Computing and Informatics, Vol. 33, 2014, 921–942

SCALABILITY AND PERFORMANCE ANALYSIS OF OPENMP CODES USING THE PERISCOPE TOOLKIT

Shajulin Benedict

HPCCLoud Research Laboratory
Department of Computer Engineering
St. Xavier's Catholic College of Engineering
Nagercoil 3, India
e-mail: shajulin@sxcce.edu.in

Michael Gerndt

Informatics I10
Technische Universität München
Boltzmannstrasse 3, Germany
e-mail: gerndt@in.tum.de

Abstract. In this paper, we present two new approaches while rendering necessary extensions to Periscope to perform scalability and performance analysis on OpenMP codes. Periscope is an online-based performance analysis toolkit which consists of a user defined number of analysis agents that automatically search for the performance properties while the application is running. In order to detect the scalability and performance bottlenecks of OpenMP codes using Periscope, a few newly defined performance properties and meta properties are formalized. We manifest our implementation by evaluating NAS OpenMP benchmarks. As shown in our results, our approach identifies the code regions which do not scale well and other performance problems, e.g. load imbalance in NAS parallel benchmarks.

Keywords: Memory accesses analysis, OpenMP, performance analysis, program transformations, speedup, supercomputers

Mathematics Subject Classification 2010: 68M14, 68M20

1 INTRODUCTION

OpenMP is one of the successful APIs for scientific applications [2, 9]. It is easy to use, scalable, and portable which attracts most of the parallel programmers to developing their applications using OpenMP.

In general, application developers perform a stepwise parallelization of a pre-existing serial program which may lead to performance problems, e.g. resource under-utilization. The application developer might not exactly pinpoint the locations of code regions which have performance problems. Additionally, they will be interested in automatically finding the speedup of code regions and the code regions which do not scale well. In recent days, researchers have started investigating into the possibility of efficiently parallelizing their applications automatically combined with some tuning mechanisms [12, 16]. In such cases, performance analysis tools are significantly important for the application developers to perform performance analysis.

A few performance analysis tools exist [4, 8] which help the user to identify whether or not their application is running efficiently on the computing resources available. However, they are not online and distributed; they do not have a support to do OpenMP or scalability analysis for code regions; or, they are commercial and expensive. Periscope [6] is an open-source online-based performance analysis toolkit that searches for performance problems using agents in a distributed fashion.

Our main contributions of this paper are as follows:

1. Proposing two new search strategies while rendering extensions to the Periscope toolkit. They are a) *OpenMPAnalysis* which is responsible for evaluating the OpenMP regions for performance bottlenecks and b) *ScalabilityAnalysis* which is responsible for finding the scalability problems.
2. We define properties formalizing the OpenMP performance problems and the scalability issues.
3. We evaluate our techniques with the OpenMP benchmarks from the NAS Parallel Benchmark (NPB) suite, i.e., LU, LU-HP, SP, BT, CG, IS, EP, MG, and FT. The experiments were executed on the Altix 4700 supercomputer which supports OpenMP-runs within a partition of 512 cores.

The rest of the paper is organized as follows. Section 2 presents existing work and Section 3 explains Periscope architecture, search methodologies, and the formalized performance properties. Section 4 discusses experimental results. Finally, Section 5 presents a few conclusions.

2 EXISTING WORK

In this section, we introduce the major existing contributions from the research areas such as scalability analysis, performance analysis, and tools. In addition, the existing works on performance analysis tools are classified.

2.1 Scalability Analysis

Designing scalable architectures, suitable information retrieval systems [18], and analyzing the scalability features of applications are remaining as a long standing research in the HPC domain.

Scalability analysis on parallel codes was introduced a couple of decades ago. Since then, scientific application developers have been profoundly interested in finding the scalability issues on their codes. There exist numerous approaches to perform scalability analysis. However, they are not capable of identifying the scalable issues online; they do not expose scalability issues for parts of the codes, and they are not automatic.

The traditional approaches used for scalability analysis were dependent on statistics, knowledge, prediction, and learning methodologies. In [7], the authors took advantage of statistics and Taylor's expansion for predicting the execution time in parallel applications. It was strictly an offline approach. An automated scalability study for the MPI parallel programs was carried out in [14].

In recent years, a few research works have extended the traditional approaches for scalability analysis, such as in [11] and [13]. In [11], the authors implemented the scalability tests based on knowledge theory and statistical theory. The tests were carried out only for MPI programs. In addition, their approach produced more trace files while analyzing the code. The works carried out in [13] relied more on prediction and machine learning. In this approach, they adopted a case-based framework on which the machine learning technique was exploited to identify the scalability issues. There also exists a tool named ADEPT [1] which predicts scalability for parallel codes based on learning methodology. The authors of [1] have predicted scalability for the whole program and compared their results with other algorithms such as the genetic algorithm.

Note that most of the literature on scalability analysis for a parallel code is based on learning, prediction, machine learning, and so forth which concentrated only on the whole program – scalability analysis is not performed for code regions; and, most of them did the analysis offline. But, in reality, parallel programmers are interested in online scalability analysis for specific fragments of the code.

2.2 Performance Analysis Tools

In general, performance analysis and tuning of threaded programs is one of the biggest challenges of the multi-core era. In such situations, the performance analysis tools are significantly important for understanding the program behavior. The existing performance analysis tools are classified as follows.

2.2.1 Profile-Based Tools

'ompP' [10] is a text-based profiling tool for OpenMP applications. It relies on OPARI for source-to-source instrumentation. It is a measurement-based profiler and

does not use program counter sampling. An advantage of this tool is its simplicity. ompP performs an overhead analysis based on synchronization, load imbalance, thread management, and limited parallelism.

Intel Thread Profiler [8] supports applications threaded with OpenMP, Windows API, or POSIX threads (Pthreads). Thread Profiler is used to identify bottlenecks, synchronization delays, stalled threads, excessive blocking time and so forth. It is a plug-in to the VTune Performance Analyzer. It provides results in the form of graphical displays.

2.2.2 Trace-Based Tools

Scalasca [5] performs an offline automatic analysis of OpenMP codes based on profiling data. OpenMP events are also inserted into a trace and can be visualized with Vampir. *Vampir* [3] is exclusively using traces of OpenMP programs and presents analysis results via a rich set of different chart representations, such as state diagrams, statistics, and timelines of events.

TAU [17] supports trace-based and profiling-based performance analysis. It performs an offline analysis and provides graphical and text-based displays. It uses Vampir to display traces of OpenMP executions.

Our implementation – scalability and performance analysis of OpenMP codes using Periscope – compared to the above approaches succeeds in the following areas:

1. online-based scalability and performance analysis
2. analysis only for the user-specified parts of the code
3. delivers the main bottlenecks including their severity
4. speedup calculation for the individual fragments of the code
5. identifies and reveals reasons behind the performance problem.

3 SCALABILITY AND PERFORMANCE ANALYSIS

In order to brief our implementation – OpenMP and Scalability analysis – which constitutes this paper, a short synopsis about the Periscope toolkit is explained. In addition, this section explains the Periscope extensions, OpenMP performance analysis, scalability analysis, and the formalized performance properties.

Periscope is an automatic performance analysis tool that searches for pre-defined performance properties which are based on measurements during program execution. The search is carried out by analysis agents based on the phase concept. Scientific applications are iterative and each iteration executes the same code, which is called the phase region in Periscope. The agents analyze the performance for an execution of the phase region, and, if necessary, refine the search for performance problems in subsequent executions.

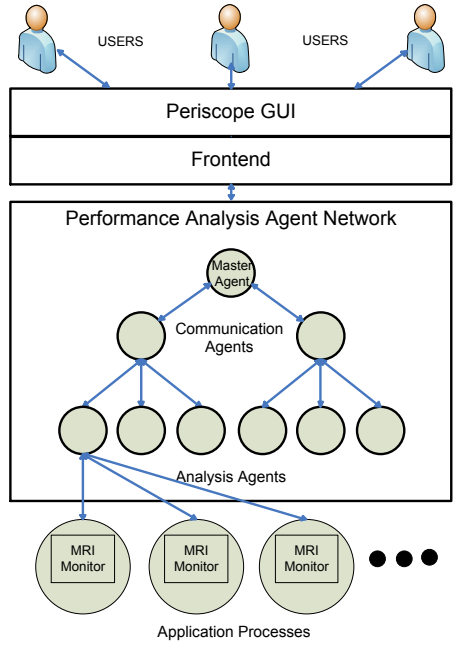

Figure 1. Periscope toolkit architecture

3.1 Entities

As shown in Figure 1, Periscope consists of four major entities, namely, User-Interface, Frontend, Analysis Agent Network, and Monitoring Request Interface (MRI) monitors. All entities have their own responsibilities to finally identify the performance problems in parallel applications, as below:

1. The User-Interface is an Eclipse-based GUI which has the capability to map and to display the performance analysis results with their corresponding code regions.
2. The Frontend starts the application and the analysis agents based on the specifications provided by the user, namely, number of processes and threads, search strategy, and so on. The Frontend can restart the application if the search is incomplete; it restarts both the application and the agent network for scalability analysis.
3. The Analysis Agent Network consists of three different agent types, namely, master agent, communication agent and analysis agent. The master agent forwards commands from the Frontend to the analysis agents and receives the found performance properties from the individual analysis agents and forwards them to the Frontend. The communication agents combine similar properties found

in their sibling agents and forward only the combined properties. The analysis agents are responsible for performing the automated search for performance properties based on the search strategy selected by the user.

4. The MRI monitors linked to the application provide an application control interface. They communicate with the analysis agents, control the application execution, and measure performance data.

The functionality of Frontend is extended for scalability analysis which will be explained in Section 3.3. A more detailed description of the architecture can be found in [15].

3.2 OpenMP Performance Analysis

With respect to OpenMP, the analysis agents follow the OpenMPAnalysis strategy to search for performance properties related to

1. extensive startup and shutdown overhead for fine-grained parallel regions,
2. load imbalance,
3. sequentialization, and
4. synchronization.

3.2.1 Search Methodology

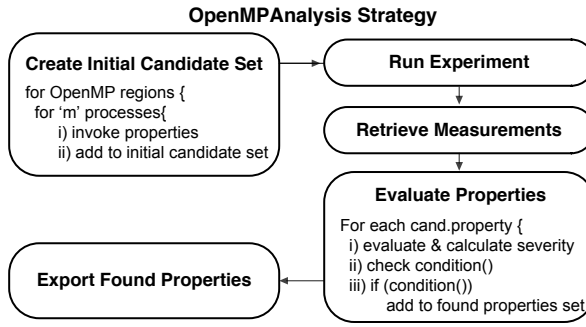

Figure 2. Methodology to find OpenMP properties and their severity using the OpenMP-Analysis strategy in Periscope

Figure 2 explains the steps involved in the OpenMPAnalysis search strategy as being executed by the analysis agent. The steps are detailed as follows:

Create initial candidates set. At this stage, the strategy first creates OpenMP candidate properties for every OpenMP region in the application. This is based on the static program information created by Periscope's source code instrumentation tool.

Configure performance measurements. Based on these candidate properties it requests the measurements of the required performance data to prove whether the properties exist in the code. Each property provides the information which data are required.

Execution of experiment. The agents release the application which then executes the phase region. The measurements are done automatically via the MRI monitor.

Evaluate candidate properties. The application stops at the end of the phase region and the agents retrieve the performance data. All the candidate properties are then evaluated whether their condition is fulfilled and the severity of the performance problems is computed. Finally, the found property set is checked whether further refinement is appropriate.

3.2.2 OpenMP Performance Properties

The OpenMP performance properties and scalability properties (which will be explained in Section 3.3.2) are specified as C++ classes. The properties provide methods to determine the required information, to calculate the condition as well as to compute the severity. The notations used while defining properties are as follows:

- *Severity*: significance of the performance problem
- *reg*: region name
- k: thread number
- n: number of threads
- T_0: execution time for a master thread
- $T_{1...(n-1)}$: execution time for the team members – other than a master thread
- *config*: configurational numbers used for a run – which represent the number of processes and the number of threads used – e.g., 2×4 means 2 processes and 4 threads in a run.
- *phaseCycles*: time spent in executing the entire phase.

The region in the source code is identified through the file identification and the first line of that region in the file. In the following, we present the individual properties that are currently included in the OpenMPAnalysis strategy.

Parallel Region Startup and Shutdown Overhead Property. For each execution of a parallel region in OpenMP, the master thread may fork multiple threads and destroy those threads at the end. The master thread continues execution after the parallel regions. In addition, initialization of thread private data and aggregation of reduction variables have to be executed. Writing too many parallel regions in an application causes overhead due to startup and shutdown process. This performance property is designed to expose parallel region startup and shutdown overhead in parallel regions.

To calculate the severity of the Parallel Region Startup and Shutdown Overhead property, we measure the following:

- the parallel region execution time for the master thread T_0
- the execution time for the parallel region body for thread k, T_k.

Severity is calculated using the formula given below:

$$\text{Severity(reg)} = \frac{T_0 - \sum_{k=0\ldots n-1}(T_k/n)}{\text{phaseCycles}} * 100. \tag{1}$$

Load Imbalance in OpenMP Regions. Load imbalance emerges in OpenMP regions from an uneven distribution of work to the threads. It manifests at global synchronization points, e.g., at the implicit barrier of parallel regions, worksharing regions, and explicit barriers. Load imbalance is a major performance problem leading to the underutilization of resources.

The Load Imbalance in Parallel Region property is reported when threads have an imbalanced execution in a parallel region. In order to calculate the severity, we measure implicit barrier wait time W; then, the difference of the unbalance time UT and the balanced time BT is calculated.

$$\text{Severity(reg)} = \frac{UT - BT}{\text{phaseCycles}} * 100. \tag{2}$$

UT and BT are represented in Equations (3) and (4).

$$UT = \max\{W_0 \ldots W_n\} \tag{3}$$
$$BT = \overline{\text{Work}} + \min\{W_0 \ldots W_n\} \tag{4}$$

where $\overline{\text{Work}}$ is the average computational work of all the threads executed during the maximum barrier wait time.

$$\overline{\text{Work}} = \sum_{0 \le k \le n} (\max\{W_0 \ldots W_n\} - W_k). \tag{5}$$

Equation (2) is common for most of the load imbalance OpenMP properties.

The parallel loop region distributes the iterations to different threads. While the scheduling strategies determine the distribution of iterations, the application developer can tune the application by selecting the best strategy. Often, choosing a better scheduling strategy with correct chunk size is a question because even most experienced developers are new to this programming sphere. A suboptimal strategy might thus lead to load imbalance. The measurements are done similar to the Load Imbalance in Parallel Region property. Measurement of the load imbalance based on the barrier time is only possible if the parallel loop is not annotated with the nowait clause.

In OpenMP, the sections construct allows the programmer to execute independent code parts in parallel. A load imbalance manifests at the implicit barrier region, similar to the parallel region, and determines the underutilization of resources.

The Load Imbalance in Parallel Sections property is further refined into two sub properties as follows:

- Load Imbalance due to not Enough Sections property is reported when the number of OpenMP threads is more than the number of parallel sections. In this case, a few threads do not participate in the computation.
- Load Imbalance due to Uneven Sections property identifies the load imbalance which is due to the fact that threads execute different number of sections.

However, the calculation for the severity is quite similar to Equation (2) except that additional static information (the number of sections in the construct) and the number of sections assigned to the threads at runtime is checked.

Application developers often use explicit barriers to synchronize threads, so that they avoid race conditions. Early threads reaching the barrier have to wait until all threads reach it before proceeding further. Severity is calculated after measuring the explicit barrier time for each thread using Equation (2). Note that the wait time in this property is the execution time of the explicit barrier.

Sequential Computation in Parallel Regions. The computational effort C of any OpenMP code in a parallel region is the sum of the execution E and the idle time I, where idle time is the time that threads wait for useful work.

$$C = \sum_{k=0...n-1} (E_k + I_k) \tag{6}$$

In general, if parallel codes spend too much time in sequential regions, this will severely limit scalability as shown in the famous Amdahl's law. Sequential regions within parallel regions are coded in the form of master, single, and ordered regions.

If a master region is computationally expensive it limits scalability. Severity of Sequential in Master Region property is the percentage of the execution time of the phase region spent in the master region.

$$\text{Severity(reg)} = \frac{T_0}{\text{phaseCycles}} * 100 \tag{7}$$

The underlying principle of a single region is similar to the master region: code wrapped by OMP MASTER directive is only intended for master thread, code wrapped by OMP SINGLE directive is intended for one and only one thread but not necessarily for the master thread. Thus, the code is executed sequentially. Severity is calculated in the same way as for the master region.

An ORDERED region in the body of a parallel loop specifies that this code is executed in the order of the sequential execution of the loop. Thus, the code is

executed sequentially. This performance property is modeled in a way to measure the performance loss due to this ordered execution constraint. Severity is computed based on the SUM of the time spent in the ordered region O_k in all the threads.

$$\text{Severity(reg)} = \frac{\sum_{k=0\ldots n} O_k}{\text{phaseCycles}} * 100 \tag{8}$$

OpenMP Synchronization Properties. In addition to the above mentioned OpenMP properties, we have defined few properties that are specific to synchronization in OpenMP, namely, critical sections and atomic regions.

For critical sections two aspects are important. The first is the contention for the lock guarding its execution. The second is the wait time of other threads while a thread is executing within a critical region. Severity of the Critical Section Overhead property is calculated by taking the maximum value of critical section overhead (CSO) among the threads. The CSO is the difference between the critical section region's execution time C and the execution time of the critical section body CB.

$$\text{Severity(reg)} = \frac{\max\{CSO_0 \ldots CSO_n\}}{\text{phaseCycles}} * 100 \tag{9}$$

where $CSO_k = C_k - CB_k$.

To eliminate the possibility of race condition, the ATOMIC directive specifies that a memory location will be updated atomically. Those who are familiar with POSIX threads are aware of the overhead for operating system calls to use semaphores for this purpose. Similarly, too many atomic operations have a negative effect on the performance. Thus, severity of this property is the percentage of time spent in an atomic region with respect to phase.

3.3 Scalability Analysis

Scalability analysis is an important aspect in parallel applications which identifies whether the application had scaled with increasing processes. Many users find scalability analysis as time consuming, difficult, and inconclusive. Our approach automatically searches for the scalability issues on OpenMP code regions such that the user need not perform the analysis manually (Please note the time the user invests for performing the scalability analysis).

3.3.1 Search Methodology

Scalability analysis for OpenMP codes in Periscope requires OpenMPAnalysis search process. The Frontend entity automatically restarts application and performs a basic OpenMP analysis with different configurations ($m \times n$), where m represents the number of processes and n represents the number of threads. A basic OpenMP analysis for an application considering single configuration is defined as a Configurational Run in Periscope. We assume that the number of processes is kept constant and the

number of threads increases in the order of 2^n when the application is restarted. For example, if the user needs scalability analysis with 4×64 configurations, then the Frontend performs 4×1, 4×2, 4×4, 4×8, 4×16, 4×32, and 4×64 configurational runs. Hence, each configurational run $(m \times n)$ will have a set of OpenMP performance properties ($OpenMP\_Prop(k)$) with its severity. After a 2^n configurational run, the Frontend will start the scalability analysis phase (SCA).

During SCA (see Algorithm 1), the Frontend first extracts the necessary details from the found properties, e.g., Code Line Number. Next, it performs speedup analysis for eligible OpenMP regions – parallel, sections, call region inside the parallel region, and so forth – based on their execution time. Then, the Frontend transfers the scalability properties which are observed from the properties from each configurational runs to Periscope GUI. At last, the Periscope GUI reports on the performance properties to the user. Note that the Frontend selects either an interactive or a batch mode operation based on the user requirements for scalability analysis.

Algorithm 1 Search Methodology for Scalability Analysis in Periscope

Require: $frontend \leftarrow user\_details$
  for $k = 1$ to 2^n **do**
    if *interactive* **then**
      $frontend \leftarrow Interactive\_mode$
      $frontend \overset{Interactive}{\Longrightarrow} Agent\_Net \wedge Appl$
    else
      if *batch* **then**
        $frontend \leftarrow Batch\ mode$
        $frontend \overset{Batch}{\Longrightarrow} Agent\_Net \wedge Appl$
      end if
    end if
    $frontend \leftarrow Agents \leftarrow \{OpenMP\_Prop \Leftrightarrow (condition \geq threshold)\}$
    $k \leftarrow k + k$
  end for
Ensure: $frontend \leftarrow OpenMP\_Prop$
  $frontend \implies SCA \leftarrow \{SCA\_Prop \Leftrightarrow (condition \geq threshold)\}$
Ensure: $frontend \leftarrow OpenMP\_Prop \cup SCA\_Prop$
  return $psc\_gui \leftarrow frontend\_Prop$
  print *found_properties*

3.3.2 Scalability-Based Performance Properties

Scalability-based performance properties are the formalized representation of scalability problems. They are formalized either from all configurations or from single configuration. Scalability-based properties are classified into three categories, as follows:

1. speedup-based properties,
2. meta-properties, and
3. sequential computation.

Most of the scalability properties discussed below are derived from all configurations. However, the Low Speedup property and Sequential Computation property are derived from single configuration.

Speedup-Based Properties. The speedup-based properties are based on the execution time of OpenMP code regions. They indicate both the positive as well as the negative aspects of the scalability analysis. For example, the speedup-based properties such as Linear Speedup for All Configurations and SuperLinear Speedup for All Configurations reveal the positive aspects on the code region, whereas the speedup-based properties such as Speedup Decreasing, Linear Speedup Failed for the First Time, and Low Speedup reveal the negative aspects on the code region.

Severity calculated for speedup-based properties is given in Equation (10).

$$\text{Severity}(\text{reg}, \text{config}) = \frac{D(\text{reg}, \text{config})}{\text{phaseCycles}} * 100 \tag{10}$$

where $D(\text{reg}, \text{config})$ is the deviation time which is dependent on configurations. The deviation time (Equation (11)) is the difference between the execution time for a code region and its sequential run time, as follows:

$$D(\text{reg}, \text{config}) = \mathit{Execution\_Time} - \left(\frac{\mathit{Sequential\_Time}}{\text{config}}\right) \tag{11}$$

if Execution Time $> \frac{\text{Sequential Time}}{\text{config}}$
else 0

SuperLinear Speedup for All Configurations property identifies the code regions which have superlinear speedup for all the configurations. In general, superlinear speedup occurs in a code region due to the cache effects in the machine. As seen in Figure 3 b), the speedup should be more than the number of threads used. Similarly, Linear Speedup property identifies the code regions which have a linear speedup. The two properties, namely, SuperLinear Speedup for All Configurations and Linear Speedup for All Configurations have zero severity and indicate a positive sign.

Speedup Decreasing property identifies the configuration from which the speedup decreases. In addition, it also points out the maximum speedup value a code region can attain in the analysis. In some cases, the speedup for a code region might not decrease for a finite number of runs although the speedup is not linear.

Low Speedup property reports on the code regions which have a low speedup value for a configuration. The low speedup value is determined based on a predefined threshold value. Additionally, one of the most interesting perspectives is to find the region which has the lowest speedup for that configuration. However, this

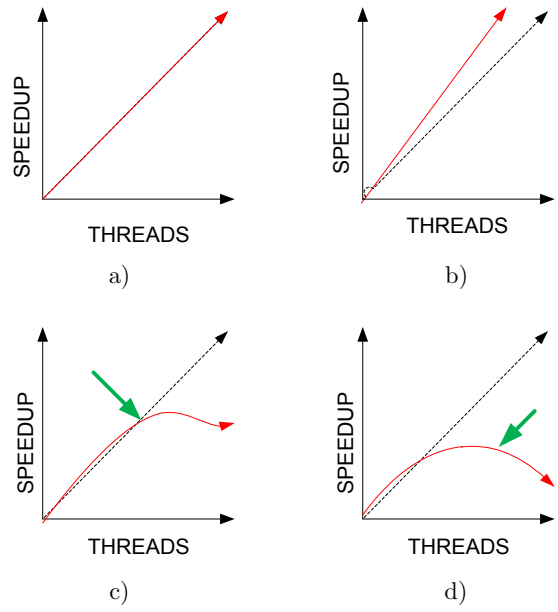

Figure 3. Speedup-based scalability properties: a) Linear Speedup, b) Super Linear Speeup, c) Speedup failed for the first time, d) Speedup Decreasing

region can be better when executed with other configurations. The Code Region with the Lowest Speedup in the Run property is formalized in order to show the most severe code region for the configuration.

Meta-Properties. In general, if the severity of OpenMP-based performance property, e.g., load imbalance, will occur for all configurational runs for the same code region, then the code region has a scalability problem. The properties, namely, Property occurring in all Configurational Runs (POAC) and Property with Increasing Severity across Configurational Runs (PISC) are formalized by investigating the OpenMP analysis results. Hence, these properties are named Meta-Properties (MP).

Property Occurring in all Configurational Runs identifies the OpenMP-based property that repeats for all configurational runs; it reports the identified property; and, it calculates severity. In some cases, severity of an OpenMP-based property keeps increasing with increased configurations. Property with Increasing Severity across Configurational Runs identifies such an OpenMP-based property. Algorithm 2 illustrates the procedure to find the meta-properties.

Sequential Computation. Apart from Meta-Properties and the Speedup-based properties, we have also defined a property that is essential in highlighting the importance of the parallelism on a code. In a parallel application, efficiency of the

Algorithm 2 Procedure to find Meta-Properties (MP)

Require: $MP = POAC = PISC = \emptyset \wedge OpenMP\_Prop \neq \emptyset$
Let: $OpenMP\_Prop = \{P_i(k)\}$ where, $i = 1, 2, \ldots, p$; $k = 1, 2, \ldots, 2^n$
$OpenMP\_Prop \vdash MP \implies$ MP is derived from $OpenMP\_Prop$
Add in List: $MP = POAC \cup PISC$, where,
$POAC(k = any) \in \{P_i(k)\} \Leftrightarrow (\exists P_i \forall k)$
$PISC(k = any) \in \{P_i(k)\} \Leftrightarrow (\exists P_i \forall k \wedge (S(P_i(k)) \geq S(P_i(k+1)))$
where, $S(P_i(k))$ is the severity of the OpenMP property
Do Severity Calculation: $S(POAC) \wedge S(PISC) \Leftrightarrow (condition \geq threshold)$
$S(POAC) = S(POAC(k = 2^n)) - S(POAC(k = initial))$
$S(PISC) = S(PISC(k = 2^n)) - S(PISC(k = initial))$
return $frontend \leftarrow MP$

code is determined by the amount of parallelism. The efficiency is affected by various factors. They are as follows:

1. Mostly parallel programs are developed from the existing sequential applications. Consequently, application developers may not change the whole portion of the code for attaining parallelism which leads to inefficiency.
2. Most of the parallel programming languages have some options to write a portion of the code sequentially. For instance, OpenMP programming language has constructs for developing sequential code regions, such as *#pragma omp master*, *#pragma omp single*, and so forth. The programmer may write the parallel application using more sequential regions which is inefficient.
3. Programmers might have a little knowledge about the implementation aspects.

Sequential Computation property brings forward the total sequential computation in the parallel code and its severity. Severity is calculated as follows:

$$\text{Severity(reg)} = \frac{\text{phaseCycles} - \text{parallelTime}}{\text{phaseCycles}} * 100. \tag{12}$$

4 EXPERIMENTAL RESULTS

To demonstrate the search approaches, the formulated performance properties, and their effects, we tested our method on an ALTIX machine located at the Leibniz Computing Center (LRZ) in Garching. On this unique machine OpenMP scales up to 500 cores per partition, for a total of 19 partitions. Each partition has 256 Itanium 2 dual core processors with a peak performance of 12.8 GFlops. The tests were carried out for the Numerical Aeronautical Simulation (NAS) OpenMP benchmarks.

4.1 NPB OpenMP Analysis

The NAS Parallel Benchmarks (NPB), a suite of performance benchmarks since 1991, were originally developed at the NASA Ames Research Center. Although the benchmarks were initially designed as a set of "paper and pencil" benchmarks, they were written for the high-end parallel supercomputers in different parallel programming languages. We have used the NPB 3.2 OMP version for our evaluation.

We have used the following NAS benchmarks for the evaluation of our OpenMP scalability analysis:

1. Block Tridiagonal (BT). BT calculates a Computational Fluid Dynamics (CFD) problem. In order to solve the problem, it uses independent systems of non diagonally dominant block tridiagonal equations.
2. Lower and Upper triangular systems (LU). LU performs a synthetic CFD calculation as in BT. The difference is that it performs the calculations for the regular sparse, lower and upper triangular systems.
3. Scalar Pentadiagonal (SP). This also calculates CFD; but, the solution is based on solving independent systems of non diagonally dominant, scalar, pentadiagonal equations. The benchmarks, namely, BT, LU, and SP seem similar because they calculate the CFD. However, the main difference relies on the solving procedures and the computation and communication aspects. For instance, BT has more computations than the others.
4. Conjugate Gradient (CG). As the name indicates, this benchmark uses the conjugate gradient method to solve matrices. Additionally, it employs unstructured matrix vector multiplication. It computes an approximation to the smallest eigenvalue of a large, sparse, symmetric positive dense matrix.
5. Embarrasingly Parallel (EP). EP aims at providing an embarrassingly high parallelism among the other benchmarks. Such parallelism is achieved due to the fact that there is a negligibly small interprocessor communication, compared to the other benchmarks.
6. Fourier Transform (FT). FT uses Fast Fourier Transforms (FFTs) to solve a 3-D Poisson partial differential equation. It emulates the essence of many spectral codes.
7. Integer Sort (IS). The Integer Sort benchmark computes a large sorting operation. The main objective of the benchmark is to test the speed of the sorting operation and to test the performance of communication between the processes.
8. MultiGrid (MG). MG performs a multigrid calculation. The MG calculation is based on long or short distance data communication on a highly structured grid. This benchmark undergoes several iterations to finalize the solutions.

4.1.1 Performance Properties

The performance analysis of NAS benchmarks was carried out with 128 threads. The benchmarks were chosen with CLASS C, i.e., the workload size of $162 \times 162 \times 162$. On the analysis run, Periscope automatically executed the benchmark several times such that each run was executed with 1, 2, 4, 8, 16, 32, 64, or 128 threads. For every run, it identified the performance bottlenecks on code regions with respect both to the scalability and the OpenMP performance issues. Finally, at the end of the runs, Periscope reported the performance data.

Table 1 shows the identified OpenMP performance properties and the scalability performance properties for the code regions.

NAS	S.C.P	File	Region	R.F.L	L.S.F	F.P.N	Severity in Percentage						
							2	4	8	16	32	64	128
BT	2.42	rhs.f	Par.Reg.	27	8	L.S	0.22	0.73	0.64	0.79	0.7	0.88	1.01
		initial.f	Par.Reg.	36	4	S.S.O	*	*	*	*	*	*	0.01
		y_solve.f	Par.Do.Reg.	52	2	L.I.L	1.34	2.63	3.32	15.21	19.16	20.3	11.6
LU	0.0	ssor.f	Par.Reg.	120	64	L.S	5.8	*	*	*	*	25.2	63.9
		ssor.f	Par.Reg.	120	64	L.I.P	*	*	*	1.9	1.1	2.19	1.13
LU-HP	0.05	jacld.f	Par.Reg	35	*	S.S.O	*	*	*	*	*	*	8.38
		jacld.f	Par.Reg	35	*	L.I.P	1.45	1.78	3.06	3.95	5.8	8.5	7.54
		jacu.f	Par.Reg	35	*	S.S.O	*	*	*	*	*	*	8.51
		jacu.f	Par.Reg	35	*	L.I.P	1.22	1.15	2.16	2.76	5.3	7.4	7.71
		blts.f	Par.Do.Reg	54	16	S.S.O	*	*	*	*	*	5.63	8.64
		blts.f	Par.Do.Reg	54	16	L.I.L	2.08	1.01	1.53	1.52	2.06	3.46	5.3
		blts.f	Par.Do.Reg	54	16	L.S	*	*	*	1	7.54	13.3	20.19
		rhs.f	Par.Reg	34	*	L.S	1.77	6.06	6.89	7.74	5.91	4.11	2.83
SP	1.95	tzetar.f	Par.Do.Reg.	26	*	L.I.L	*	*	1.06	1.4	1.4	1.79	1.87
		z_solve.f	Par.Do.Reg.	35	4	L.I.L	*	2.33	3.4	3.1	2.9	4.0	2.66
		z_solve.f	Par.Do.Reg.	35	4	L.S	4.1	8.42	9.1	12	11.59	11.99	12.3
CG	0.01	cg.f	Par.Reg.	772	2	L.I.P	*	2.57	7.33	20.3	30.2	48.2	53.39
		cg.f	Par.Reg.	772	2	L.S	2.14	3.95	17.6	36.2	49.3	75.8	82.95
EP	0.01	ep.f	Par.Reg.	168	128	L.S	0.05	0.29	0.56	0.36	0.38	0.56	2.11
		ep.f	Par.Do.Reg.	130	2	L.S	*	*	*	0.06	5.21	1.2	6.53
		ep.f	Par.Do.Reg.	130	2	S.S.O	*	*	0.01	0.06	5.2	1.16	6.34
FT	0.01	ft.f	Par.Do.Reg.	226	4	L.I.L	*	1.07	3.44	4.08	12.89	21	33.84
		ft.f	Par.Do.Reg.	226	4	L.S	2.33	5.69	9.27	13.05	17.04	31.3	44.25
IS	0.01	is.c	Bar.Reg.	577	8	L.I.B	1.11	11.5	5.5	6.34	8.36	11.3	10.38
		is.c	Par.Reg.	812	8	L.S	0.42	*	20.9	36.2	56.8	64.8	70.6
MG	0.01	mg.f	Par.Do.Reg.	609	2	L.S	18.2	29.2	35.9	35.9	34.2	16.5	28.2
		mg.f	Par.Do.Reg.	609	2	L.I.L	*	24.0	27.9	29.7	24.7	32.5	6.3

Table 1. Identified OpenMP properties in NPB benchmarks. A '*' indicates that the property was not reported.

'L.I.L' Load Imbalance in parallel loop

'L.I.P' Load Imbalance in parallel region

'L.I.B' Load Imbalance in barrier region

'S.S.O' Parallel region startup and shutdown overhead

'S.C.P' Sequential Computation Property

'L.S.F' Linear speedup failed for the first time

'L.S' Low Speedup

'R.F.L' Region first line number

'F.P.N' Found property name.

The results provided in Table 1 are bound to a particular code region in an application. We have listed some interesting observations for the concerned code regions below:

1. EP showed the highest parallelism. It scaled with the linear speedup till 128 threads whereas most of the other applications such as CG and MG did not scale.
2. The Start Up and Shutdown Overhead property for the parallel regions (S.S.O) showed up only for BT and LU-HP with 128 threads. However, the RFL 130 of EP had the overhead even for 8 threads.
3. Severity due to load imbalance for the applications such as CG and LU-HP kept increasing from 2 threads whereas severity for some code regions, as in MG, decreased for large runs.
4. The Sequential Computation property represents how much of the code in an application is sequential. In most of the NAS benchmarks, severity due to the Sequential Computation is less. The severity of S.C.P for the BT application is comparatively high (2.42 per cent) although the value is negligibly small.
5. Periscope identified a few code regions which have a low speedup. For instance, RFL 120 of ssor.f from LU, RFL 772 of cg.f from CG, RFL 812 of is.c from IS, RFL 226 of ft.f from FT, and so forth. The reason for the low speedup for the LU benchmark is discussed in Section 4.1.2.

4.1.2 NAS LU: Reason for the Low Speedup

It was interesting for us to infer why there is an increased severity for the Low Speedup property on some parts of the code regions of the NAS parallel benchmarks (especially LU and CG) and to check if there really exists a scaling problem for those code regions. To this extent, we investigated in detail the LU benchmark (refer to Table 1).

Severity of the Low Speedup property of RFL 120 of ssor.f from the LU benchmark is 63.9 per cent. This means that the performance problem is rather severe

when compared to the phase region of the LU benchmark. On inspecting the particular code region, we noticed that the piece of code is a parallel region consisting of the master, omp do, and barrier constructs in addition to some function calls. They form the lower triangular part of the Jacobian matrix and perform the lower triangular solution; similarly, they do the same with the upper triangular part. This triangular form which comprises lots of dependent computations weakens the speedup of the code region. The inner parts of the code region and their corresponding severity values for the low speedup property can be seen in Figure 4.

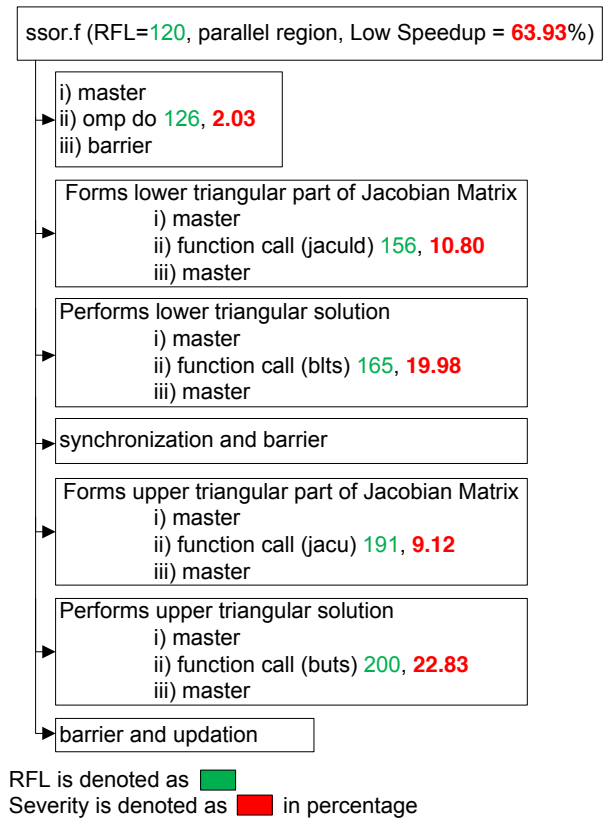

Figure 4. Code regions of the parallel region (RFL 120 of ssor.f) of the LU benchmark and their corresponding Low Speedup property

4.1.3 NAS-LU: Comparison of Class A and Class C

Additionally, we ran the scalability analysis for different workloads of the LU benchmark – LU with Class A ($64 \times 64 \times 64$) and Class C ($162 \times 162 \times 162$). Figure 5 shows

the speedup achieved for the code region with different classes. As can be seen, the LU benchmark with class C scaled well even for the large number of threads.

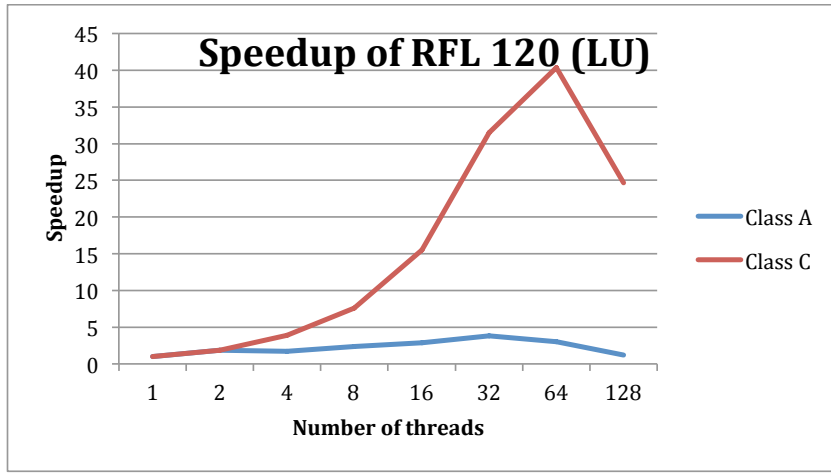

Figure 5. Speedup obtained for the code region (RFL 120 of ssor.f) of the LU benchmark

Table 2 describes the scalability based properties that are identified by Periscope while comparing different classes of LU benchmark. The Linear Speedup Failed for the First Time property showed up at thread number 4 for Class A and at thread number 64 for Class C. Although the linear speedup failed at thread number 4 for Class A, the Speedup Decreasing property appeared only at thread number 64; but the speedup decreased only at 128 threads for Class C. Interestingly enough, the RFL 120 code region always had the Code Region with the Lowest Speedup property with different executions.

Property Name	Threads for Class A	Threads for Class C
Linear Speedup Failed for the First Time	4	64
Speedup Decreasing	64	128
Code Region with the Lowest Speedup	ALL	ALL

Table 2. Comparison of Class A and Class C implementation of the NAS-LU benchmark based on the scalability-based performance properties of Periscope

4.1.4 NAS BT: Phase-Region and Its Effects

Periscope has an option for the user to specify which part of the code is of most interest to him to perform the analysis. By this, the user need not waste his/her

time or computational resources by investigating performance problems on other expendable code regions. The particular code region marked for the analysis is named as a phase region. If the phase region is not marked for analysis, Periscope considers the whole application.

To show the importance of phase region on computationally intensive scientific applications, we have experimented with the BT benchmark, a benchmark with lots of computations. On experimentation, we noticed that the whole experimentation, the scalability and the OpenMP performance analysis for the benchmark took 3.5 hours if the phase region is not included. However, it only needs 19 minutes to do the same operation when phase region is marked for the analysis. The phase region was marked in the core computational part of NAS BT. As expected, the performance data sensed either with or without the phase region remain the same.

5 CONCLUSIONS

In this paper, we have presented two approaches – OpenMP analysis and scalability analysis – for analyzing the performance and scalability issues of OpenMP codes. The two approaches were incorporated with Periscope, an online-based performance analysis toolkit, for detecting the performance bottlenecks of applications in a distributed manner. Additionally, a few modifications were carried out in the functionalities of Periscope entities to attain the goals of the paper. For instance, the Frontend entity of Periscope toolkit was modified to perform several runs automatically so that the scalability analysis is achieved.

In addition, the performance properties relating to scalability and OpenMP analysis were formalized. Those performance properties were utilized by the analysis agents of Periscope to automatically prioritize the performance issues of the code regions when compared with the phase region of the application in terms of severity.

We have tested our implementation for NAS OpenMP benchmarks on a supercomputer named ALTIXmachine at Leibniz Computing Center (LRZ) in Garching. The test results revealed the identified OpenMP and scalability performance properties for the search conducted on NAS OpenMP codes. In addition, we have validated the obtained results.

REFERENCES

[1] Deshmeh, A.—Machina, J.—Sodan, A.: ADEPT Scalability Predictor in Support of Adaptive Resource Allocation. In IEEE IPDPS 2010, DOI: 10.1109/IPDPS.2010.5470430, pp. 1–12.

[2] Marongiu, A.—Burgio, P.—Benini, L.: Supporting OpenMP on a Multicluster Embedded MPSoC. Microprocessors and Microsystems, Vol. 35, 2011, No. 8, pp. 668–682.

[3] Knüpfer, A.—Brunst, H.—Doleschal, J.—Jurenz, M.—Lieber, M.—Mickler, H.—Müller, M. S.—Nagel, W. E.: The Vampir Performance Analysis Tool-Set. In Tools for High Performance Computing, Springer 2008, pp. 139–155.

[4] Basupalli V.—Yuki, T.—Rajopadhye, S.—Morvan, A.—Derrien, S.—Quinton, P.—Wonnacott, D.: ompVerify – Polyhedral Analysis for the OpenMP Programmers. In OpenMP in Petascale Era, LNCS 2011, Vol. 6665, pp. 37–53.

[5] Geimer, M.—Wolf, F.—Wylie, B. J. N.—Abraham, E.—Becker, D.—Mohr, B.: The Scalasca Performance Toolset Architecture. In Concurrency and Computation: Practice and Experience, Vol. 22, 2010, No. 6, pp. 702–719.

[6] Gerndt, M.—Kereku, E.: Search Strategies for Automatic Performance Analysis Tools. Euro-Par 2007, LNCS 2007, Vol. 4641, pp. 129–138.

[7] Lyon, G.—Kacker, R.—Linz, A.: A Scalability Test for Parallel Code. In Software – Prac. and Exp., Vol. 25, 1995, No. 12, pp. 1299–1314.

[8] Intel Thread Profiler. Available at `https://software.intel.com/en-us/articles/intelthread-profiler-product-overview`, 2014.

[9] Wang, J.—Zhang, H.—Wu, R. J.—Yang, L.—Liu, Y. B.: Study of Parallel Algorithm Based on OpenMP in Myocardial Simulations. In Advanced Materials Research, Vol. 204–210, 2011, pp. 1584–1587.

[10] Fürlinger, K.—Gerndt, M.: ompP – A Profiling Tool for OpenMP. In OpenMP Shared Memory Parallel Programming, IWOMP, LNCS 2008, pp. 15–23.

[11] Kluge, M.—Knüpfer, A.—Nagel, W. E.: Knowledge Based Automatic Scalability Analysis and Extrapolation for MPI Programs. In Euro-Par 2005, LNCS 2005, Vol. 3648, 2005, pp. 176–184.

[12] Mustafa, D.—Aurangzeb—Eigenmann, R.: Performance Analysis and Tuning of Automatically Parallelized OpenMP Applications. In OpenMP in the Petascale Era, LNCS 2011, Vol. 6665, pp. 151–164.

[13] Avesani, P.—Bazzanella, C.—Perini, A.—Susi, A.: Facing Scalability Issues in Requirements Prioritization with Machine Learning. In Proc. of the 13th Int. Conf. on Requirements Eng., IEEE 2007, pp. 10–19.

[14] Sarukkai, S. R.—Mehra, P.: Automated Scalability Analysis of MPI Programs. In Proc. of Winter 1995, IEEE, DOI: 1063-6552/95, pp. 21–32.

[15] Benedict, S.—Brehm, M.—Gerndt, M.—Guillen, C.—Hesse, W.—Petkov, V.: Automatic Performance Analysis of Large Scale Simulations. PROPER-EuroPar, Vol. 6043, 2010, pp. 199–207.

[16] Benedict, S.: Threshold Acceptance Algorithm Based Energy Tuning of Scientific Applications Using Energy Analyzer. In ISEC 2014, ACM Publishers 2014, doi 10.1145/2590748.2590759.

[17] Shende, S. S.—Malony, A. D.: The TAU Parallel Performance System. Int. J. of HPC Applications, ACTS Collection Special Issue, Vol. 20, 2006, No. 2, pp. 287–311.

[18] Liu, Y.—Li, M.—Khan, M.—Qi, M.: A MapReduce Based Distributed LSI for Scalable Information Retrieval. Computing and Informatics, Vol. 33, 2014, No. 2, pp. 259–280.

Shajulin Benedict received his Ph. D. degree in grid scheduling from Anna University, Chennai. After his Ph. D. award, he joined a research team in Germany to pursue PostDoctorate under the guidance of Professor Gerndt. Currently, he works as a Professor in SXCCE. He leads HPCCLoud Research Laboratory in India. He is also a Guest Scientist in TUM, Germany. His research interests include grid scheduling, performance analysis of parallel applications, cloud computing and so forth. He is a University Rank Holder for his academic excellence. Currently, he has received two research project grants – one from Germany and one from Department of Science and Technology, India.

Michael Gerndt received his Ph. D. in computer science in 1989 from the University of Bonn. He developed SUPERB, the first automatic parallelizer for distributed memory parallel machines. For two years, in 1990 and 1991, he held a postdoc position at the University of Vienna and joined Research Centre Juelich in 1992 where he concentrated on programming and implementation issues of shared virtual memory systems. This research led to his habilitation in 1998 at Technische Universität München (TUM). Since 2000 he is a Professor for architecture of parallel and distributed systems at TUM. His current research focuses on programming models and tools for scalable parallel architectures. He is leading the development of the automatic performance analysis tools Periscope and of iOMP, an extension of OpenMP for invasive computing. iOMP is a research project in the new Transregional Collaborative Research Center InvasIC (TR 89) funded by the German Science Foundation. In addition he is heading projects on parallel programming languages and their implementation on multicore processors as well as resource management in cloud environments funded by public and industry sources. He is the contact person of the Faculty of Informatics for international affairs.