. Tools such as Intel's VTune[8] use interrupt intervals based on the number of instructions issued.

2.1.2 Instrumentation-Based Profiling Approach

In order to observe performance, additional instructions or probes are typically inserted into a program. This process is called instrumentation. The execution of a program is regarded as a sequence of significant events. As events execute, they activate the probes which perform measurements. Thus, instrumentation exposes key characteristics of an execution. Performance evaluation tools present this information in the form of performance metrics. Performance bottleneck detection tools go one step further and automate the process of identifying the cause of poor performance.

With instrumentation-based profiling, measurements are triggered by the execution of instructions added to the code to track significant events in the program such as the entry or exit of a routine, the execution of a basic block or statement, and the send or receipt of a message communication operation. Typically, such profilers present the cost of executing different routines in a program. Tools insert instrumentation code during different phases of compilation and execution.

Table I, shows the comparison of sampling and instrumentation profiling. Each method has some advantages than the other does not have. While the sampling does not incur overhead, it does not track all events in the application**.**

TABLE I
SAMPLING AND INSTRUMENTATION COMPARISON

	Sampling	Instrumentation
Overhead	low	High
System-wide profiling	Yes, profiles all application, drivers, OS functions	Just application and instrumented DLLs
Detect unexpected events	Yes , can detect other programs using OS resources	No
Data collected	Counters, processor an OS state	Call graph , call times, critical path
Data granularity	Assembly level instr., with source line	Functions, sometimes statements

2.2 Event Tracing Approach

While profiling is used to get aggregate summaries of metrics in a compact form, it can't highlight the time varying aspect of the execution. Event tracing is more appropriate to study the spatial and temporal aspect of performance data. Event tracing is the activity of capturing an event or an action that takes place in the program. Event tracing usually results in a log of the events that characterize the execution. Each event in the log is an ordered tuple typically containing a time stamp, a location (e.g., node, thread), an identifier that specifies the type of event (e.g., routine transition, user-defined event, message communication, etc.) and event-specific information. Event tracing is commonly employed in debugging and performance analysis.

Instrumentation can perturb an application and modify its behavior. Event tracing is the most invasive form of instrumentation because the volume of data generated is quite large and thus it may perturb the application more than other forms of performance measurement

In a parallel execution, trace information generated on different processors must be merged. This is usually based on the time-stamp which can reflect logical time or physical time. The logical time uses local counters for each process incremented when a local event takes place. The physical time uses reference time obtained from a common clock, usually a globally synchronized real-time clock.

Trace buffers keep the ordered and merged logs. They can be shared or private. On shared memory multiprocessors, multiple threads can easily access a shared trace buffer providing an implicit ordering and merging of trace records and maximizing the buffer memory utilization. Private buffers can be used in both shared memory and distributed memory systems. Since the buffer is private, there is no contention for it, but the trace buffer memory is not optimally utilized due to varying rates of event generation in tasks.

In either case, the trace buffer needs to be periodically flushed to disk. There are two policies used in flushing the buffer to the disk: Flush one buffer when it fills (FOF) and Flush All the buffers when One Fills (FAOF).

IV. PERFORMANCE TOOLS

The basic purpose of application performance tools, are to help the user identify whether or not their application is running efficiently on the computing resources available. The benefits of using the tool can be to decrease execution time, to increase efficiency of resource utilization, or a combination of the two. Many software tools are provided to measure the performance. These performance tools typically rely on at least three layers, which consist of: instrumentation, measurement, and analysis. The instrumentation layer defines the measured performance events. The measurement layer determines what performance event is actually captured and how it is measured by the tool. The analysis layer processes the performance data and summarizes it into a form that can be displayed in performance tools.

We present a survey of various tools that can be used to aid in performance analysis of OpenMP programs. This section explains in detail the different software tools according to their classification. This section is organized as follows: section 1 describes the profiling-based performance tools. Section 2 describes the tracing-based performance tools. Finally, section 3 describes the profiling&tracing based performance tools.

1. Profiling-Based Performance Tools

This section describes in detail the performance tools that based on profiling, and shows how they work to measure the performance. PAPI, POMP, ompP, and Intel Thread Profiler are designed based on profiling approach.

1.1 Performance Application Programming Interface (PAPI)

The purpose of the PAPI [9] project is to design, implement a portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors. PAPI provides two interfaces: high level interface for the acquisition of simple measurements, and low level interface. The low level interface manages hardware events in user defined groups called EventSet. The high level interface simply provides the ability to start, stop and read the counters for a specified list of events. The different features of the low-level interface give the designer much more flexibility.

PAPI also defines a set of hardware performance events. Both the high and low level interface along with the event definitions together support standardized control and access to hardware counters under most hardware architectures

PAPI supports a number of platforms (e.g., Cray T3E, SGI IRIX, IBM AIX Power, Sun Ultrasparc Solaris, Linux/x86, Linux/IA-64, HP/Compaq Alpha Tru64 Unix). The implementation for a given platform attempts to map as many of the standard PAPI events as possible to the available platform-specific events. The implementation also attempts to use available hardware and operating system support. e.g., for counter multiplexing, interrupt on counter overflow, and statistical profiling. Multiplexing means PAPI allows more counter to be counted than can be supported by the hardware.

The dynaprof tool developed as part of the PAPI project uses dynamic instrumentation to allow the user to either load an executable or attach to a running executable and then dynamically insert instrumentation probes [10]. Dynaprof provides a PAPI probe for collecting hardware counter data and a wallclock probe for measuring elapsed time, both on a per-thread basis. Users may optionally write their own probes. A probe may use whatever output format is appropriate, for example a real-time data feed to a visualization tool or a static data file dumped to disk at the end of the run.

The PAPI project has developed two tools that show graphical display of PAPI performance data [9]. The first tool, called the *perfometer*, provides a runtime trace of a chosen PAPI metric. The second tool, called the *profometer*, provides a histogram that relates the occurrences of a chosen PAPI event to text addresses in the program. PAPI can integrate with Visual Profiler, SvPablo, and DEEP for collecting data and graphically viewing the results.

PAPI isn't developing the capability of relating the frequency of events to source code locations, so that developer can't locate portions of the program that are the source of performance problems.

1.2 POMP

Different performance monitoring interfaces have been proposed to the OpenMP Architecture Review Board (ARB) to include in the specifications of the language. The POMP[11] interface was proposed and was not supported because it was highly dependent on source level instrumentation and the programming model was somewhat complicated. The primary goal is to define a clear and portable API for OpenMP that makes execution events visible to runtime monitoring tools, primarily tools for performance measurement. POMP is an interface that enables performance tools to detect OpenMP events. POMP supports measurements of OpenMP constructs, OpenMP API calls and user functions/regions. POMP presents a uniform interface for providing such a mapping between the performance data and OpenMP events.

The POMP interface for OpenMP provides a performance API target for source to source instrumentation tools [12] (e.g., OPARI) allowing for instrumented OpenMP codes that are portable across compilers and machine platforms. Defined as a library API, the interface exposes OpenMP execution events of interest (e.g., sequential, parallel, and synchronization events) for performance observation, and passes OpenMP context descriptors to inform the performance interface library of region-specific information. The OPARI tool rewrites OpenMP directives in functionally equivalent, but source instrumented forms, inserting POMP performance calls where appropriate.

The advantages of POMP includes the following: the interface does not restrict the implementation of OpenMP compilers or runtime systems, it is compatible with other performance monitoring interfaces such as PMPI, and it permits multiple instrumentation methods (e.g. source, compiler, or runtime).

POMP is quite complicated and incur too much overhead because of these disadvantage[13]: First, POMP requires that a data structure storing source level information and IDs are passed as a parameter for each POMP call that can be quite expensive in terms of overhead. Second, POMP was designed for source level instrumentation. Instrumentation at this level can interfere with compiler optimizations. Third, POMP requires modification of the code that interferes with the execution of the original code.

1.3 A profiling tool for OpenMP (ompP)

ompP[4] is a simple text-based profiling tool for OpenMP applications written in C/C++ or FORTRAN. ompP supports the measurement of hardware performance counters using PAPI [9] and supports some advanced features such as overhead analysis and detection of common inefficiency situations (performance properties). ompP relies on OPARI[14] for source-to-source instrumentation.

ompP is a profiling tool for OpenMP applications designed for Unix-like systems. ompP differs from other profiling tools. First, ompP is a measurement based profiler and does not use program counter sampling. The instrumented application invokes ompP monitoring routines that enable a direct observation of program execution events (like entering or exiting a critical section). An advantage of the direct approach is that its results are not subject to sampling inaccuracy and hence they can also be used for correctness testing in certain contexts. The second difference is in the way of data collection and representation. While other profilers work on the level of functions, ompP collects and displays performance data in the OpenMP user model of the execution. For example, the data reported for critical section contains not only the execution time but also lists the time to enter and exit the critical construct (enterT and exitT, respectively) as well as the accumulated time each threads spends inside the critical construct (bodyT) and the number of times each thread enters the construct (execC).

Furthermore, ompP supports querying hardware performance counters through PAPI and the measured counter values appear as additional columns in the profiles. In addition to OpenMP constructs that are instrumented automatically using OPARI, a user can mark arbitrary source code regions such as functions or program phases using a manual instrumentation mechanism. Profiling data is reported by ompP both as flat profiles as well as callgraph profiles, giving inclusive and exclusive times in the latter case. ompP performs an overhead analysis where four well-defined overhead classes (synchronization, load imbalance, thread management, limited parallelism) are quantitatively evaluated. ompP also tries to detect common inefficiency situations, such as load imbalance in parallel loops, contention for locks and critical sections, etc. The profiling report contains a list of the discovered instances of these – so called – performance properties sorted by their severity (negative impact on performance).

Performance data is collected on a region stack basis. Stack of entered OPARI regions is maintained and data is attributed

to the stack that leads to a certain region. This region stack is currently maintained for POMP regions only, i.e., only automatically instrumented OpenMP constructs and user-instrumented regions are placed on the stack, general functions or procedures are not handled unless manually instrumented by the user. Hardware counters can be used with ompP, the number of hardware counters that can be recorded simultaneously by ompP is a compile time constant set to 4 per default.

The performance data collected by ompP is kept in memory and written to a report file when the program finishes. Two formats are specified plaintext ASCII and comma separated values (CSV).

The report shows the different regions view, it shows the execution of the application in callgraph or tree, flat region profiles, and callgraph region profiles. It also shows the overhead analysis which is four OpenMP related categories (imbalance, synchronization, limited parallelism, and thread management). In additional it shows the performance properties which capture the common situation of inefficient execution.

The tool can be used to quickly identify regions of inefficient behavior. In fact by analyzing execution counts the tool is also useful for correctness debugging in certain cases.

Another benefit is the immediate availability of the textual profiling report after the program run, as no further post-processing step is required. Furthermore the tool is naturally very portable and can be used on virtually any platform making it straightforward to compare the performance on a number of different platforms.

The limitation of ompP is no visual representation of profiling data. Furthermore, the user-defined functions are not instrumented unless the user manually instruments the functions.

ompP support Linux/x86, IA64, ALX, Linux platforms, with gcc, intel, IBM PHI compilers.

1.4 Intel Thread Profiler

The Thread Profiler[15] is a plug-in to the VTune Performance Analyzer[8]. The Thread Profiler supports applications threaded with OpenMP, Windows API, or POSIX threads (Pthreads). Thread Profiler is used to identify bottlenecks that limit the parallel performance of your multi-threaded application. It used to find the best sections of code to optimize for sequential performance and for threaded performance and compare scalability across different numbers of processors or using different threading methods. It is used also to locate synchronization delays, stalled threads, excessive blocking time, and ineffective utilization of processors.

The Thread Profiler enables to create activities in the VTune Performance Environment. Activities are at the core of the data-collection process in the VTune analyzer. Within an Activity, user can specify the types of performance data wish to collect. For each type of performance data, user needs to configure the appropriate data collector and define the application/module profiles. These profiles contain information about the application to execute and the modules to analyze. Running an activity is analogous to running an experiment. When the user's activity is run, the data collectors collect performance data and save it in the Activity results. The result can be viewed later, without having to rerun the activity. User can also drag and drop different activity results into a view to compare data from different experiments.

In addition to sampling, the tool also provides call graph monitoring. The gathered data can be visualized, and the tool also provides code tuning advice. VTune is a Windows tool with limited Linux support. Also a commercial license is required.

When using Thread Profiler to analyze applications on systems with more than 32 processors, the overhead may become excessive. Intel Thread Profiler supports analysis of code using one of the following collection modes: Instrumented and OpenMP-specific.

By using one or a combination of Binary Instrumentation and Source Instrumentation to prepare the application for data collection with the Intel(R) Thread Profiler, depending on the needs of the program.

Intel Thread Profiler is very useful for analyzing bottlenecks in threaded code. Thread Profiler quickly pinpoints problem areas and shows the reasons for the slowdown, to restructure the code for better threaded performance. Intel Thread Profiler shows the timeline to understanding what threads are doing and how they interact. It Pinpoints the exact location of performance issues in call stacks and source code to aid analysis. It measures the number of cores that are effectively utilized by the application to determine actual parallel performance. It is analysis the OpenMP and estimate of the performance potential of different designs. It's used critical path analysis to help focus on more significant performance issues.

The latest release supports C++ applications developed using the Microsoft Windows compilers in Microsoft Visual Studio 2005 and 2008. It supports the latest multi-core processors on the new Intel Core 2 Duo and Intel Core 2 Quad processors.

Intel profile display the Critical Path View which illustrates some of the basic techniques of concurrency level and thread interactions. Intel profile also displays the profile view of the selected critical path data loaded. The Profile View allows user to group, filter and sort the data in order to get a better understanding about the performance of the threads within the application.

It also shows Timeline View. The Timeline View is able to display the motion (how the critical path passed from one thread to another) of the critical path across threads. Keeping track of these transitions, however, is an expensive operation and will increase memory usage and cause a noticeable slow down in execution time within Thread Profiler.

Intel profiler can display Source Views. This display allows the programmer to judge how those objects could be used more effectively.

Intel Thread Profiler monitors execution of applications to detect threading performance issues, including thread

overhead and synchronization impact. Intel Thread Profiler provides results in graphical displays to help quickly pinpoint the locations in code that directly affect execution time where fewer than the optimal number of threads were executing, identify synchronization objects that may be impacting thread execution, and ascertain load balance issues between threads.

2. *Tracing-Based Performance Tools*

This section describes in detail the performance tools that based on event tracing, and shows how they work to measure the performance. KOJAK, Sun Performance Analyser, and Copper are designed based on event trace approach.

2.1 Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks (KOJAK)

The KOJAK project [3] is a tool used to measure the performance analysis environment for parallel programs.

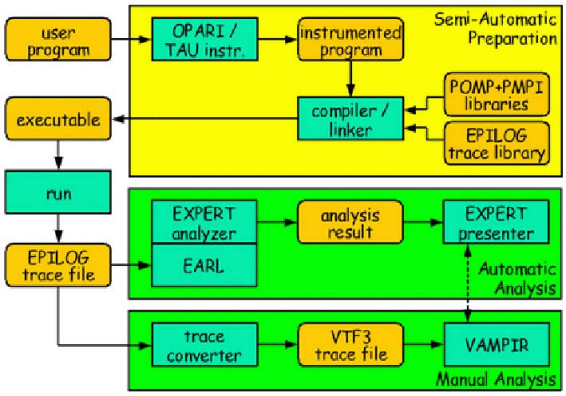

Fig. 3 Kojak Architecture [3]

An overview about the architecture of the Kojak prototype and its components are given in Fig. 3. The analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data. The first subprocess is called semi-automatic because it requires the user to slightly modify the makefile and execute the application manually.

To begin the process, the user supplies the application's source code, written in C, C++, or Fortran, to OPARI [14], which performs automatic instrumentation of OpenMP constructs and redirection of OpenMP-library calls to instrumented wrapper functions on the source-code level based on the POMP API. Instrumentation of user functions is done either on the source-code level using TAU or using a compiler-supplied profiling interface. Instrumentation for MPI events is accomplished using a PMPI wrapper library, which generates MPI-specific events by intercepting calls to MPI functions. All MPI, OpenMP, and user-function instrumentation call the EPILOG (Event Processing, Investigating and LOGging) run-time library, which provides mechanisms for buffering and trace-file creation. At the end of the instrumentation process the user has a fully instrumented executable.

Running this executable generates a trace file in the EPILOG format. After program termination, the trace file is fed into the EXPERT (Extensible Performance Tool) analyzer. The analyzer uses EARL (Event Analysis and Recognition Language) to provide a high-level view of the raw trace file. We call this view the enhanced event model, and it is where the actual analysis takes place. The analyzer generates an analysis report, which serves as input for the EXPERT presenter. In addition, it is possible to convert EPILOG traces to VTF3 format and analyze them manually with the VAMPIR event trace analysis tool (version 3.X or later only).

KOJAK support Linux on IA32, IA64, X86_64/EMT6, Cray XT3 platform with GNU, Intel, PGI, Sun, and IBM compilers.

2.2 Sun Performance Analyzer

The Sun Performance Analyzer[16] supports OpenMP, MPI and hybrid MPI/OpenMP. Special efforts have been made to allow for the analysis of OpenMP programs. At runtime, the performance data collection module communicates with the OpenMP runtime library in order to obtain specific information about the state of all OpenMP threads.

The latest Sun Studio release supports the SPARC and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, Pentium, and Xeon EM64T. The Performance Analyzer displays the raw data in a graphical format as a function of time.

The tool consists of collector and analyzer. The Collector tool collects performance data using a statistical method called profiling and by tracing function's calls. The data can include call stacks, microstate accounting information, thread synchronization delay data, hardware counter overflow data, Message Passing Interface (MPI) function call data, memory allocation data, and summary information for the operating system and the process. The Collector can collect all kinds of data for C, C++ and FORTRAN programs, and it can collect profiling data for applications written in the Java programming language. It can collect data for dynamically generated functions and for descendant processes. The Collector collects three different kinds of data: profiling data, tracing data and global data. Both profiling data and tracing data contain information about specific events, and both types of data are converted into performance metrics. Global data is not converted into metrics, but is used to provide markers that can be used to divide the program execution into time segments. The global data gives an overview of the program execution during that time segment. Metrics are assigned to program instructions using the call stack that is recorded with the event-specific data. If the information is available, each instruction is mapped to a line of source code and the metrics assigned to that instruction are also assigned to the line of source code.

The Performance Analyzer reads the event data that is collected by the Collector and converts it into performance metrics. The metrics are computed for various elements in the structure of the target program. The Performance Analyzer processes the data and displays various metrics of

performance at the level of the program, the functions, the source lines, and the instructions. Then, it displays the data in tabular and graphical displays.

2.3 COPPER

COPPER [17] is integration of OpenUH compiler and KOJAK [3] for collaborative application tuning. The process starts with the instrumentation of OpenMP constructs on the source-code level using a preprocessor called OPARI [14]. In the next step, the application is compiled by OpenUH and, at the same time, the compiler inserts instrumentation into the user code to generate traces for KOJAK. After application termination, KOJAK analyzes the resulting trace file and provides higher-level feedback that is returned to OpenUH's feedback optimization module.

In the COPPER environment as in Fig. 4, user regions are automatically instrumented by the OpenUH compiler. The OpenUH compiler supports the instrumentation of function, loop, conditional branch, and compare and goto program units. OPARI performs automatic instrumentation of OpenMP constructs according to the POMP profiling interface for OpenMP.

At runtime, the instrumented executable generates a single trace file that can be searched off-line for inefficiency patterns using the EXPERT analyzer[18].

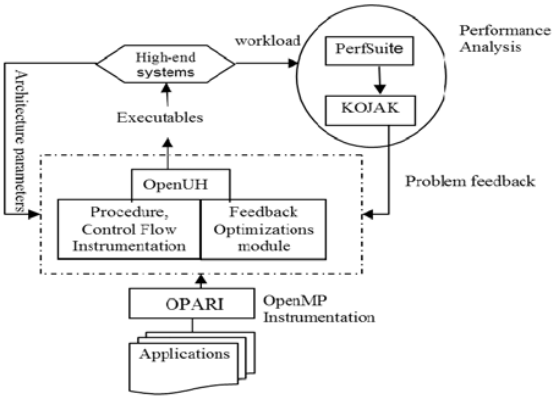

Fig. 4 COPPER architecture [17]

When reading a parallel program containing OpenMP directives, KOJAK automatically invokes OPARI to insert POMP performance calls where appropriate. As a final step of the instrumentation, KOJAK links the application with a library implementing the POMP API to generate appropriate events and write them to the trace buffer. Thus, with the help of the PMPI library and the OpenUH user-region instrumentation, their approach provides a fully automatic solution to the instrumentation of OpenMP and mixed-mode MPI/OpenMP applications.

The analysis process transforms the traces into a compact XML representation that maps higher-level performance problems onto the call tree and the hierarchy of system resources, such as nodes, processes, and threads. The XML file can be viewed in the CUBE performance browser or, alternatively, automatically processed by third-party tools using the CUBE API.

3. Profiling &Tracing Based Tools

This section describes in detail the performance tools that based on profiling and tracing. It shows how these tools work to measure the performance. OPARI, and TAU are designed based on profiling and tracing approaches.

3.1 OpenMP Pragma and Region Instrumentor (OPARI)

OPARI[14] is a source-to-source translation tool which automatically adds all necessary calls to the POMP runtime measurement library which allows collecting runtime performance data of Fortran, C, or C++ OpenMP applications. It is based on the idea of OpenMP pragma/directive rewriting.

OPARI [14] is a source-to-source translator based on a very fuzzy parser, and not a full compiler, which performs the OpenMP directive and API call transformations. The basic idea behind OPARI is to define a standard API to a performance measurement library that can be used to instrument a user's OpenMP application program for monitoring OpenMP events based on OpenMP directive rewriting [19]. This instrumentation could be done by a source-to-source translation tool prior to the actual compilation or within an OpenMP compilation system. Performance tool developers then only need to implement the functions of this interface to enable their tool to measure and analyze OpenMP programs. Different measurement modes (profiling and tracing) can easily be accommodated in this way.

The instrumentation is based on transformations of OpenMP construct, OpenMP run-time library routine, and used function. The transformations apply for both Fortran77 and Fortran90 OpenMP 2.0 directives and C/C++ OpenMP 1.0 pragmas and it does not support the instrumentation of user functions yet.

For each instrumented OpenMP construct OPARI creates a region descriptor structure that contains information such as the name of the construct, the source file, begin and end line numbers.

To integrate performance tools with the proposed OpenMP performance interface, two issues must be addressed. First, the OpenMP program must be instrumented with the appropriate performance calls. Second, a performance library must be developed to implement the OpenMP performance API for the particular performance tool. EXPERT and TAU have been integrated with the proposed OpenMP performance interface. The benefits of the proposed performance interface are several. First, it gives a performance API target for source-to-source instrumentation tools (e.g., OPARI), allowing for instrumented OpenMP codes that are portable across compilers and machine platforms. Second, the performance library interface provides a target for tool developers to port performance measurement systems. This enables multiple performance tools to be used in OpenMP performance analysis. Third, the API also offers a target for OpenMP compilers to generate pomp calls that can both access internal,

compiler-specific performance libraries and external performance packages. Finally, if the OpenMP community could adopt an OpenMP performance interface such as the one we proposed, it would significantly improve the integration and compatibility between compilers and performance tools, and, perhaps more importantly, the portability of performance analysis techniques.

OPARI has the following small limitations[20],[21]: It does not support the instrumentation of user functions yet. It requires some directives to be added for well-written code for example at the end of loop or Structured block or if-then-else. Depend on the programming language. OPARI as describe in[21] lacks the directives, and uses a different way of passing parameters to the API routines, and the API prescribes the layout of the instrumentation library data structures, thus constraining the implementers. OPARI makes implicit barriers explicit. Unfortunately, this method cannot be used for measuring the barrier waiting time at the end of PARALLEL directives because they do not allow a NOWAIT clause. Therefore, OPARI adds an explicit barrier with corresponding performance interface calls here. For OPARI, this means that actually two barriers get called. But the second (implicit) barrier should execute and succeed immediately because the threads of the OpenMP team are already synchronized by the first barrier. The OpenMP standard (unfortunately) allows compilers to ignore NOWAITs, which means that in this case OPARI inserts an extra barrier and the POMP functions get invoked on this extra (and not the real) barrier. OPARI can't instrument the internal synchronization inside!$OMP WORKSHARE as required by the OpenMP standard.

3.2 Tuning and Analysis Utilities (TAU)

TAU [2] has the ability to instrument at multiple stages of code transformation. It allows for routine, basic-block and statement-level timers to generate profiles as well as event-traces. For profiling, it allows a user to choose from measurement options that range from wallclock time, CPU time and process virtual time to hardware performance counters.

The TAU performance system [2] is designed as a tool framework, whereby tool components and modules are integrated and coordinate their operation using well- defined interfaces and data formats. The TAU framework architecture is organized into three layers: instrumentation, measurement, and analysis.

The instrumentation mechanisms in TAU support several types of performance events, including events defined by code location (e.g. routines or blocks), library interface events, system events, and arbitrary user-defined events. TAU is also aware of events associated with message passing and multi-threading parallel execution. The instrumentation layer is used to define events for performance experiments. Thus, one output of instrumentation is information about the events for a performance experiment. This information will be used by other tools.

TAU's measurement system is organized into four parts. First, the event creation and management part determines how events are processed. Two types of events are supported: entry/exit events and atomic events. In addition, TAU provides the mapping of performance measurements for "low-level" events to high-level execution entities. Second, the performance measurement part supports two measurement forms: profiling and tracing. Third, the performance data sources part defines what performance data is measurable and can be used in profiling or tracing. TAU supports different timing sources, choice of hardware counters through the PAPI [22] or PCL interfaces, and access to system performance data. Finally, the OS and runtime system part provide the coupling between TAU's measurement system and the underlying parallel system platform.

TAU comes with both text-based and graphical tools to visualize the performance profiles. ParaProf is TAU's parallel profile analysis and visualization tool. Also distributed with TAU is the PerfDMF tool providing multi-experiment parallel profile management. Given the wealth of third-party trace analysis and visualization tools, TAU does not implement its own. However, trace translation tools are implemented to enable use of Vampir, Jumpshot, and Paraver. It is also possible to generate EPILOG[2] trace files for use with the Expert analysis tool. All TAU profile and trace data formats are open. The framework approach to TAU's architecture design guarantees the most flexibility in configuring TAU capabilities to the requirements of the parallel performance experimentation and problem solving the user demands.

TAU provides a tau2vtf program to convert TAU traces to VTF3 format. In addition, TAU offers tau2epilog, tau2slog2, and tauconvert programs to convert to EPILOG, SLOG2, and Paraver formats, respectively. For convenience, the TAU tracing systems also allows traces files to be output directly in VTF3 and EPILOG formats.

TAU also supports OpenMP instrumentation using a preprocessor tool called OPARI [19]. OPARI inserts POMP annotations and rewrites OpenMP directives in the source code. TAU's POMP library tracks the time spent in OpenMP routines based on each region in the source code.

The TAU project has used PDT to implement a source-to-source instrumentor (tau instrumentor). TAU can use OPARI for automatic instrumentation of OpenMP constructs. OPARI may be used in conjunction with PDT for comprehensive automatic instrumentation of OpenMP and mixed mode parallel programs.

TAU can use DyninstAPI[23] to construct calls to the TAU measurement library and then insert these calls into the executable code. TAU can use PAPI to generate profiles based on hardware counter data. TAU has filtering and feedback mechanisms for reducing instrumentation overhead. TAU supports parallel profiling which support flat profiling, callpath profiling, calldepth profiling and phase profiling. On other hand TAU also supports parallel tracing.

TAU suffers no limitations in the ability to make low-overhead performance measurement. However, a significant amount of performance data can be generated for large processor runs. The TAU Para-Prof tool provide the user with means to navigate through the profile dataset

V. Discussion and Conclusion

Typically, the comparison among the performance tools can be summarized according to the following aspects:

- The version: it describes the tool version.
- The platform: it describes the hardware architecture.
- The operation system: it describes the OS supported by the tools.
- The parallel language: it describes the parallel language supported by the tool.
- The compiler: it describes the compiler used by the tool.
- The instrumentation: it describes the instrumentation type.
- The measurement; that describes the usage of measurement approach.
- the analysis; that specifies how the measured data are analyzed and processed;
- The presentation; that defines how the gathered performance data is displayed to the user.

Table II, III show the comparison of the tools which are related to the performance OpenMP. Unfortunately, none of these tools has been used to measure the performance of OpenMP on windows multi-core platform. Only Intel Thread profiler has been used to analyse the performance of OpenMP on multi-core architecture.

Our research is to understand the performance of the OpenMP application when run on windows multi-core. It is recommended that further tools should be designed to better understand the system and the user function behavior, as well as OpenMP directives.

TABLE II
Software Tools Comparison

	PAPI	OPARI	TAU
Version	PAPI 3.6.2	OPARI 1.1	TAU 2.17.3
Platform	Intel Pentium, IA-64, AMD Athlon, IBM POWER series,Cray T3E, SGI, Sun Ultrasparc, Compaq Alpha	SGI, Altix, Ultrasparc II, Itanium 2, Pentium3, Pentium 4 and IBM platforms.	SGI Power Challenge and Origin 2K, IBM SP2, Intel Teraflop, Cray T3E, HP 9000, Sun, Compaq Alpha Linux cluster, Intel Linux cluster
Operating Systems	Linux, True64, Unicos, AIX, Irix, Solaris	Linux	LINUX , windows, Solaris, SGI, IBM, and MacOS X, IRIX [tau web]
Parallel language	C and fortran, can support OpenMP	OpenMP application by C/C++ and fortran	For parallel programs (MPI,OpenMP) written in Fortran, C, C++, Java, Python.
Compiler	GCC	OPARI was tested with KAI's KCC version 3.4 and 4.0, GNU's g++ version 2.95.3 and later, IBM xlC 5.x, SUN CC 6.1, and SGI 7.3.1., PGI	KAI, PGI, GNU, Fujitsu, Sun, Microsoft, SGI, Cray
Instrumentation System	performance counters	Automatically source-to-source translator based on fuzzy parser. Rewriting OpenMP directive (directive transformations).	Tau insert probes at several level of program transformation process in the source, preprocessor, compiler, wrapper library, binary, interpreter, component, virtual machine, and multi-level instrumentation
Measurement System	Hardware sampling profile for system behavior.	Profiling and tracing by integrated with other tools (64)	Both profiling and tracing. Hardware counter.
Analysis System	On-line	Integrated with other tools TAU and EXPERT	Online analysis. Using a set of anayliss tools (ParaProf, PerfDMF).
Visualization	Perfometer,profometer, it is integrated with other tools(vprof,SvPablo, DEEP,TAU, VAMPIR,KAPPro).	No visual representation but integrated into TAU and EXPERT.	TAU trace can be convert to VTF3,EPILOG, SLOG2, AND Paraver formats to used with other visualization tools.

TABLE II
SOFTWARE TOOLS COMPARISON

	ompP	Kojak	Sun Performance Analyzer	Intel thread profiler
Version	0.6.7	2.2b3	12	3.1
Platform	Linux/x86, IA64, ALX, Linux	Linux IA-32, IA-64, EM64T/x86_64,IBM Power3 / Power4 based clusters ,SGI Mips based clusters (O2k, O3k) ,SGI IA-64 based clusters (Altix) ,SUN Solaris Sparc and x86 based clusters,DEC/HP Alpha based clusters, Cray T3E, XD1 and X1,IBM BG/L,NEC SX, Hitachi SR-8000	The SPARC® and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, Pentium, and Xeon EM64T	Support for the latest multi-core processors on the new Intel Core 2 Duo and Intel Core 2 Quad processors.
Operating Systems	Unix like OS., Linux, AIX	Linux,	Linux, Solaris	Windows Vista, Windows XP Professional, Windows Server 2003, or Windows XP Professional x64 Edition or newer
Parallel language	OpenMP written in C/C++ or FORTRAN.	MPI, OpenMP, SHMEM, and combinations thereof written by C, C++, or Fortran,	MPI, OpenMP, hybrid MPI/OpenMP written by c/c++ or fortran	OpenMP, Windows API, or POSIX threads (Pthreads).
Compiler	Linux, Solaris, AIX and Intel, Pathscale, PGI, IBM, gcc, SUN studio compilers	GNU, SUN,PGI Intel compilers, openUH	Sun Studio compiler	Intel C++ Compiler 8.1 for Windows or higher, Intel Fortran Compiler 8.1 for Windows.
Instrumentation System	relies on OPARI for source-to-source instrumentation, PAPI for hardwar counter,profiling	using OPARI to performs automatic source-code level instrumentation using TAU to perform source-code or using a compiler-supplied profiling interface. Using PMPI wrapper library, which generates MPI-specific events by intercepting calls to MPI functions.	No instrumentation. Using .it used sun‘s collector which communicate between the collector and the per	plug-in to Vtune time-based and event-base hardware counter samples. Binary instrumentation And source instrumentation.
Measurement System	Profile only count and time for instrumented OpenMP constructs. Hardware counter can be used.	Event trace	Perform statistical profiling of a wide range of performance data and tracing of functions calls, global data of system routines. Hardware counters	time-based and event-base hardware counter sample
Analysis System	Off-line	Off-line analysis Using EXPERT analyzer. Trace file can convert into VTF3 format and analyze them manually with VAMPIR event trace analysis tools.	Off-line analysis	Off-line analysis
Visualization	Only two formats of output are specified plaintext ASCII and comma separated values (CSV). It shows the call graph region profiles	Using CUBE Also can export to Vampir’s VT3 format Using EPILOG for Event Processing Using EARLfor Event Analysis Using EARL to provide a high-level view of the raw trace file.	displays the raw data in a graphical format as a function of time/ plain text, call tree	It display the Critical Path View, and timeline,display the source code view

REFERENCES

[1] Graham, S.L., M. Snir., and C.A., Getting up to Speed: the Future of Supercomputing. National Academies Press, 2004.
[2] Shende, S.S. and A.D. Malony, The TAU parallel performance system. International Journal of High Performance Computing Applications, 2006. 20(2): p. 287-311.
[3] Mohr, B. and F. Wolf, KOJAK – A Tool Set for Automatic Performance Analysis of Parallel Programs, in Euro-Par 2003 Parallel Processing. 2003. p. 1301-1304.
[4] Fürlinger, K. and M. Gerndt, ompP: A Profiling Tool for OpenMP, in OpenMP Shared Memory Parallel Programming. 2008. p. 15-23.
[5] Board, A.R. openMP. 2008 [cited; Available from: http://www.openmp.org/.
[6] Luiz, D. and W. Felix, CATCH - A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications, in Proceedings of the 8th International Euro-Par Conference on Parallel Processing. 2002, Springer-Verlag.
[7] Nagel, W.E., et al., VAMPIR:Visualization and analysis of MPI resources. SUPERCOMPUTER, 1996. 12(1): p. 69–80.
[8] Intel. Intel VTune Performance Analyzer. 2007 [cited; Available from: http://www.intel.com/cd/software/products/asmo-na/eng/219898.htm.
[9] Browne, S., et al., A portable programming interface for performance evaluation on modern processors. International Journal of High Performance Computing Applications, 2000. 14(3): p. 189-204.
[10] P. Mucci, Dynaprof 0.8 user's guide. Technical report. Nov. 2002.
[11] Mohr, B., et al. A Performance monitoring Interface for OpenMP. in Proc. of 4th european Workshop on OpenMP(EWOMP). 2002. Rome,Italy.
[12] Sameer Shende, et al., Integrated Tool Capabilities for Performance Instrumentation and Measurement.
[13] Bui, V., PerfOMP A Runtime Performance event Monitoring Interface for OPENMP. 2007.
[14] Mohr, B., et al., Toward a Performance Tool Interface for OpenMP: An approch based on directive rewriting. In Proceeding of the Third workshop on OpenMP, September, 2001.
[15] Intel Thread Profiler. 2007 [cited; Available from: http://www.intel.com/cd/software/products/asmo-na/eng/threading/286749.htm.
[16] SUN, M.I. Sun Collector and Performance Analyzer. [cited; Available from: http://developers.sun.com/sunstudio/index.jsp.
[17] Hernandez, O., et al., Performance Instrumentation and Compiler Optimizations for MPI/OpenMP Applications, in OpenMP Shared Memory Parallel Programming. 2008. p. 267-278.
[18] Wolf, F. and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. in Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on. 2003.
[19] Bernd, M., et al., Design and Prototype of a Performance Tool Interface for OpenMP. J. Supercomput., 2002. 23(1): p. 105-128.
[20] Pallas, G., Public OpenMP Instrumentation Interface Specification, I.S.T.I. PROGRAMME, Editor.
[21] Luiz, D., M. Bernd, and S. Seetharam. An Implementation of the POMP Performance Monitoring Interface for OpenMP Based on Dynamic Probes. in Proceedings of the fifth European Workshop on OpenMP. 2003.
[22] University of Tennessee, PAPI User's Guide, 3.5.0 edition.
[23] Bryan, B. and K.H. Jeffrey, An API for Runtime Code Patching. Int. J. High Perform. Comput. Appl., 2000. 14(4): p. 317-329.