Automatic Performance Debugging of SPMD-style Parallel Programs

Xu Liu[a,d], Jianfeng Zhan[a,1,*], Kunlin Zhan[c], Weisong Shi[b], Lin Yuan[a,c], Dan Meng[a], Lei Wang[a]

[a]*Institute of Computing Technology, China Academy of Sciences, Beijing 100190, China*
[b]*Department of Computer Science, Wayne State University*
[c]*Graduate University of Chinese Academy of Sciences*
[d]*Department of Computer Science, Rice University*

Abstract

Automatic performance debugging of parallel applications includes two main steps: locating performance bottlenecks and uncovering their root causes for performance optimization. Previous work fails to resolve this challenging issue in two ways: first, several previous efforts automate locating bottlenecks, but present results in a confined way that only identifies performance problems with apriori knowledge; second, several tools take exploratory or confirmatory data analysis to automatically discover relevant performance data relationships, but these efforts do not focus on locating performance bottlenecks or uncovering their root causes.

The simple program and multiple data (SPMD) programming model is widely used for both high performance computing and Cloud computing. In this paper, we design and implement an innovative system, *AutoAnalyzer*, that automates the process of debugging performance problems of SPMD-style parallel programs, including data collection, performance behavior analysis, locating bottlenecks, and uncovering their root causes. AutoAnalyzer is unique in terms of two features: first, *without any apriori knowledge*, it automatically locates bottlenecks and uncovers their root causes for performance optimization; second, it is lightweight in terms of the size of performance data to be collected and analyzed. Our contributions are three-fold: first, we propose two effective clustering algorithms to investigate the existence of performance bottlenecks that cause process behavior dissimilarity or code region behavior disparity, respectively; meanwhile, we present two searching algorithms to locate bottlenecks; second, on a basis of the rough set theory, we propose an innovative approach to automatically uncovering root causes of bottlenecks; third, on the cluster systems with two different configurations, we use two production applications, written in Fortran 77, and one open source code—MPIBZIP2 (`http://compression.ca/mpibzip2/`), written in C++, to verify the effectiveness and correctness of our methods. For three applications, we also propose an experimental approach to investigating the effects of different metrics on locating bottlenecks.

Keywords: SPMD parallel programs, automatic performance debugging, performance bottleneck, root cause analysis, performance optimization

1. Introduction

How to improve the efficiency of parallel programs is a challenging issue for programmers, especially non-experts without the deep knowledge of computer science, and hence it is a crucial task to develop an automatic performance debugging tool to help application programmers analyze parallel programs' behavior, locate performance bottlenecks (in short *bottlenecks*), and uncover their root causes for performance optimization.

Although several existing tools can automate analysis processes to some extent, previous work fails to resolve this issue in three ways. First, with traditional performance debugging tools [13] [40] [36], though data collection processes are often automated, detecting bottlenecks and uncovering their root causes need great manual efforts. Second, several previous efforts can only automatically identify critical bottlenecks *with apriori knowledge* specified in terms of either *the execution patterns* that represent situations of inefficient behaviors [2] [3] [4] [5] or *the predefined performance hypotheses/thresholds* [7] [9] or *the decision tree classification* trained by microbenchmarks [26]. Third, while a lots of existing tools [15] [19] [20] [21] [24] [26] [28] [35] [15] take exploratory or confirmatory data analysis approaches to automatically discovering relationships of relevant performance data, these efforts do not focus on locating performance bottleneck and uncovering their root causes of performance bottlenecks.

The SPMD [43] programming model is widely used for high performance computing [45]. Recently, as an instance [44], Mapreduce-like techniques [49] [46] also pro-

mote the wide use of the SPMD programming model in Cloud computing [45] [47] [48]. This paper focuses on how to automate the process of debugging performance problems of SPMD style programs: including collecting performance data, analyzing application behavior, detecting bottlenecks, and uncovering their root causes, *but not including performance optimization.* To that end, we design and implement an innovative system, *AutoAnalyzer.*

Without human involvement, our tool uses source-to-source transformation to automatically insert *the instrumentation code* into the source code of a parallel program, and divide the whole program into *code regions, each of which is a section of code executed from start to finish with one entry and one exit.* For a SPMD-style parallel program, if we exclude code regions in the master process responsible for the management routines, each process or thread should have similar behavior. At the same time, if a code region takes up a trivial proportion of a program's running time, the performance improvement of the code region will contribute little to the overall performance of the program. From the above intuition, in this paper we pay attentions to two types of performance bottlenecks: bottlenecks that cause process or thread behavior dissimilarity, which we call *dissimilarity bottlenecks*, and bottlenecks that cause code region behavior disparity—significantly different contributions of code regions to the overall performance, which we all *disparity bottlenecks.* After collecting the performance data of code regions from four hierarchies: application, parallel interface, operating system, and hardware, AutoAnalyzer proposes a series of innovative approaches to searching code regions that are dissimilarity and disparity bottlenecks and uncovering their root causes for performance optimization. Our contributions are concluded as follows:

- For SPMD-style parallel applications, we utilize two effective clustering algorithms to investigate the existence of performance bottlenecks that cause process behavior dissimilarity or code region behavior disparity, respectively; if there are bottlenecks, we present two searching algorithms to locate performance bottlenecks.

- On a basis of the rough set theory, we propose an innovative approach to automatically uncovering root causes of bottlenecks.

- We design and implement AutoAnalyzer. On the cluster systems with two different configurations, we use two production applications and one open source code—MPIBZIP2 to verify the effectiveness and correctness of our system. We also investigate the effects of different metrics on locating bottlenecks. our experiment results showed for three applications, our proposed metrics outperforms the cycles per instruction (CPI) and the wall clock time in terms of locating disparity bottlenecks.

The rest of this paper is organized as follows: Section 2 formulates the problem. Section 3 outlines the related work, followed by the description of our solution in Section 4. The implementation and evaluation of AutoAnalyzer are depicted in Section 5 and Section 6, respectively. Finally, concluding remarks are listed in Section 7.

2. Problem Statement

A code region is a section of code that is executed from start to finish with one entry and one exit. A code region can be a function, subroutine or loop, which can be nested within another one. After dividing the whole program into n code regions $CR_{j,\ j=1\ldots n}$, we organize $CR_{j,\ j=1\cdots n}$ as a tree structure with the whole program as the root. According to the definition of the tree structure, for any node CR_j, its depth is the length of the path from the root to CR_j. For example, in Fig.1, the depth of *code region* 1 is one. We call a code region of the depth L an L-code region.

In our system, to accurately measure the contribution of each code region to the overall performance of the program, we require that *code regions that have the same depth can not be overlapped.* For code regions with different depths, we encourage the nesting of code regions because deep nesting leads to fine granularity, which is helpful in narrowing the scope of the source code in locating bottlenecks. For example, in Fig.1, for two 1-code regions, *code region* 1 and *code region* 2 do not intersect. For *code region* 1, its two children nodes: *code region* 4 and *code region* 6 are nested within it.
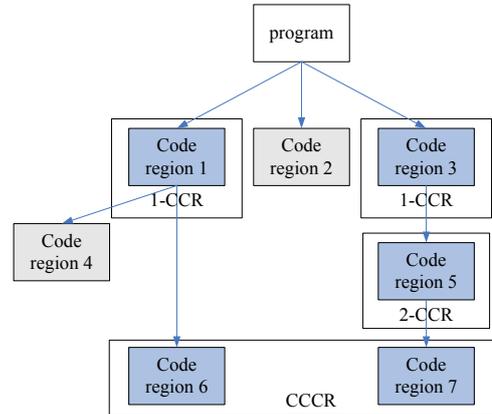

Figure 1: The code region tree of a parallel program.

For a parallel program, if its processes or threads have similar behavior, performance vectors of all processes or threads should be classified into one cluster, or else *there are dissimilarity bottlenecks, indicating load imbalance.* For each code region, if we average its performance data among all processes or threads, we can measure its contribution to the overall performance. We will identify a code region that takes up a significant proportion of a program's running time and has the potential for performance

improvement as a disparity bottleneck. Of course, we can not exhaust all types of bottlenecks, since users hope programs to run faster and faster.

Our work focuses on how to automatically locate dissimilarity and disparity bottlenecks, and uncover their root causes for performance optimization. However, automatic performance optimization is not our target.

3. Related Work

Table 1: The comparison of different systems. Yes indicates it is automatic; else not.

system	data collection	behavior analysis	bottle necks	root causes	optimi zation
HPC Viewer	yes	no	no	no	no
HPC TOOLKIT	yes	no	no	no	no
TAU	yes	no	no	no	no
EXPERT	yes	yes	yes[a]	no	no
Paradyn	yes	yes	yes[b]	no	no
Aksum	yes	yes	yes[c]	no	no
Perf Explorer	yes	yes	no	no	no
Tallent *et al.* [16]	/	/	/	/	yes
Auto Analyzer	yes	yes	yes	yes	no

[a]with apriori knowledge
[b]with apriori knowledge
[c]with apriori knowledge

Table 1 summarizes the differences of the related systems from five perspectives: data collection, behavior analysis, bottleneck detection, uncovering root causes and performance optimization. Hollingsworth et al [32] proposes a plan to develop a test suite for verifying the effectiveness of different tools in terms of locating performance bottlenecks. If it succeeds, this test suite can provide a benchmark for evaluating the accuracy of locating bottlenecks for different tools in terms of the false positive and the false negative. Unfortunately, this project seems ended without updating its web site.

The traditional approach for performance debugging is through automated data collection and visualizing performance data, while performance analysis and code optimization need great manual efforts. With this approach, application programmers need to learn appropriate tools, and rely on their expertise to interpret data and its relation to the code [24] so as to optimize the code. For example, HPCViewer [13], HPCTOOLKIT [40], and TAU [36] display the performance metrics through a graphical user interface. Users depend on their expertise to choose valuable data, which is hard and tedious.

With apriori knowledge, previous work proposes several automatic analysis solutions to identify critical bottlenecks. The EXPERT system [2] [3] [4] [5] describes performance problems using a high level of abstraction in terms of execution patterns that result from an inefficient use of the underlying programming models, and performs trace data analysis using an automated pattern-matching approach [5]. The Paradyn parallel performance tool [7] starts searching for bottlenecks by issuing instrumentation requests to collect data of a set of pre-defined performance hypotheses for the whole program. Paradyn starts its search by comparing the collected performance data with the predefined thresholds, and the instances where the measured value for the hypothesis exceeds the threshold are defined as bottlenecks [9]. Paradyn starts a hierarchical search of the bottlenecks, and refines this search by using stack sampling [11] and pruning the search space through considering the behavior of the application during previous runs [8]. Using a decision tree classification, which is trained by the microbenchmarks that demonstrate both efficient and inefficient communication, Vetter et al [26] automatically classify individual communication operations, and reveal the cause of communication inefficiencies in the application. The Aksum tool [33] automatically performs multiple runs of a parallel application and detects performance bottlenecks by comparing the performance achieved varying the problem size and the number of allocated processors. The key idea in the work of [38] [39] is to extract performance knowledge from parallel design patterns or model that represent structural and communication patterns of a program for performance diagnosis.

Several previous efforts propose exploratory or confirmatory data analysis [24] or fuzzy set method [30] to automated discoveries of relevant performance data. The PerfExplorer tool [19] [20] [21] addresses the need to manage large-scale data complexity using techniques such as clustering and dimensionality reduction, and performs automated discovery of relevant data relationships using comparative and correlation analysis techniques. By clustering thread performance for different metrics, PerfExplorer should discover these relationships and which metrics best distinguish their differences. Calzarossa *et al.* [15] proposes a top-down methodology towards automatic performance analysis of parallel applications: first, they focuses on the overall behavior of the application in terms of its activities, and then they consider individual code regions and activities performed within each code region. Calzarossa *et al.* [15] utilizes clustering techniques to summarize and interpret the performance information by identifying patterns or groups of code regions characterized by a similar behavior. Ahn *et al.* [28] use several multivariate statistical analysis techniques to analyze parallel performance behavior, including cluster analysis and F-ratio, factor analysis, and principal component analysis. Ahn *et al.* [28] show how hardware counters could be used to analyze the performance of multiprocessor parallel machines. The primary goal of the SimPoint system [31] is to reduce long-

running applications down to tractable simulations. Sherwood *et al.* [31] define the concept of basic block vectors, and use those concepts to define the behavior of blocks of execution, usually one million instructions at a time. Truong *et al.* [29] [30] propose a fuzzy set approach to search bottlenecks. However, it does not intend to uncover the root causes of bottlenecks. Tallent *et al.* [16] propose the approaches to measure and attribute parallel idleness and parallel overhead of multi-threaded parallel applications.

Tiwari *et al.* [35] describes a scalable and general-purpose framework for auto-tuning compiler-generated code, which generates in parallel a set of alternative implementations of computation kernels and automatically selects the one with the best-performing implementation. Tu *et al.* [41] [42] propose a new parallel computation model to characterize the performance effects of the memory hierarchy on multi-core clusters in both vertical and horizontal levels. Babu *et al.* [34] make a case for techniques to automate the setting of tuning parameters for MapReduce programs. Zhang *et al.* [27] propose a precise request tracing approach to debug performance problems of multi-tier services of black boxes.

Our system has two distinguished differences from other systems as shown in Table 1: first, in addition to automatic performance behavior analysis, we automatically locate bottlenecks of SPMD-style parallel programs without apriori knowledge; second, we automatically uncover the root causes of bottleneck for performance optimization. With regard to proposing performance vectors to represent behavior of parallel application, AutoAnalyzer is similar to the work in [15] [19] [20] [21] [29], but we investigate the effect of different metrics on locating bottlenecks. Different from PerfExplorer [19] [20] [21], which leverages sophisticated clustering techniques, AutoAnalyzer adopts comparatively simple clustering algorithms, which are lightweight in terms of the size of performance data to be collected and analyzed.

4. Our Solution

This section includes four parts: Section 4.1 summarizes our approach, followed by the description of the approaches to investigating the existence of bottlenecks in Section 4.2. How to locate bottlenecks is given out in Section 4.3. Finally, we propose an approach to uncovering the root causes of bottlenecks.

4.1. Summary of our approach

Our method includes four major steps: instrumentation, collecting performance data, locating bottlenecks, and uncovering their root causes.

First, we instrument a whole parallel program into code regions. Our tool uses source-to-source transformation to automatically insert instrumentation code into the source code, which requires no human involvement.

Second, we collect performance data of code regions. For each process or thread, we collect the following performance data of code regions: (1) application-level performance data: *wall clock time* and *CPU clock time*; (2) hardware counter performance data: *clock cycle, instructions retired, L1 cache miss, L2 cache miss, L1 cache access, L2 cache access*; (3) communication performance data: *MPI communication time*—the executing time in MPI library and *MPI communication quantity*—the quantity of data transferred by the MPI library; (4) operation system level performance data: *disk I/O quantity*—the quantity of data read and written by disk I/O. On a basis of hardware counter performance data, we obtain two derived metrics: *L1 cache miss rate* and *L2 cache miss rate*. For example L1 cache miss rate can be obtained according to the formula—*((L1 cache miss) / (L1 cache access))*.

Third, we utilize two clustering approaches to investigating the existence of bottlenecks. If there are bottlenecks, we use two searching algorithms to locate bottlenecks.

Finally, on a basis of the rough set theory, we present an approach to uncovering the root causes of bottlenecks.

4.2. Investigating the existence of bottlenecks

In this section, we present how to investigate existence of dissimilarity bottlenecks and disparity bottlenecks, respectively.

4.2.1. The existence of dissimilarity bottlenecks

For a SPMD program, each process or thread is composed of the same code regions. If we exclude code regions in the master process responsible for the management routines, the high behavior similarity of each process or thread indicates the balance of workload dispatching and resources utilizing, and vice versa [15]. So we use a similarity analysis approach to investigate the existence of dissimilarity bottlenecks.

The performance similarity is analyzed among all participating processes or threads to discover the discrepancy. We presume that the whole program is divided into n code regions, and the whole program has m processes or threads. In our approach, each process' or thread' performance is represented by a vector $\vec{V_i}$, where i is the process or thread rank. T_{it} represents the performance measurement of the t_{th} code region in the i_{th} process or thread. So $\vec{V_i}$ is described as $\vec{V_i} = (T_{i1}, T_{i2} \cdots, T_{in})$.

We define the Euclidean distance—$Dist_{ij}$ of two vectors $\vec{V_i}$ and $\vec{V_j}$ in Equation(1).

$$Dist_{ij} = \sqrt{(T_{i1} - T_{j1})^2 + \cdots + (T_{in} - T_{jn})^2} \quad (1)$$

We choose the CPU clock time of each code region as the main measurement. Different from the wall clock time, the CPU clock time only measures the time during which the processor is actively working on a certain task, while the wall clock time measures the total time for a process

to complete. We also observe the effect of choosing different metrics—the wall clock time on locating dissimilarity bottlenecks in Section 6.4

On a basis of Equation(1), we present a simplified OPTICS clustering method [1]—Algorithm 1 to classify all processes or threads. We choose the simplified OPTICS clustering method because it has advantage in discovering isolated points. In this approach, the performance vector of each process or thread is considered as a point in an n-dimension space. A set of points is classified into one cluster if the point density in the area, where these point scattered, is larger than the defined threshold. If a point is not included into any clusters, we consider it an isolated point, which is also a new cluster.

Algorithm 1 The simplified OPTICS clustering algorithm **{}**

---

1. **repeat**
2.   select a performance vector $\overrightarrow{V_p}$ not belonging to any clusters.
3.   count=0;
4.   **for** each point $\overrightarrow{V_q}$ $(q \neq p)$ in the n-dimension space **do**
5.   **if** (distance($\overrightarrow{V_p}$, $\overrightarrow{V_q}$) < threshold) **then**
6.   count++;
7.   //We set the threshold as 10% × length ($\overrightarrow{V_p}$).
8.   **end if**
9.   **end for**
10.   **if** $count > count\_threshold$ **then**
11.   confirm that this is a new cluster.
12.   **end if**
13. **until** all vectors are compared.

For a SPMD program, if Algorithm 1 classifies performance vectors of all processes or threads into one cluster, indicating all processes have similar performance behavior, we confirm that there are no dissimilarity bottlenecks, or else there are dissimilarity bottlenecks.

4.2.2. The existence of disparity bottlenecks

For each code region, if we average performance data among all processes or threads, we can measure its contribution to overall performance. We will identify a code region that takes up a significant proportion of a program's running time and has the potential for performance improvement as a disparity bottleneck.

We propose a single normalized metric, named *the code region normalized metric (in short, CRNM)*, as the measurement basis for performance contribution of each code region to the overall performance of the application. For each code region, CRNM is defined in Equation (2):

$$CRNM = \frac{CRWT}{WPWT} \star CPI \qquad (2)$$

In Equation (2), CRWT is *the wall clock time of the code region*; WPWT is *the wall clock time of the whole*

program; CPI is *the average cycles per instruction of each code region*. In Section 6.4, we also investigate the effects of choosing other metrics, e.g., CPI and wall clock time of each code region, on locating disparity bottlenecks.

As shown in Fig.2, the procedure of searching disparity bottlenecks is as follows:

First, for each processes or thread, we obtain the CRNM value of each code region. If a code region is not on the call path in a process or thread, its CRNM value is zero. Since a SPMD program can contain 'if' statements, we obtain the average value of each code region among all processes or threads.

Second, we use a k-means clustering method [12] to classify each code region according to the average CRNM value. We choose the k-means clustering method because it can classify data into k clusters without user providing the threshold value. We define five severity categories: *very high (4), high (3), medium (2), low (1), and very low (0)*. The k-means clustering method finally classifies each code region into one of the severity categories according to its CRNM value.

Third, if a code region is classified into one of severity categories of *very high* or *high*, we consider it as a critical code region (CCR).
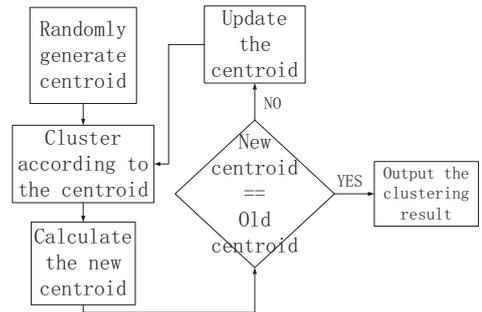

Figure 2: The k-means clustering approach [12].

4.3. Locating bottlenecks

When users confirm there are bottlenecks, they need to locate bottlenecks. We call the code region that is a bottleneck *a critical code region (in short CCR)*. A CCR of the depth L is called an *L-CCR*. If a CCR satisfies the following conditions, we call it *a core of critical code regions (in short, CCCR)*: (1) the CCR is a leaf node in the code region tree; (2) for a CCR, its children nodes are not CCR. For example, in Fig.1, both code region 6 and code region 7 are CCCR.

We propose a top-down searching algorithm—Algorithm 2 to locate dissimilarity bottlenecks as follows:

According to Line 17-26 in Algorithm 2, a CCCR has higher effect on the clustering results than the other children of its parent CCR, and hence we only consider CCCR as dissimilarity bottlenecks, on which users should focus for performance optimization. If the number of clusters or

Algorithm 2 The searching algorithm for dissimilarity bottlenecks**{}**

n: the number of code regions;
r: the number of 1-code region;
m: the number of process or threads;

1. CCR_set=null;
2. CCCR_set=null;
3. **for** each code region j, $j = 1...n$ **do**
4. $T\_backup_{ij} = T_{ij}, i = 1...m$.
5.   **if** its depth is greater than one **then**
6. $T_{ij} = 0, i = 1...m$.
7.   **end if**
8. **end for**
9. Obtain the clustering results.
10. **for** each code region j, $j = 1...n$ **do**
11.   **if** its depth is equal with one **then**
12. $T_{ij} = 0, i = 1...m$.
13.   Obtain the new clustering results.
14.   **if** the clustering result changes **then**
15.   Add code region j into CCR_set.
16.   Recursively analyze children of code region j.
17.   **for** each child code region k **do**
18. $T_{ik} = T\_backup_{ik}, i = 1...m$.
19.   Obtain the new clustering results.
20.   **if** the clustering result does not change **then**
21.   Add code region k into CCR_set.
22.   **if** (CCR k is a leaf node) or (its any child is not a CCR) **then**
23.   CCR k is a CCCR.
24.   **end if**
25.   **end if**
26.   **end for**
27. $T_{ij} = T\_backup_{ij}, i = 1...m$.
28.   **end if**
29.   **end if**
30. **end for**
31. **if** CCR_set is null **then**
32.   Combine s adjacent 1-code regions into composite code regions without overlapping, $s \geq 2$.
33.   Repeat the above analysis.
34.   **if** CCR_set is null and $s < (r-1)$ **then**
35.   increment s and repeat the above analysis.
36.   **end if**
37. **end if**

members of a cluster change, we think the clustering result changes, or else not.

We also propose a simple searching algorithm to refine the scope of disparity bottlenecks as follows:

- If a leaf node j is a CCR, then the code region j is a CCCR.

- For a none-leaf CCR j, if its severity degree is larger than that of each child node, then we consider the code region j as a CCCR.

4.4. Root Cause Analysis

In this section, we introduce the background material of the rough set theory, and present the approaches to recovering roots causes of dissimilarity and disparity bottlenecks, respectively.

4.4.1. The rough set approach [14] [17]

The rough set approach is a data mining method that can be used for classifying vague data. In this paper, we use the rough set approach to uncovering the root causes of dissimilarity and disparity bottlenecks.

We start with introducing some basic terms, including *information system*, *decision system*, *decision table*, and *core*.

An information system is a pair $\Lambda = (U, A)$, where U is is a non-empty finite set of objects, called the universe, and A is a non-empty finite set of attributes such that $a : U \rightarrow V_a$ for every $a \in A$. The set V_a is called the value set of a.

A decision system is any information system of the form $\Lambda = (U, A \cup d)$, where $d \notin A$ is the decision attribute. The elements of A are called conditional attributes.

As shown in Table 2, a decision table is used to describe the decision system. Each entry of a decision table consists of three parts: *object ID, conditional attributions, and decision attribution*. For example, in Table 2, the set of object ID is $\{0, ..., 3\}$, the set of attributions is $\{a_1, ..., a_4\}$, and the set of decisions is $\{N, P\}$.

The core attributions are the attributions that are critical to distinguishing with the decision attributions. How to find the core attributions is a main research field in the rough set approach. One of the solutions is to create *a discernibility matrix* [14] according to the decision table, and then obtain the core attributions using a discernibility matrix as follows:

For a decision system, its decision-relative discernibility matrix is a symmetric $n \times n$ with entries c_{ij} given in Equation 3. Each entry thus consists of the set of attributions upon which x_i and x_j differ [14].

$$c_{ij, \, i,j=1,...,n} = \begin{cases} (a \in A | a(x_i) \neq a(x_j) & \text{if } d(x_i) \neq d(x_j)) \\ (\phi & \text{otherwise }) \end{cases}$$

(3)

A discernibility function f_Λ for the decision table Λ is a Boolean function of m Boolean variables $a_1, a_2, ..., a_m$ defined in Equation 4. For example, for Table 2, the discernibility functions are shown in Equation 5.

$$f_\Lambda(a_1, ...a_m) = \bigwedge \{ \bigvee c_{ij} \mid 1 \le i \le j \le n, c_{i,j} \ne \phi \} \quad (4)$$

Table 2: An example of decision table

ID	a_1	a_2	a_3	a_4	decision
0	sunny	hot	high	False	N
1	sunny	hot	high	True	N
2	overcast	hot	high	False	P
3	sunny	cool	low	False	P

$$\begin{pmatrix} \phi & \phi & a_1 & a_2, a_3 \\ & \phi & a_1, a_4 & a_2, a_3, a_4 \\ & & \phi & \phi \\ & & & \phi \end{pmatrix}$$

Figure 3: The discernibility matrix for the decision table in Table 2.

The core attributions are the same conjunctive terms shared by the discernibility functions of each object, which are defined in Equation 4.

$$\left. \begin{aligned} f_\Lambda(a_1, a_2, a_3, a_4) = &(a_1) \wedge (a_2 \vee a_3) \\ &(a_1 \vee a_4) \wedge (a_2 \vee a_3 \vee a_4) \end{aligned} \right\} \quad (5)$$

According to Equation 5, the same conjunctive terms are $\{a_1, a_2\}$ or $\{a_1, a_3\}$, which are the core attributions of Table 2.

4.4.2. Root cause analysis

For performance optimization, users need to know the root causes of bottlenecks. In this section, we propose the rough set theory based approach to uncovering the root causes of dissimilarity and disparity bottlenecks, and give suggestions for performance improvements.

As shown in Fig.4, we create the decision table for dissimilarity bottlenecks as follows: we choose *the rank of each process* as the object ID. We select *L1 cache miss rate, L2 cache miss rate, disk I/O quantity, network I/O quantity and instructions retired* as five different attributions $a_{k,k=1...5}$.

We take the attribution a_1 (L1 cache miss rate) as an example. For process i, the entry of the decision table corresponding to a_1 is obtained as follows:

For the performance vector $\overrightarrow{T_i}$, where $i = 1...m$, we assign T_{ij} with the L1 cache miss rate of *the j_{th} code region* in *process i*.

After having created the performance vector, we use the simplified OPTICS clustering algorithm to classify performance data. If $\overrightarrow{T_i}$ is classified into *a cluster with the ID of x* according to the approach introduced in Section 4.2.1, for process i, we assign the entry corresponding to a_1 with x.

For *process i*, the decision value is the ID of the cluster into which *process i* is classified according to the metrics of *the CPU clock time*.
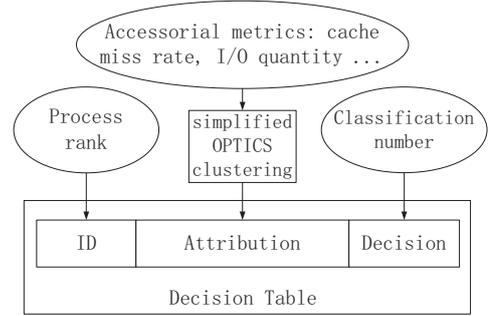

Figure 4: The approach to uncovering the root causes of dissimilarity bottlenecks.

For disparity bottlenecks, we create the decision table as follows:

We use the code region ID to identify each table entry. We also select L1 cache miss rate, L2 cache miss rate, disk I/O quantity, network I/O quantity and executing instruction number as five different attributions.

We take attribution a_1 (L1 cache miss rate) as an example. For *code region j*, the element of the decision table corresponding to a_1 is obtained as follows:

For each code region, we obtain the average L1 cache miss rate in all processes or threads. We use the K-means clustering algorithm to classify the average L1 cache miss rates of each code region into five categories: *very high (4), high (3), medium (2), low (1), and very low (0)*. For *code region j*, if its severity category is higher than *medium*, we assign the entry corresponding to the attribution a_1 with 1, otherwise 0.

For *code region j*, if it is a disparity bottleneck according to the approach proposed in Section 4.2.2, then the decision value is 1, otherwise 0.

After having created the decision table, we obtain the core attributions according to the approaches proposed in Section 4.4.1. Since the core attributions are the ones that have dominated effects on the decision, we consider them as the root causes of disparity bottlenecks.

5. AutoAnalyzer implementation

In order to evaluate the effectiveness of our proposed methods, we have designed and implemented a prototype, AutoAnalyzer. Presently, AutoAnalyzer supports debugging of performance problems of SPMD style MPI applications, written in C, C++, FORTRAN 77, and FORTRAN
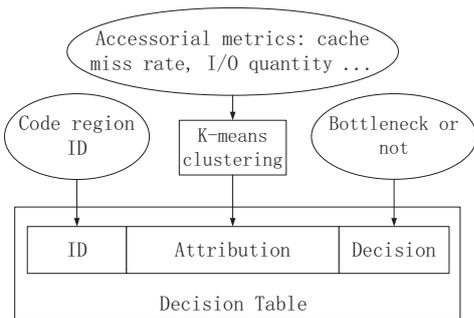
Figure 5: The approach to uncovering the root causes of disparity bottlenecks.

90. We are also extending our work to MapReduce [49] and other data-parallel programming models [46]. Fig. 6 shows AutoAnalyzer architecture.
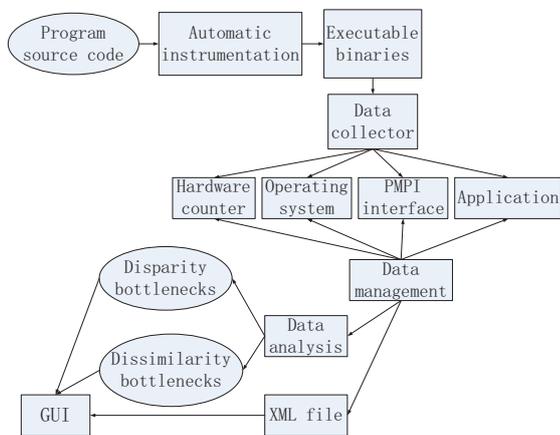

Figure 6: The *AutoAnalyzer* Architecture.

The major components of *AutoAnalyzer* include *automatic instrumentation, data collector, data management, and data analysis.*

Automatic instrumentation. On a basis of OMPi [16]—a source-to-source compiler, we have implemented the source code level instrumentation. Without human involvement, our tool uses source-to-source transformation to automatically insert instrumentation code. After having parsed the program, the system builds the abstract syntax tree (AST). AST shows program's structure information, e.g., the begin and end of *functions, procedures or loops.* With the structure information, our tool can automatically insert instrumentation codes, and divide a program into code regions.

Our tool supports several instrumentation modes: outer loop, inner loop, mathematical library, parallel interface library like MPI, system call, C/FORTRAN library, and user-defined functions or procedures. Without any restrictions on instrumentation, a program can be divided into hundreds or thousands of code regions. For example, after instrumentation, a parallel program of 2,000 lines is divided into more than 300 code regions. This situation has negative influence on the performance analysis because AutoAnalyzer needs to collect and analyze a large amount of performance data. To decrease the size of performance data, we propose two solutions: first, we adopt two rounds of analysis. For the first round, we divide a parallel program into coarse-grained code regions, e.g., per function, for roughly locating bottlenecks; for the second round, we divide *the code regions that are possible bottlenecks* into fine-grained code regions, e.g., loops. Second, users can selectively choose one or more modes to instrument the code, or interact with the GUI of the tool to eliminate, merge, and split code regions.

Data collector. We collect performance data from four hierarchies: application, parallel interface, operating system, and hardware.

In the application hierarchy, we collect the wall clock time and the CPU clock time of each code region. In the parallel interface hierarchy, we have implemented an MPI library wrapper to record MPI routines' behavior of both point-to-point and collective communication. The wrapper is implemented by wrapping the MPI standard profiling interface—PMPI. In the wrapper, we instrumented codes to collect performance data of MPI library, e.g., the executing time and the quantity of data transferred in MPI library.

In the operating system hierarchy, we use systemtap (`http://sourceware.org/systemstap/`) to monitor disk I/O, recording the execution time and quantity of data read and written in I/O operations. Systemtap is based on Kprobe, which is implemented in the Linux kernels. Kprobe can instrument the system calls of the Linux kernel to obtain the executing time and functions' parameters as well as I/O quantity.

In the hardware hierarchy, we use PAPI (`http://icl.cs.utk.edu/papi/`) to count hardware events, including L1 cache miss, L1 cache access, L2 cache miss, L2 cache access, and instructions retired.

Data management. We collect all performance data on different nodes and send them to one node for analysis. All data are stored in XML files.

Data analysis. We analyze performance data of code regions so as to search bottlenecks and uncover their root causes.

Before using AutoAnalyzer, users need to perform the following setup work. Before installing PAPI, they must make sure that the kernel has been patched and recompiled with the PerfCtr or Perfmon patch. Then they can compile the PAPI source code to install it. SystemTap is also dependent upon the installation of several packages: kernel-debuginfo, kernel-debuginfo-common RPMs, and the kernel-devel RPM. Before installing Systemtap, users need to install these packages. However, with the support of state-of-the-practice operating system deployment tool, like SystemImager, which is open source, we can automate the deployment of AutoAnalyzer.

6. Evaluation

In this section, we use two production parallel applications, written in Fortran 77, and one open-source parallel application, written in C++, to evaluate the correctness and effectiveness of AutoAnalyzer.

The first program is *ST*, which calculates the seismic tomography using a refutations method. ST is on the production use in the largest oil company in China. Fig.7 shows the model obtained with *ST*. The second one is a parallel NPAR1WAY module of SAS. SAS is a system widely used in data and statistical analysis. The third one is MPIBZIP2—a parallel implementation of the bzip2 block-sorting file compressor that uses MPI and achieves significant speedup on cluster machines (`http://compression.ca/mpibzip2/`).

In Section 6.1, Section 6.2, and Section 6.3, for three applications we choose the CPU clock time as the main performance measurement for searching dissimilarity bottlenecks, and our proposed CRNM as the main performance measurement for disparity bottlenecks, respectively. In Section 6.4, we investigate the effects of different metrics on locating bottlenecks.

6.1. ST

In this section, we use a production parallel application of 4307 line codes—ST, to evaluate the effectiveness of our system. To identify a problem, a user of our tools does little to start. The tool automatically instruments the code. After analysis, the tool informs the user about bottlenecks and their root causes. For ST, it took about 2 days for a master student in our lab to locate bottlenecks and rewrite about 200 lines to optimized the code.
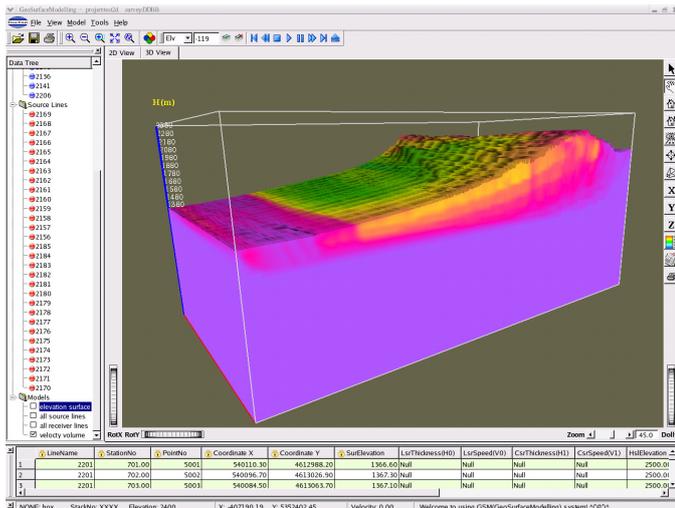

Figure 7: The model obtained with *ST*.

Out testbed is a small-scale cluster system, connected with 1000 Mbps networks. Each node has two processors, each of which is AMD Opteron with $64KB$ L1 data cache,
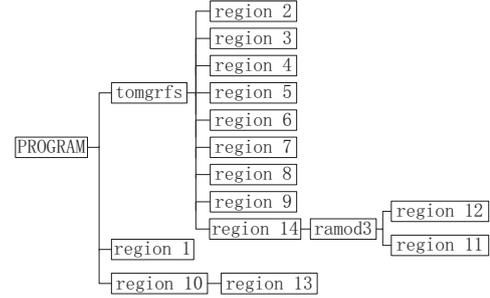

Figure 8: The code region tree of ST. *Code region 11, 12* are in subroutine *ramod*3, which is nested in *code region 14*. All code regions contain loops.

$64KB$ L1 instruction cache, and $1MB$ L2 cache. The OS version is $linux - 2.6.19$.

In the rest of this section, we give the detail of locating bottleneck and optimizing performance. Section 6.1.1 reports a case study of ST with coarse-grain code regions for locating bottlenecks and optimizing application. Section 6.1.2 reports a case study of ST with fine-grain code regions.

6.1.1. Locating bottlenecks and optimizing the applications

To reduce the number of code regions, AutoAnalyzer support an instrumentation mode that allows a user to select whether to instrument *functions or procedures*, or *outer loops*. In this subsection, we instrument ST into 14 coarse-grain code regions, and Fig. 8 shows the code region tree. For ST, a configuration parameter—*the shot number* decides the amount of data input. For this experiment, the shot number is 627.

According to the similarity analysis approach proposed in Section 4.3, AutoAnalyzer outputs the analysis result for each process behavior of ST, which is shown in Fig.9. We can find that all processes are classified into five clusters. For a SPMD program, the analysis results indicate that dissimilarity bottlenecks exist. According to the searching result, we can conclude that *code region 11 and code region 14* are CCRs. Since code region 11 is the child node of code region 14, we consider code region 11 as a CCCR, which is the location of the problem.

We create the decision table to analyze the root causes of *code region 11*.

Table 3 shows the decision table. In the decision table, the attributions $a_{k,k=1,2,3,4,5}$ represents *L1 cache miss rate, L2 cache miss rate, disk I/O quantity, network I/O quantity, and instructions retired*, respectively. Fig.10 shows the discernibility matrix.

Performance similarity
there are 5 clusters of processes
cluster 0: 0
cluster 1: 1 2
cluster 2: 3
cluster 3: 4 6
cluster 4: 5 7
dissimilarity severity, *S*: 0.783958
CCCR: code region 11
CCR tree:
 code region 14 (1-CCR) ---> code region
                    11 (2-CCR & CCCR)

Figure 9: The analysis results of similarity measurement.

Table 3: Decision table for the dissimilarity bottlenecks

ID	a_1	a_2	a_3	a_4	a_5	D
0	0	0	0	0	0	0
1	0	0	0	0	1	1
2	0	0	0	0	1	1
3	1	0	0	0	2	2
4	0	1	0	0	3	3
5	1	1	0	1	4	4
6	1	2	0	1	3	3
7	1	2	0	0	4	4

$$
\begin{pmatrix}
\phi & a_5 & a_5 & a_1,a_5 & a_2,a_5 & a_1,a_2,a_4,a_5 & a_1,a_2,a_4,a_5 & a_1,a_2,a_5 \\
 & \phi & \phi & a_1,a_5 & a_2,a_5 & a_2,a_4,a_5 & a_1,a_2,a_4,a_5 & a_1,a_2,a_5 \\
 & & \phi & a_1,a_5 & a_2,a_5 & a_2,a_4,a_5 & a_1,a_2,a_4,a_5 & a_1,a_2,a_5 \\
 & & & \phi & a_1,a_2,a_5 & a_2,a_4,a_5 & a_2,a_4,a_5 & a_2,a_5 \\
 & & & & \phi & a_1,a_4,a_5 & \phi & a_1,a_2,a_5 \\
 & & & & & \phi & a_2,a_5 & \phi \\
 & & & & & & \phi & a_4,a_5 \\
 & & & & & & & \phi
\end{pmatrix}
$$

Figure 10: The discernibility matrix for Table 3.

According to the approach proposed in Section 4.4.1, we find that a_5 is the core attribution, which indicates that the variance of instructions retired in different processes is the root cause of code region 11.

Fig.11 verifies our analysis, from which we can discover obvious differences of instructions retired of *code region 11* among different processes.

Using the K-means clustering approach, AutoAnalyzer outputs the analysis result for each code region of ST, which is shown in Fig.12. The severity degree of *code region 14, code region 11, code region 8* is larger than medium, respectively. According to the analysis result, we confirm that *code regions 14, code region 11 and code region 8* are CCR. Since code region 11 is nested within code region 14 and the severity degree of code region 11 is the same as code region 14, so code region 11 is a CCCR.
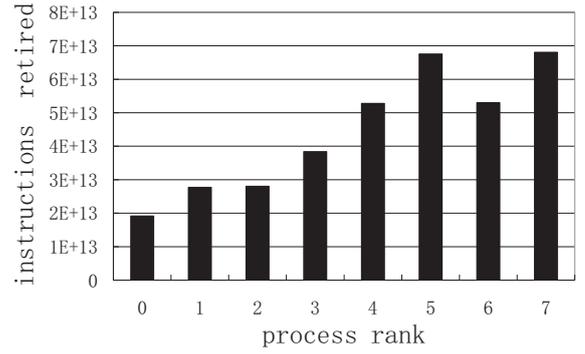

Figure 11: The variance of instructions retired of *code region 11* in different processes.

Since no code region is nested in code region 8, so code region 8 is also a CCCR. We focus on code region 8 and code region 11 for performance optimization.

very high: code regions: 14,11
high: code regions: 8
medium: code regions: 5,6
low: code regions: 2
very low: code regions: 1,9,3,7,10,12,13,4

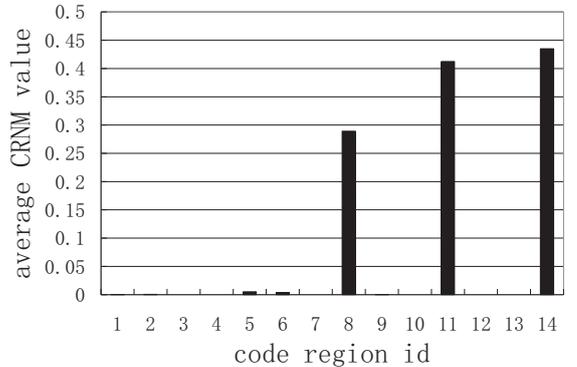Figure 12: The analysis results of the k-means clustering approach.

Figure 13: The average CRNM of each code region.

We analyze the root causes of disparity bottlenecks with the rough set approach. The decision table is shown in Table 4. In the decision table, attribution $a_{k,k=1,2,3,4,5}$ represents L1 Cache miss rate, L2 cache miss rate, disk I/O quantity, network I/O quantity, and instructions retired, respectively.

According to the approach proposed in Section 4.4.1, we find that $\{a_2, a_3\}$ is the core attributions, which indicates high L2 cache miss rate and high disk I/O quantity are the root causes of disparity bottlenecks. Then we search the decision table and find that the root cause of code region 8

10

Table 4: Decision table used for searching disparity bottlenecks.

ID	a_1	a_2	a_3	a_4	a_5	D
1	0	0	0	0	0	0
2	1	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	1	1	0	0	1	0
6	1	0	0	0	1	0
7	0	0	0	0	0	0
8	0	0	1	0	1	1
9	1	0	0	0	0	0
10	1	0	0	0	0	0
11	1	1	0	0	1	1
12	0	0	0	0	0	0
13	0	0	0	0	0	0
14	1	1	0	0	1	1

is high disk I/O quantity and the root cause of code region 11 is high L2 cache miss. From the performance data, we can observe that the disk I/O quantity of code region 8 is as high as $106G$ and the L2 cache miss rate of code region 11 is high as 17.8%.

In order to eliminate the dissimilarity bottleneck—code region 11, we replace the static load dispatching in the master process, adopted in the original program, with a dynamic load dispatching mode. After the optimization, we use AutoAnalyzer to analyze the optimized code again. The analysis results show that all processes, excluding the code regions in the master process responsible for the management routines, are classified into one cluster, indicating that all processes have the similar performance with balanced workloads.

We take the following approaches to optimizing the disparity bottlenecks—code region 8 and code region 11. First, we improve *code region 8* by buffering as many as data into the memory. Second, we improve the data locality of *code region 11* by breaking the loops into small ones and rearranging the data storage.

We use AutoAnalyzer to analyze the optimized code again. The new analysis results show code region 8 is not a disparity bottleneck again, while code region 11 is still a disparity bottleneck, but the average CRNM value of code region 11 decreases from 0.41 to 0.26. The root cause of code region 11 is no longer the high L2 caches miss rate, but the large quantity of instructions retired.

Fig.14 shows the performance of ST before and after the optimization. With the disparity bottlenecks eliminated, the performance of ST rises by 90% in comparison with the original program. With the dissimilarity bottlenecks eliminated, the performance of ST rises by 40% in comparison with the original program. With both disparity and dissimilarity bottlenecks eliminated, the performance of ST rise by 170% in comparison with the original pro-
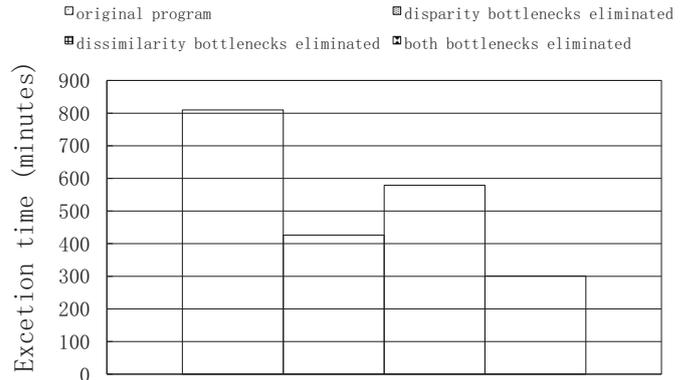
gram.

Figure 14: ST performance before and after the optimization.

6.1.2. A case study of ST with fine-grain code regions

In this subsection, on a basis of the code region tree shown in Fig.8, we divide the program into fine-grained code regions, which is shown in Fig. 15. For saving time, we choose the shot number as 300, and the run time of application is about 9815.52454 seconds. Please note that with the exception of newly added code regions, the same code regions in Fig.8 and Fig. 15 keep the same ID.

We use the simplified OPTICS clustering algorithm to find dissimilarity bottlenecks. From the analysis result, we can find that code region 14, code region 11, and code region 21 are CCRs. Since code region 21 is nested within code region 11 and the latter is also nested within code region 14, we confirm that code region 21 is a CCCR, which is the location of the problem.

From Fig.8 and Fig. 15, we can observe the newly identified dissimilarity bottleneck— code region 21 is nested within code region 11, which is identified as a dissimilarity bottlenecks in Section 6.1.1 when a coarse-grain code region tree is adopted.
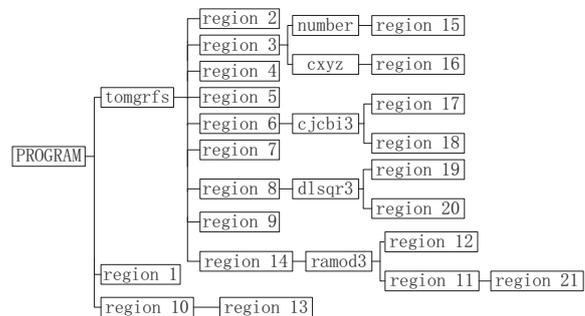

Figure 15: The refined code region tree.

We also use the k-means clustering approach to locating disparity bottlenecks. From the analysis results, we conclude code region 19 and code region 21 are disparity bottlenecks.
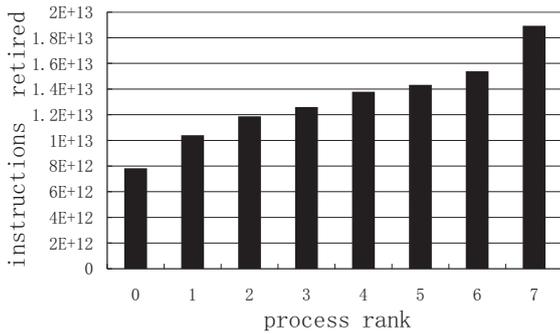
Figure 16: The variance of instructions retired of *code region 21* in different processes.

Figure 17: The average CRNM of each code region in eight processes.

From Fig.8 and Fig. 15, we can observe the newly identified disparity bottlenecks—code region 19 and code region 21 are nested within code region 8 and code region 14, respectively, which are identified as disparity bottlenecks in Section 6.1.1 when a coarse-grain code region tree is adopted. These results show our two-round analysis, introduced in Section 5, indeed can refine the scope of both dissimilarity bottlenecks and disparity bottlenecks. Fig.16 shows the variance of instructions retired of code region 21 in different processes.

6.2. NPAR1WAY

NPAR1WAY is a module of the SAS (Statistical Analysis System) responsible for reading, writing, managing, analyzing, and displaying data. SAS is Widely used in data and statistical analysis. The parallel NPAR1WAY module uses MPI to calculate the exact p-value to achieve high performance. AutoAnalyzer divides the whole program into 12 code regions to separate functions, subroutines, and outer loops.

Out testbed is a small-scale cluster system. Each node has two processors, each of which is a 2 GHz Intel Xeon Processor E5335 with quad cores, $128KB$ L1 data cache, $128KB$ L1 instruction cache, and 8 MB L2 cache. The operating system is Linux 2.6.19.

6.2.1. Bottleneck detection

The analysis results of AutoAnalyzer shows all processes are classified into one cluster, which indicates that no dissimilarity bottleneck exists. AutoAnalyzer also analyzes the application performance from the perspective of each code region. The analysis results show that the severity degrees of code region 3 and code region 12 are larger than medium, and we consider them as CCR. Because there are no nested code regions in code region 3 and code region 12, both of two code regions are CCCRs, which we consider disparity bottlenecks.

We also use the rough set approach to uncover the root causes of disparity bottlenecks. In the decision table, the attributes $a_{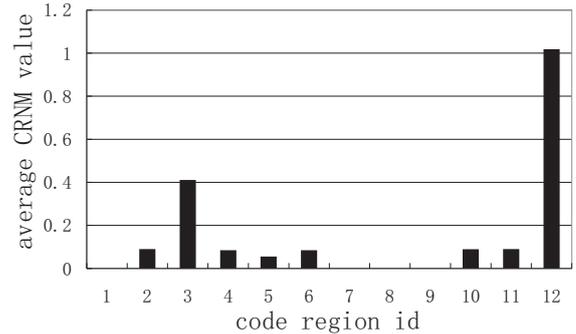k,k=1,2,3,4,5}$ represents L1 cache miss rate, L2 cache miss rate, disk I/O quantity, network I/O quantity, and instructions retired, respectively.

Through analyzing the discernibility matrix, we conclude that $\{a_4, a_5\}$ are the core attributions, which indicates that both high network I/O quantity and high instructions retired are root causes of the disparity bottlenecks. Then we search the decision table and find that code region 3 has high quantity of instructions retired. Meanwhile code region 12 has both high quantity of instructions retired and high network I/O quantity. From the performance data, we can see that instructions retired of code region 3 and code region 12 take up 26% and 60% of the total instructions retired of the program, respectively. At the same time, the network I/O quantity of code region 12 takes up 70% of the total network I/O quantity of the program.

6.2.2. The performance optimization

According to the root causes uncovered by AutoAnalyzer, we optimize the code to eliminate the disparity bottlenecks. The performance of NPAR1WAY rises by 20% after the optimization.

We optimize code region 3 and code region 12 by eliminating *redundant common expressions*. For example, there is one common multiply expression occurring three times in code region 3. We use one variable to store the results of the multiply expression at its first appearance, and later directly use the variable to avoid subsequent redundant computation. In this way, we can decrease massive instructions by eliminating redundant common expressions in deep loops.

Then we analyze the code again. For the optimized code region 3, the analysis results show that the quantity of instructions retired and the wall clock time are reduced by 36.32% and 20.33%, respectively. For the optimized code region 12, the analysis results show that the instructions retired and the wall clock time are reduced by 16.93% and 8.46%, respectively. For code region 12, we fail to eliminate high network I/O quantity.

12

6.3. Analysis of an open source application—MPIBZIP2

MPIBZIP2 is a parallel implementation of the bzip2 block-sorting file compressor that uses MPI and achieves significant speedup on cluster machines. The output is fully compatible with the regular bzip2 data so any files created with MPIBZIP2 can be uncompressed by bzip2 and vice-versa. This software is open source and distributed under a BSD-style license. AutoAnalyzer divides the whole program into 16 code regions to separate functions, subroutines, and outer loops. Fig.18 shows the code region tree. Out testbed is just the same as that in Section 6.2.
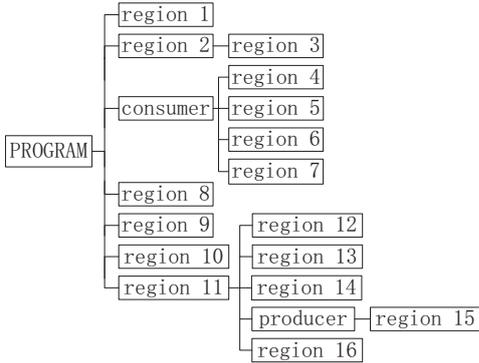
Figure 18: The code region tree of TMPIBzip2.

Excluding the code regions that are responsible for management routines in the master process, we use the simplified OPTICS clustering algorithm to find dissimilarity bottlenecks. From the analysis result, we find all processes are classified into one cluster, and we confirm that there are no dissimilarity bottlenecks in MPIBZIP2. We also use the K-means clustering approach to analyzing the disparity bottlenecks. The analysis results show that the severity degrees of code region 6, and code region 7 are larger than medium, and we consider them as CCR. Since there are no nested code regions in code region 6 and code region 7, both of two code regions are CCCRs, which we consider disparity bottlenecks. Fig.19 shows the average CRNM of each code region of MPIBZIP2.
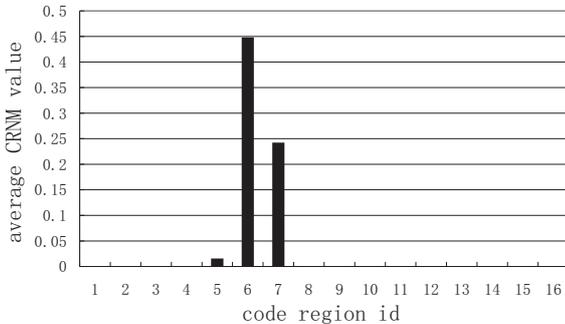

Figure 19: The average CRNM of each code region of MPIBZIP2.

We uncover the root causes of disparity bottlenecks with the rough set approach. In the decision table, the attributes $a_{k,k=1,2,3,4,5}$ represents L1 cache miss rate, L2 cache miss rate, disk I/O quantity, network I/O quantity and instructions retired, respectively. Through analyzing the discernibility matrix, we conclude that $\{a4, a5\}$ are the core attributions, which indicates that network I/O quantity and instructions retired are root causes of the disparity bottlenecks. Then we search the decision table and find that the root cause of code region 6 is high quantity of instructions retired and the root cause of code region 7 is high network I/O quantity. From the performance data, we also observe that instructions retired of code region 6 take up 96% of the total instructions retired of the program. At the same time,the network I/O quantity of code region 7 take up 50% of the total network I/O quantity of the program.

Through reading the source code, we found out that code region 6 calls the $BZ2\_bzBuffToBuffCompress()$ function to compress the data. $BZ2\_bzBuffToBuffCompress()$ is a third-party function and packaged in the static library libbz2.a of bzip2. Code region 7 call $MPI\_Send()$ to send the compressed data to the master process. Those two bottlenecks are difficult to optimize. For the first bottleneck, we need to improve the mature compression algorithm; for the second bottleneck, we need to decrease the data transferred to the master process, however the data has been compressed. We fail to optimize the code.

6.4. Effect of different metrics on bottleneck detections

For three applications, we investigate the effect of different metrics on locating bottlenecks. For ST, NPAR1WAY, and MPIBZIP2, the number of code regions is 14, 12, and 16, respectively. For ST, we perform the experiments on the same testbed as that in Section 6.1, but the shot number is changed from 627 to 300 for saving time. For two other applications, the testbed is the same as that in Section 6.2.

We choose the CRNM value, the CPI, and the wall clock time of each code region as the main performance measurement to locate disparity bottlenecks, respectively.

Our experiment shows CRNM is more valuable than CPI or the wall clock time on locating disparity bottlenecks. For example, for ST, using CRNM, AutoAnalyzer identifies code region 8, code region 11, and code region 14 as CCR, and we significantly improve the application performance through optimizing them, as shown in Section 6.1.1; using the average wall clock time of each code region, AutoAnalyzer identifies code region 2,5, 6, 10 as disparity bottlenecks in addition to code region 8, 11 and 14. From Fig. 20, we can observe code region 2, 5, 6, 10 take up trivial proportion of the running time of the application. Using CPI, AutoAnalyzer identifies code region 2, 8 as disparity bottlenecks, while code region 11 and code region 14, which take up most of the running time of the application, are ignored.
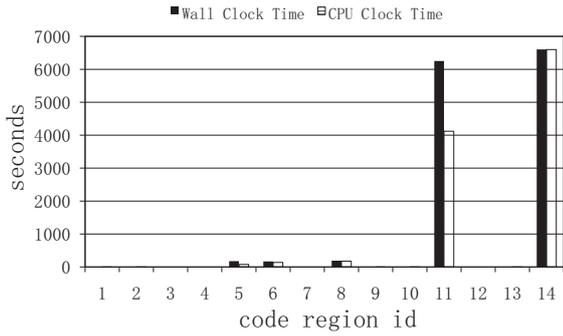
Figure 20: The average wall clock time and CPU clock time of each code region of ST.

Fig.20, Fig.21, and Fig.22 show the average wall clock time and CPU clock time, the average CRNM, and CPI of each code region of ST, respectively.
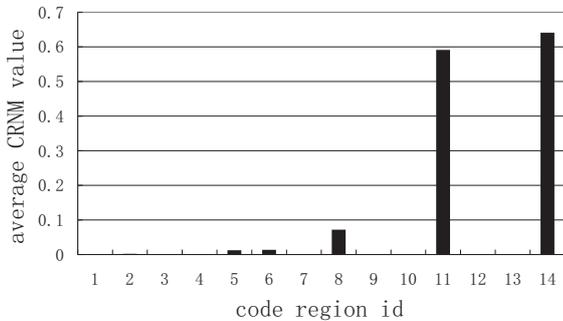

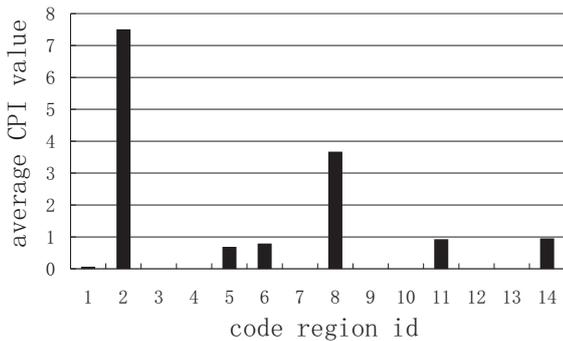Figure 21: The average CRNM of each code region of ST.

Figure 22: The average CPI of each code region of ST.

CRNM is more valuable than CPI or wall clock time on locating disparity bottlenecks because of the following two reasons: first, by using the ratio of *the wall clock time of a code region* to *the wall clock time of the whole program,* our metrics can judge the performance contribution of a code region to the overall performance of a program. Second, CPI measures the efficiency of instruction execution. Derived from the total instructions retired and the total executing cycles, CPI is a basic metric that reflects all

hardware events: cache or TLB miss, cache line invention, pipeline stall caused by data dependency or branches misprediction and so on. So our normalized CPI represents a measurement of the importance of a code region to the overall performance of the application.

We choose the wall clock time and the CPU clock time as the main measurement to locate dissimilarity bottlenecks, respectively. For three applications, we utilize two metrics to locate dissimilarity bottlenecks, respectively. As an example, Fig.20 compares the average wall clock time and the average CPU clock time of each code region of ST, and Fig.23 shows the wall clock time and the CPU clock time of code region 11 of ST in different processes, which is identified as a dissimilarity bottleneck in Section 6.1.1. Though two measurements have some differences, our results show they have the same effects on locating dissimilarity bottlenecks.
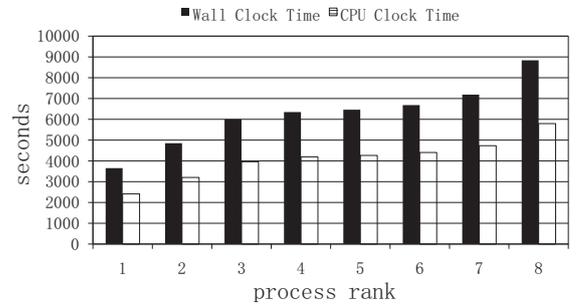

Figure 23: The wall clock time and CPU clock time of code region 11 of ST in different processes.

7. Conclusions

This paper presented a series of innovative methods in automatic performance debugging of SPMD-style parallel programs. For SPMD-style parallel applications, we utilized two effective clustering algorithms to investigate the existence of two types of bottlenecks: dissimilarity bottlenecks that cause process behavior dissimilarity and disparity bottlenecks that cause code region behavior disparity; if there are bottlenecks, we presented two searching algorithms to locate performance bottlenecks. On a basis of the rough set theory, we proposed an innovative approach to automatically uncovering root causes of bottlenecks. We designed and implemented AutoAnalyzer. On the cluster systems with two different configurations, we used two production applications and one open source code—MPIBZIP2 to verify the effectiveness and correctness of our methods. Meanwhile, we also investigate the effects of different metrics on locating bottlenecks, and our experiment results showed for three applications, our proposed metrics—CRNM outperforms CPI and wall clock time in terms of locating disparity bottlenecks; the wall clock time and the CPU clock time have the same effects on locating dissimilarity bottlenecks.

14

In the near future, we will extend our method to more generalized parallel applications beyond the SPMD style.

Acknowledgment

References

[1] M. Ankerst, M. M. Breunig, H. Kriegel, and J. Sander, OPTICS: ordering points to identify the clustering structure. SIGMOD Rec. 28, 2 (Jun. 1999) 49-60.

[2] B. Mohr and F. Wolf. KOJAK, A Tool Set for Automatic Performance Analysis of Parallel Applications. In Ninth Intl. Euro-Par Conference (Euro-Par 2003), Klagenfurt, Austria, August 2003.

[3] F. Wolf, and B. Mohr, Automatic performance analysis of hybrid MPI/OpenMP applications. J. Syst. Archit. 49, 10-11 (Nov. 2003) 421-439.

[4] T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, and J. L. Traff. Knowledge specification for automatic performance analysis: APART technical report, revised edition. Tech. Rep. FZJ-ZAM-IB-2001-08, Forschungszentrum Jǎgùlich GmbH, Aug. 2001.

[5] F. Wolf, B. Mohr, J. Dongarra, and S. Moore, Automatic analysis of inefficiency patterns in parallel applications: Research Articles. Concurr. Comput. : Pract. Exper. 19, 11 (Aug. 2007) 1481-1496.

[6] B. Mohr. OPARI-OpenMP Pragma and Region Instrumentor. Available from < http://www.fz-juelich.de/jsc/kojak/opari/>.

[7] J. K. Hollingsworth, and B. P. Miller, Dynamic control of performance monitoring on large scale parallel systems. In Proceedings of ICS '93 (July.1993) 185-194.

[8] K. L. Karavanic, and B. P. Miller, Improving online performance diagnosis by the use of historical performance data. In Proceedings of SC 09 (Nov. 1999), 42.

[9] H. W. Cain, B. P. Miller, and B. J. Wylie, A Callgraph-Based Search Strategy for Automated Performance Diagnosis. In Proceedings of ICPP 2000 (August 29 - September 01, 2000) 108-122.

[10] A. R. Bernat and B. P. Miller, Incremental call-path profiling. In Technical report, University of Wisconsin, 2004.

[11] P. C. Roth, and B. P. Miller, Deep Start: A Hybrid Strategy for Automated Performance Problem Searches. In Proceedings of the 8th Euro-Par (Aug. 2002) 86-96.

[12] J.A. Hartigan, and M.A. Wong, A k-means clustering algorithm. In Applied Statistics, 28 (1979) 100-108.

[13] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent, HPCVIEW: A Tool for Top-down Analysis of Node Performance. J. Supercomput. 23, 1 (Aug. 2002) 81-104.

[14] J. Komorowski, Z. Pawlak, L. Polkowsk and A. Skowron, Rough sets: A tutorial. Springer-Verlay (1999) 3-9.

[15] M. Calzarossa, L. Massari, and D. Tessera, A methodology towards automatic performance analysis of parallel applications. Parallel Comput. 30, 2 (Feb. 2004) 211-223.

[16] N. R. Tallent, and J. M. Mellor-Crummey, Effective performance measurement and analysis of multithreaded applications. In Proceedings of the 14th PPoPP (Feb.2009) 229-240.

[17] Z. Pawlak, Rough sets. In International Journal of Information and Computer Science, 11(1982) 341-356.

[18] V.V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, A Portable C Compiler for OpenMP V.2.0. In Proc. of the 5th European Workshop on OpenMP (EWOMP03), Aachen, Germany (October 2003).

[19] K. A. Huck, and A. D. Malony, PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In Proceedings of SC 05(Nov. 2005), 41.

[20] K. A. Huck, O. Hernandez, and V. Bui, S. Chandrasekaran, B. Chapman, A. D. Malony, L. C. McInnes, and B. Norris, Capturing performance knowledge for automated analysis. In Proceedings of SC08 (Nov.2008), NJ 1-10.

[21] K. A. Huck, A. D. Malony, S. Shende, and A. Morris, Knowledge support and automation for performance analysis with PerfExplorer 2.0. Sci. Program. 16, 2-3 (Apr. 2008) 123-134.

[22] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr. A Scalable Approach to MPI Application Performance Analysis. In LNCS, 3666 (2005) 309-316.

[23] B. Di Martino, E. Mancini, M. Rak, R. Torella, and U. Villano, Cluster systems and simulation: from benchmarking to off-line performance prediction: Research Articles. Concurr. Comput. : Pract. Exper. 19, 11 (Aug. 2007) 1549-1562.

[24] D. RodrÍlguez, A Statistical Approach for the Analysis of the Relation Between Low-Level Performance Information, the Code, and the Environment. In Proceedings of the 2002 international Conference on Parallel Processing Workshops (August, 2002) 282.

[25] X. Liu, J. Zhan, D. Meng, M. Zou, B. Tu, Similarity Analysis in Automatic Performance Debugging of SPMD Parallel Programs. Workshop on Node Level Parallelism for Large Scale Supercomputers, Co-located with ACM/IEEE SC08.

[26] J. Vetter, Performance analysis of distributed applications using automatic classification of communication inefficiencies. In Proceedings of the 14th ICS (May. 2000) 245-254.

[27] Z. Zhang, J. Zhan, Y. Li, L. Wang, D. Meng, B. Sang, Precise request tracing and performance debugging for multi-tier services of black boxes. In Proceedings of the 39th DSN (June 2009) 337-346

[28] D. H. Ahn, and J. S. Vetter, Scalable analysis techniques for microprocessor performance counter metrics. In Proceedings of SC 02 1-16.

[29] H.-L. Truong and T. Fahringer. SCALEA: a Performance Analysis Tool for Parallel Programs. In Concurrency and Computation: Practice and Experience, 15(11-12) (2003) 1001-1025.

[30] H.-L. Truong, T. Fahringer, Soft Computing Approach to Performance Analysis of Parallel and Distributed Programs. In proceedings of Euro-Par 2005 50-60

[31] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder,Automatically characterizing large scale program behavior. In Proceedings of the 10th ASPLOS (Oct. 2002) 45-57.

[32] J.K. Hollingsworth, M. Steele, Grindstone: A test suite for parallel performance tools, Computer Science Technical Report CS-TR-3703, University of Maryland, October 1996.

[33] T. Fahringer, M. Geissler, G. Madsen, H. Moritsch and C. Seragiotto. On using Aksum for semi automatically searching of performance problems in parallel and distributed programs. In Proceedings of 11th PDP (2003) 385-392.

[34] S. Babu, Towards automatic optimization of MapReduce programs. In Proceedings of 1st SoCC (2010) 137-142.

[35] A. Tiwari, C. Chen, J. Chame, M. Hall, J. K. Hollingsworth, A Scalable Autotuning Framework for Compiler Optimization ,IPDPS 2009 (May. 2009).

[36] S. S. Shende, and A. D. Malony, The TAU parallel performance system. In The International Journal of High Performance Computing Applications, 20, 2 (2006) 287-311,.

[37] K. A. Huck, and A. D. Malony. Performance forensics: knowledge support for parallel performance data mining. http://ix.cs.uoregon.edu/ khuck/papers/parco2007.pdf.

[38] L. Li, and A. D. Malony, Knowledge engineering for automatic parallel performance diagnosis: Research Articles. Concurr. Comput. : Pract. Exper. 19, 11 (Aug. 2007) 1497-1515.

[39] L. Li and A. D. Malony. Automatic Performance Diagnosis of Parallel Computations with Compositional Models. In Proc. IPDPS 07 (2007).

[40] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J.

Mellor-Crummey, and N. R. Tallent, HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurr. Comput. : Pract. Exper. 22, 6 (April 2010) 685-701.

[41] B. Tu, J. Fan, J. Zhan, and X. Zhao, Performance analysis and optimization of MPI collective operations on multi-core clusters, The Journal of Supercomputing, DOI: 10.1007/s11227-009-0296-3, online.

[42] B. Tu, J. Fan, J. Zhan, and X. Zhao, Accurate Analytical Models for Message Passing on Multi-core Clusters, In Proceedings of the 17th PDP (Feb 2009) 133-139.

[43] F. Darema, SPMD model: Past, present and future, Recent Advances in Parallel Virtual Machine and Message Passing Interface: Eighth European PVM/MPI Users' Group Meeting, Santorini/Thera, Greece, 2001.

[44] S. Pallickara, J. Ekanayake, and G. Fox, Granules: A Lightweight Runtime for Scalable Computing with Support for Map-Reduce, Cloud Computing and Software Services: Theory and Techniques: CRC Press (Taylor and Francis), 07/2010.

[45] J. Ekanayake, S. Pallickara, and G.C. Fox, Performance of Data Intensive Supercomputing Runtime Environments , Bloomington, IN, Indiana University, 08/01/2008.

[46] P. Wang, D. Meng, J. Han, J. Zhan, B. Tu, X. Shi, and L. Wan, Transformer: A New Paradigm for Building Data-Parallel Programming Models. IEEE Micro 30, 4 (July 2010) 55-64.

[47] L. Wang, J. Zhan, W. Shi, Y. Liang, and L. Yuan, In cloud, do MTC or HTC service providers benefit from the economies of scale?. In Proceedings of the 2nd MTAGS (2009). 10 pages.

[48] L. Wang, J. Zhan, W. Shi, and Y. Liang, In Cloud, Can Scientific Communities Benefit from the Economies of Scale? Accepted by IEEE Transaction on Parallel and Distributed Systems. March, 2011.

[49] J. Dean and S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Communications of the ACM, 51 (January 2008) 107-113.

[50] X. Hu, N. Cercone, Learning in Relational Databases: a Rough Set Approach, Computational Intelligence, 2(1995) 323-337

[51] X. Liu, Y. Lin, J. Zhan, B. Tu, D. Meng, Automatic Performance Debugging of SPMD Parallel Programs. Technical Report, Institute of Computing Technology, Chinese Academy of Sciences. http://arxiv.org/abs/1002.4264