2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

Profiling HPC Applications with Low Overhead and High Accuracy

Jingyuan Zhao[†$], Xin Liu[‡], Yao Liu[†$], Penglong Jiao[†], Jinshuo Liu[§], Wei Xue[¶]
[†]School of Data Science & Engineering, East China Normal University, Shanghai, China
[‡]National Research Centre of Parallel Computer Engineering and Technology, Beijing, China
[§]School of Cyber Science and Engineering, Wuhan University, Wuhan, China
[$]Shanghai Key Laboratory of Multidimensional Information Processing, East China Normal University, Shanghai, China
[¶]Department of Computer Science and Technology, Tsinghua University, Beijing, China

patrickzhao10@163.com, yyylx@263.net, liuyao@cc.ecnu.edu.cn,
pljiao.cs@gmail.com, liujinshuo@whu.edu.cn, xuewei@tsinghua.edu.cn

***Abstract*—As the parallel scale of HPC applications represented by earth system models becomes larger and the computing cost becomes higher, the performance of HPC applications is increasingly critical. Profiling HPC applications accurately helps to model the applications and find the performance bottlenecks. However, due to the complexity of HPC applications, the diversity of programming languages, the differences of individual programming habits, and multiple architectures, accurate profiling becomes very tough. In this paper, we propose LPerf: a low-overhead and high-accuracy profiler for HPC applications. To reduce the profiling overhead and improve the profiling accuracy, we propose a preprocessing method which can automatically instrument with tunable granularity thus significantly reducing the run-time overhead of profiling, an aggregated caller-callee relationship which is used to locate relationship of functions efficiently, and a profiling-aware method which can precisely calculate running time of functions. The experimental results show that the error rate of profiling reaches 0.02%, and the overhead reaches 1.6%, in the earth system model named CAS-ESM. Compared with the baselines, the precision, accuracy, and overhead of LPerf have reached the state of the art.**

***Index Terms*—LLVM, instrumentation, profiling, performance modeling**

I. Introduction

The performance of HPC applications becomes more crucial due to the increasing complication of tasks. Complicated applications need to be run on the high-performance computing platform because the platform utilizes parallel technology to accelerate computation. However, even though the computing power of high-performance computing platforms is developing rapidly, the platform utilization of HPC applications is decreasing. The peak performance of HPC applications does not even reach 20% of the peak performance of the platforms. This shows that the performance of HPC applications has great potential for improvement. An important indicator of the performance of applications is running time. The reduction of running time not only makes some time-critical tasks possible, but also lowers the computing cost. Therefore, the reduction of the running time of the applications is a matter of great concern. Accurate profiling is helpful for performance modeling and mining performance bottlenecks. However, due to the increasing complexity of tasks, accurate profiling is a huge hurdle. The difficulty of accurate profiling can be caused by many different factors across many different levels from complex operating systems to various code of applications. So, how to accurately profile has become an important research topic. In this paper, we propose LPerf for profiling HPC applications which can significantly lower the overhead and enhance the accuracy. For the HPC applications with multiple MPI communicators, a three-level profiling scheme is proposed to distinguish and profile different model components according to the MPI communicators. The major contributions of this paper are summarized as follows.

- A preprocessing method which can automatically instrument with tunable granularity thus significantly reducing the run-time overhead of profiling during the compilation stage is addressed, to lower the overhead and improve the applicability of LPerf.
- An aggregated caller-callee relationship which is used to locate relationship of functions efficiently is designed. The aggregated caller-callee relationship reduces memory and time overhead, and the red-black tree is adapted to further accelerate locating.
- To obtain exclusive time of functions, a profiling-aware method which can precisely calculate running time of functions is proposed. This method can distinguish the running time of functions with probes and without probes.

The rest of the paper is organized as follows. Section 2 lists preliminaries for profiling and LLVM. Section 3 describes the difficulties and challenges of profiling. Section 4 introduces the details of implementation. Section 5 analyzes the experimental results. Section 6 concludes the paper.

Jingyuan Zhao, Xin Liu and Yao Liu contributed equally to this work. Corresponding author: Yao Liu (liuyao@cc.ecnu.edu.cn) and Wei Xue (xuewei@tsinghua.edu.cn).

978-0-7381-2646-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00180

II. Preliminaries for Profiling and LLVM

A. Preliminaries for Profiling

To accurately profile, there have been some tools, and most of them propose methods of profiling by instrumentation. Instrumentation can be performed at different stages. One is to instrument at the stage of object code [7]. For this method, performing analysis to record the location of instrumentation is necessary. However, due to the incomplete grammatical and semantic information in the object code, the locating and analysis of programs are inaccurate occasionally [6]. The other is to instrument at the stage of source code. This method requires lexical, syntax, and semantic analysis of source code, and the analysis requires a large amount of work and has some modifications for different programming languages [18]. Due to the complicated analysis of source code, this method is not widely applicable.

At the stage of object code, there are some classic tools. The gprof is a profile tool of GNU for C and Fortran applications which is widely applied to performance analysis [15]. It has the ability to record running time of functions, but the timing result is inaccurate occasionally [9] [28]. The GPerftools proposed by google can obtain high-accuracy profiling results, but barely support the profiling of MPI functions.

At the stage of source code, there are also some powerful tools. The LIKWID designed by Jan Treibig can record the running time of each thread, but it cannot record the running time of MPI functions, and can only be applied to the x86 platform [26]. The Timemory designed by Jonathan R. Madsen can record the running time of MPI functions, but only support C++ applications [21]. The HPCtoolkit designed by Adhiantol can support the applications written by many kinds of languages, but the timing result is inaccurate occasionally [1]. The TAU (Tuning and Analysis Utilities) designed by Sameer is a comprehensive toolkit for performance analysis of applications, which reaches the state of the art. However, the problem of overhead hinders TAU [25].

LLVM releases a new chance to perform instrumentation at the stage of intermediate files (IR files) to avoid the defects of instrumentation at the stage of object code and source code [17].

B. Introduction to LLVM

LLVM is a collection of modular, reusable compilers and tools. Clang and Flang are compilers that support C and Fortran applications in LLVM. Together with LLVM, they form a complete tool chain, support multiple operating systems and hardware architectures, and can replace the GCC compilation system [24].

The Pass framework of LLVM which is a supplement to the compiler is an important part of LLVM [8]. They have the ability to analyze, optimize and transform the intermediate files (IR files). When the user-defined Passes are implemented, they will be compiled into a dynamic-link library through specific compilation instructions, and the optimizer of LLVM is used to make the dynamic-link library to process the IR files generated by the source file through LLVM [29]. Sequentially, analysis, optimization or transformation to the IR file will be implemented and a new IR file will be generated. By performing instrumentation at the stage of IR files, we can avoid not only the occasional inaccuracy of locating and analyzing program caused by performing instrumentation at the stage of object code, but also the complicated analysis caused by performing instrumentation at the stage of source code [14].

To reduce the overhead and improve the accuracy, a profiler based on LLVM referred as to LPerf is proposed, which can preprocess and automatically instrument with tunable granularity, locate call relationships of functions with low redundancy, and computing the running time of functions with profiling-aware [2].

III. Challenges and Overview of LPerf

In this section, we elaborate the challenges and introduce the overview of LPerf.

A. Challenges

To record the running time of functions, it is necessary to insert probes into the entry and exit of functions. As the structures of the applications are very complicated, it is tough to automatically instrument [20]. Some existing tools perform manual instrumentation instead of automatic instrumentation.

Profiling precisely is also a hurdle because the running time of some functions are too small to be recorded. However, if these functions are called many times, the sum of the running time of these functions will be very large [16]. If the total running time of these functions cannot be recorded, it will cause a huge error to the profiling result.

Due to multi-level calls, the running time of a function includes not only its own running time, but also the running time of its subfunctions [10]. To accurately analyze function performance, exclusive time, i.e., its own running time, should be focused. Only inserting probes into the entry and exit of functions cannot obtain exclusive time of functions [12]. Therefore, how to achieve exclusive time of a function is a challenge.

Call relationships are the key to achieve exclusive time. Complete call relationships are large and redundant thus leading to locating insufficiently [27]. Locating the call relationships is inevitable as the function with the same call relationships may appears many times and the redundant call relationships hinder locating. Briefly, the probe will bring overhead to the source program. On the one hand, the call relationships will increase the overhead of memory. If the scale of application is very large, the call relationships will become a great burden in the memory. On the other hand, the locating will increase the overhead of running time. If the running time of the probe cannot be reduced, the running time of the instrumented applications will increase dramatically [3].

B. Overview of Proposed LPerf

To profile the large-scale HPC applications, such as CAS-ESM on the specified scientific computing platform, LPerf is designed. The computing platform is equipped with X86-based Hygon CPUs without GPUs or DCUs. Considering the similar performance of CAS-ESM of MPI version and MPI+OpenMP version, to pursue low overhead, high accuracy and precision, LPerf focuses on the resources assignment among processes.

Generally, LPerf includes three phases of preprocessing, running and analyzing, as shown in Figure 1.

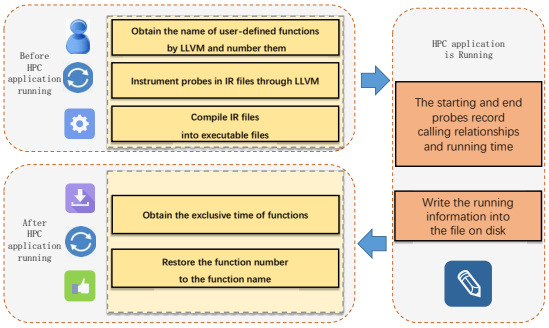

Fig. 1. Overall flow of the proposed LPerf

Before the application runs, the IR files of applications are generated and a Pass is used to scan the IR file through LLVM. After the scan is completed, we obtain the names and numbers of all user-defined functions including the MPI Wrapper functions.

Especially, to take over the built-in MPI functions, MPI Wrapper technology is used to encapsulate the built-in MPI functions, whose names begin with "MPI_". According to the MPI standard, each MPI operation can be implemented by the functions beginning with "MPI_" and "PMPI_". Considering this feature, a set of MPI Wrapper functions with the same names as the built-in MPI functions is created. In the MPI Wrapper functions, PMPI_ functions are called to implement the specific MPI operation. Through MPI Wrapper technology, the MPI Wrapper functions take over the built-in MPI functions.

Then, to generate an instrumented intermediate file, the corresponding function numbers and starting probe after the entry of functions, and the ending probe before the exit of functions are inserted into the functions by the other Pass. After that, we transform the instrumented intermediate file into an executable file through LLVM.

When the application is running, the starting probe is used to record two timestamps. The first is the starting time of the running time of the function excluding the probe. The second is the starting time of running time of the function including the probe. Besides, the function number is also recorded. The ending probe is used to record the call relationships of functions, calculate the running time of the function, accumulate the number of calls and running time of functions. All the information will be recorded in each process. Before the application stops, the customized function inserted at the end of the MAIN function will write the running information in memory into profiling result files on disk.

After the application runs, LPerf traverses and calculates the profiling result files of the selected process which usually is rank 0, to obtain the exclusive time of a function by subtracting the sum of the running time of subfunctions from the running time of the function according to Equation 1. Figure 2 shows the components of the running time of functions. Finally, LPerf restore the function number of each function to the function name. The time complexity of LPerf is O(nlogn).

$$T_e = T_r - \sum T_{rs} \tag{1}$$

where T_e is the exclusive time of a function, T_r is the running time of the function, T_{rs} is the running time of one subfunction of the function.

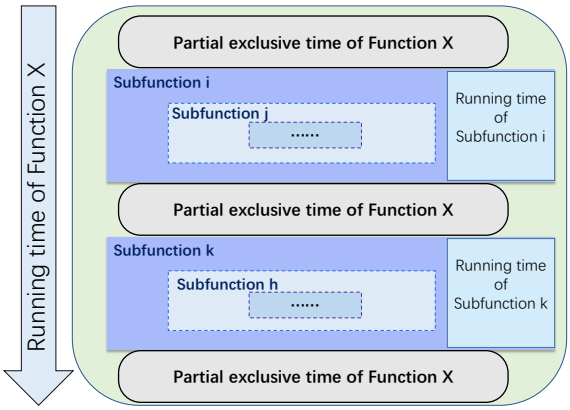

Fig. 2. The components of running time of functions

IV. Methodology

In this section, we present the implementation details of the preprocessing method which can automatically instrument with tunable granularity thus significantly reducing the runtime overhead of profiling, the aggregated caller-callee relationship which is used to locate call relationships of functions efficiently, and the profiling-aware method which can precisely calculate running time of functions.

A. Preprocessing and Automatically Instrumenting with Tunable Granularity

LPerf prefetches the functions information and instruments automatically at the stage of compilation. Figure 3 shows the overall flow of preprocessing and instrumenting.

LLVM frontend, such as Clang and Flang, transforms the source file into the intermediate file (IR file). Optimizer modifies the IR file by Pass Framework [4]. In this paper, two customized Passes which extract function names from IR files and instrument IR files respectively are implemented. The former is used to obtain the function names and encode them, which helps to speed up the functions locating. The latter is

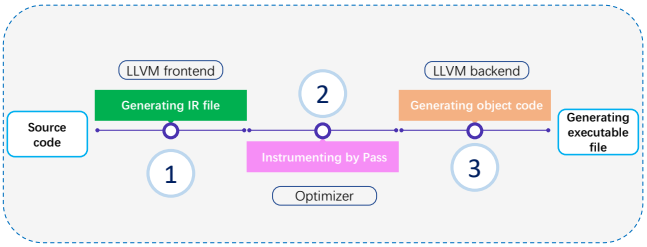

Fig. 3. Overall flow of the preprocessing and instrumenting

used to insert the function number and starting probe after the entry of functions and the ending probe before the exit of functions. After that, LPerf transforms the instrumented IR files into the object files through the LLVM backend. Figure 4 describes the position and content of instrumentation.

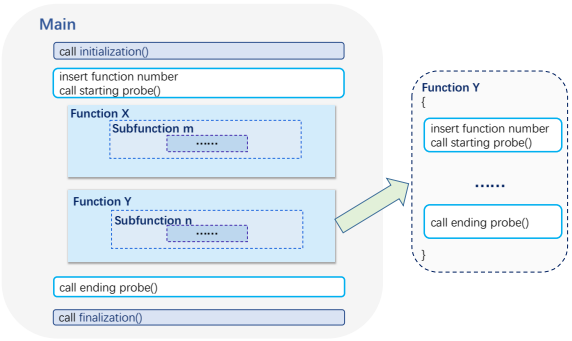

Fig. 4. The position and content of instrumentation

Furthermore, for performance modeling, according to Occam's razor principle, if the predictive accuracy of the two models is similar, a simpler model should be used. Even with a smaller number of features, the approximate predictive accuracy can also be obtained. If the running time of some functions with a few instructions are ignored, there will be few impacts on the results of performance modeling. To balance the overhead and accuracy, LPerf is designed to instrument with tunable granularity. When the customized Pass is used to scan and obtain the name of a user-defined function, the number of instructions of the functions in the IR file will be counted. When instrumenting through the other customized Pass, users can determine that the functions whose numbers of instructions in IR files are lower than a user-defined threshold are not instrumented to reduce the overhead for performance sampling. Therefore, users can balance precision and efficiency by themselves.

B. Efficiently Locating Call Relationships with Low Redundancy

Recording the call relationships of functions according to source code possibly misses the call relationships caused by the function pointer. Therefore, LPerf records the call relationships of functions in the starting and ending probes instrumented at the stage of compilation during runtime. This method has the advantage of recording all the call relationships but brings more overhead to the profiler. To diminish the negative impact, an efficient traversal algorithm to locate call relationships of functions with low redundancy is required.

To obtain the exclusive time, the call relationships and running time of functions are required to record. The call relationship from the main function to the current function is defined as complete call relationships. However, complete call relationships of functions will generate a large redundancy and lead to a large memory overhead. Meanwhile, when a function with the same call relationship is called for the second time, locating the function to record the running time will be slowly in probe. Therefore, the caller-callee relationship of the current function and the caller function is recorded instead of the complete call relationships. LPerf reserves the running time of functions and running time of subfunctions. By the caller-callee relationship, the running time of the function excluding the running time of subfunctions will be calculated and obtained. Actually, the caller-callee relationship is regarded as the primary key. Due to the decrease of redundancy, locating the call relationships and recording the running time of the functions become dramatically fast. The design of aggregated caller-callee relationship not only reduces memory overhead, but also enables faster localization. To further speed up the traversal procedure, the red-black tree is adaptive to store call relationships. The detail of the red-black tree is shown in Figure 5. In addition, each process will maintain a red-black tree and record the running time of functions. Generally, the profiling result of rank 0 is selected to analyze.

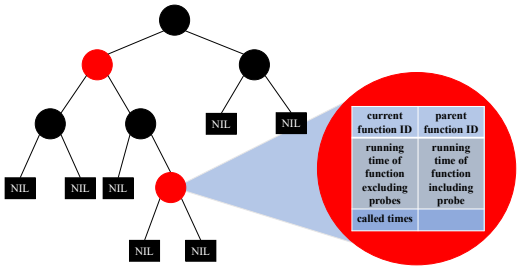

Fig. 5. The red-black tree adapted in the traversal algorithm

C. Profiling-aware Method of Calculating the Running Time of Functions

Although we have proposed some methods to reduce overhead, the overhead caused by instrumentation is still unavoidable [11]. To improve the accuracy of profiling, the profiling-aware method of calculating the running time of functions is proposed.

As the running time of the probe will be recorded and affect the accuracy of the exclusive time of a function, the running time of the starting probe and ending probe should be subtracted from the exclusive time of a function [22]. The precise exclusive time is the running time of functions excluding the probe subtract the sum of the running time of subfunctions including the probe.

To obtain the running time of functions including the probes, the timestamp recorded at the end of the ending probe subtracts the timestamp recorded at the beginning of the starting probe. To obtain the running time of function excluding the probes, the timestamp recorded at the beginning of the ending probe subtract the timestamp recorded at the end of the starting probe. Figure 6 shows the components of the running time of functions.

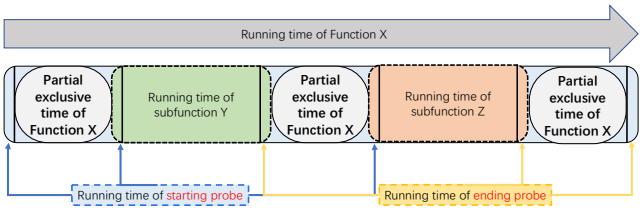

Fig. 6. The components of running time of functions

Additionally, to improve the precision of profiling, the "rdtsc" which is a system timer with low overhead is used to record the running time at cycle level. During the running time of instrumented applications, the caller-callee relationships and two kinds of running time will be recording by starting and end probes, and the running information will be stored into the temporary files on disk before the application ends [5]. At the last stage which is data processing, the precise exclusive time will be obtained by subtracting the running time of probes and stored into the profiling files.

D. Implementation of Starting and Ending Probe

The starting and ending probe record four timestamps, i.e., two timestamps of entering and exiting the probe and two timestamps of entering and exiting the probe. Considering the multi-level function calls, the mentioned timestamps are stored in the stack with running time of probes and the stack without running time of probes respectively. Besides, a stack with function numbers is required to store function numbers.

1) Starting probe: The starting probe records two timestamps of entering and exiting the probe. The first is the starting time of the running time of the function excluding the probe. The second is the starting time of running time of the function including the probe [5].

The corresponding pseudocode is shown as Algorithm 1. In algorithm 1, *starting_time_with_probe* is the starting time of the running time of the function including the probe. *running_time_stack* is the stack with the running time of probes.*function_numbers_stack* is the stack with function numbers. *starting_time_without_timer* is the starting time of the running time of the function excluding the probe.

2) Ending probe: The ending probe records two timestamps of entering and exiting the probe. The first is the ending time of the running time of the function excluding the probe. The second is the ending time of the running time of the function including the probe.

Pop the timestamp from the stack without running time of probes, which is the second timestamp of the starting probe. To obtain the running time of function excluding the starting and ending probes, the first timestamp of the end probe subtract the second timestamp of the starting probe [19].

Algorithm 1 Starting Probe.

Output: *starting_time_without_probe*;

1: record the current time in *starting_time_with_probe*
2: put *starting_time_with_timer* into *running_time_stack*
3: put the function number into *function_numbers_stack*
4: record current time in *starting_time_without_probe*

Then, the ending probe pops the top 2 function numbers in the stack with function numbers, which are the current function number and the caller function number.

After that, pop the stack with the running time of probes, which is the first timestamp of the starting probe. To obtain the running time of function including the starting and ending probes, the second stamp of the ending probe subtract the first timestamp of the starting probe [30]. Finally, the current function number and the caller function number are set as the primary key to find whether the function has been recorded in the red-black tree of function information.

If it exists, accumulate and update the number of calls, the running time of the function excluding the probe, and the running time of the function including the probe respectively. If it does not exist, insert the call relationships and running time of functions into the red-black tree of function information.

The corresponding pseudocode is shown as Algorithm 2. In Algorithm 2, *ending_time_excluding_probes* is the ending time of the function excluding probes. *running_time_excluding_probes* is the running time of function excluding probes. *running_time_including_probes* is the running time of function with probes. *ending_time_including_probes* is the ending time of the function including probes. *called_times* is the times of the current function called by the caller function.

V. Experiments

To verify the effectiveness of LPerf, a series of experiments are designed. A benchmarking application is selected as the test application to verify that LPerf can accurately measure the running time of application with less overhead, and an earth system model is selected as the test application to verify the effectiveness of LPerf for complex applications [23].

A. Setup

1) Experimental platform: The Hygon cluster which has 600 computing nodes is selected as the experimental platform. The specific configuration of the Hygon cluster is shown in Table 1. Every computing node is equipped with an X86-based Hygon CPU which integrates 32 cores. The parallel storage system of the Hygon cluster consists of 4 Opara nodes and 36 Ostor nodes. And, Intel OPA 100Gb is utilized by Hygon as a computing network.

In our experiments, 64 processes are used. In addition, to ensure the fairness of comparison with the baselines, the threshold of instrumentation granularity is fixed at one, that is, all functions are profiled.

Algorithm 2 Ending Probe

Input: *starting_time_without_probe*;

record current time in *ending_time_of_the_function_excluding_probes*
get *running_time_excluding_probes* by subtracting *starting_time_without_probe* from *ending_time_excluding_probes*
get top node in *function_numbers_stack* which is function number
pop top node in *function_numbers_stack*
get top node in *function_numbers_stack* which is caller function number
pop top node in *function_numbers_stack*
record current time in *ending_time_of_the_function_including_probes*
get *running_time_including_probes* by subtracting *starting_time_including_probes* from *ending_time_of_including-_probes*
while (find the child-parent call relationship in the red-black tree of function information) **do**
    if the child-parent call relationship is found **then**
        accumulate *called_times*, *running_time_with_timer*, *running_time_excluding_probe* respectively
    else
        insert the running information of the current function into the red-black tree
    end if
end while

TABLE I
SETUP OF HYGON CLUSTER

Computing cluster	600 computing nodes
Storage cluster	Sugon parastro300 parallel storage system, including 4 opara and 36 osteor nodes
Network system	Intel OPA 100GB dedicated computing network

2) Test applications: Experiments are conducted on NAS Parallel Benchmark (NPB) and CAS-ESM (Chinese Academy of Sciences Earth System Model) respectively. The NPB is a set of parallel scientific computing applications used to evaluate the performance of supercomputers. It is an application to solve computational fluid dynamics problems, including 5 kernel applications and 3 pseudo applications. For example, SP is a pseudo application and it is used to solve the problem of the five-diagonal equation. The computation of NPB is predefined and divided into different categories. Some experiments are conducted on SP which is written by Fortran as the test. application [13].

In addition, the earth system models are also representative parallel scientific computing applications, which are mainly used to quantify the laws of the earth and the relationships between human activities and earth changes. Among them, CAS-ESM which is written by C and Fortran is an advanced and widely used earth system model released by the Chinese Academy of Sciences. CAS-ESM integrates independent model components, such as atmospheric, ocean, sea ice, land, dynamic global vegetation, atmospheric aerosol and chemistry model components. These model components are coupled with a coupler to form a complex earth system which has a huge amount of computation.

3) Baselines: Gprof is a profile tool installed in most compilers and widely used for performance analysis of C and Fortran applications. Additionally, TAU is a comprehensive profiling and tracing toolkit for performance analysis of parallel applications, which reaches the state of the art.

4) Metrics: The performances of gprof, TAU and LPerf are evaluated from precision, error rate and overhead. Precision is a critical metric for profiling because an unprecise profiler may miss the functions with short running time but large accumulated running time. Error rate indicates the gap between the profiling result and the built-in timing result. Overhead means additional running time caused by the profiler. Error rate and overhead are formulated as follows respectively.

$$T_{error} = |T_{profiling} - T_{original}| / T_{original} \tag{2}$$

$$T_{overhead} = |T_{withprofiler} - T_{original}| \tag{3}$$

where $T_{profiling}$, $T_{original}$, $T_{withprofiler}$ are defined as the profiling result, the built-in timing result, and the running time with profiler respectively. Especially, $T_{withprofiler}$ is different from $T_{profiling}$ which excludes the running time of profiler.

B. Validation for Benchmarking Applications

1) The Precision of Profiling NPB: To test the validation of LPerf on benchmarking applications, experiments are designed to use gprof, TAU and LPerf to obtain the running time of some functions in SP of NPB. As too many functions in SP to illustrate, the representative functions obtained by gprof, TAU, and LPerf are selected, as showed in table 2. Experimental results show that gprof cannot record the running time of ADI, set_constants_, and probe_read, TAU can record the running time of ADI, set_constants, and LPerf can record the running time of all functions. Overall, gprof could not record the

TABLE II
THE RUNNING INFORMATION OF FUNCTIONS IN NPB

Gprof	Function name	Called times	Running time(s)
	adi	1001	0.00
	set_constants_	1	0.00
	timer_read	1	0.00
TAU	Function name	Called times	Running time(s)
	adi	1001	225.82
	set_constants_	1	0.002
	timer_read	1	0.00
LPerf	Function name	Called times	Running time(s)
	adi	1001	110.28
	set_constants_	1	0.00156
	timer_read	1	$60 * 10^{-9}$

running time of the functions whose single running time is small; TAU could not record the running time of the functions whose single running time is small and called times is small; LPerf could record the running time of all the functions.

Imprecise profiling results obtained by gprof cause that running time of some called functions is recorded as 0. This leads to that the running time of functions whose running time is small but called many times cannot be recorded. However, the sum of the running time of these functions could be huge. If their running time cannot be recorded, it could cause significant errors in profiling results. LPerf can avoid this problem by using "rdtsc" which is a system timer with low overhead to obtain cycle-level running time of functions.

2) The Accuracy of Profiling NPB: The amount of computation of SP of NPB could be tuned by predefined sizes and iterations. The experiments are performed on SP with different sizes varying from 300 to 500 and different iterations varying from 3000 to 5000. 5 sets of example data of different sizes and iterations are obtained. As the error rate of gprof, 25%, is two orders of magnitude higher than TAU and LPerf. To show the results distinctively, we only list the results of TAU and LPerf in Figure 7 and Figure 8. The error rate of LPerf is lower than TAU at the size of 300, 400 and 500 and at the iteration of 3000, 3500 and 5000.

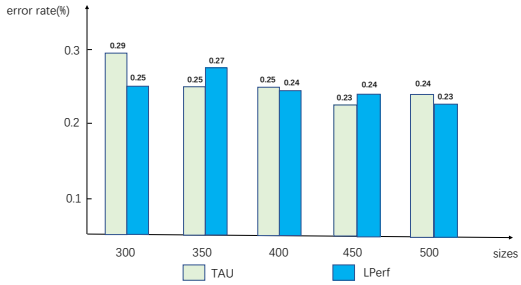

Fig. 7. Error rate of LPerf in different sizes

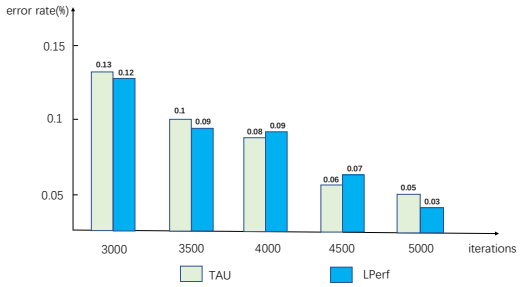

Fig. 8. Error rate of LPerf in different iterations

The experimental results show that the error rate of running time obtained by LPerf is comparable to TAU, and is much lower than gprof. In the best case, the error rates of gprof, TAU and LPerf are 20.4%, 0.05% and 0.03% respectively. Because of the profiling-aware method of calculating the running time of functions, high accuracy of LPerf is obtained.

3) The Overhead of Profiling NPB: The experiments are performed on SP with different sizes varying from 300 to 500 and different iterations varying from 3000 to 5000. 5 sets of example data of different sizes and iterations are obtained. Due to the overhead of gprof, 0.1%, is much lower than TAU and LPerf. To show the results distinctively, we only list the results of TAU and LPerf. Figure 9 and Figure 10 are the overhead of running time recorded by TAU and LPerf.

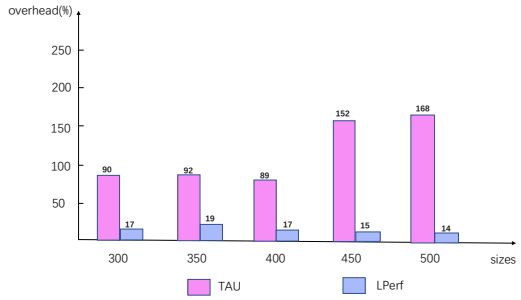

Fig. 9. Overhead of LPerf in different sizes

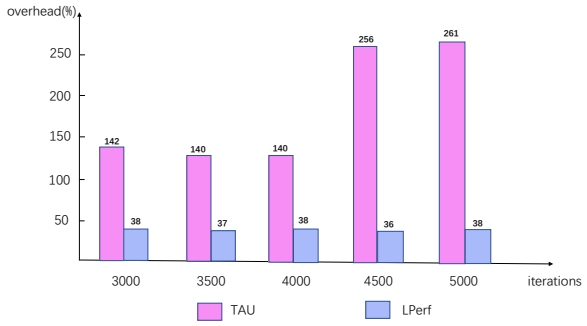

Fig. 10. Overhead of LPerf in different iterations

The experimental results show that the overhead of running time obtained by LPerf is significantly lower than TAU. In the

best case, the overhead of gprof, TAU and LPerf is 0.1%, 89% and 14%. Because of efficiently locating call relationships with low redundancy and automatically instrumenting with tunable granularity, low overhead of LPerf is obtained.

C. Validation for Large-scale Applications

1) The Profiling Results For CAS-ESM: To test the validation of LPerf for large-scale applications, experiments are designed to record the running time of functions in CAS-ESM through LPerf. For CAS-ESM, a three-level profiling scheme is proposed to distinguish and profile different model components according to the MPI communicators. That is, LPerf measures performance at the function level first, then profiles different components in an application according to the MPI communicators, and finally component performance is aggregated to obtain the overall performance. In the case of limited computing resources, it is essential to assign computing resources of different model components.

Due to gprof and TAU can not profile the model components of CAS-ESM, we only test the performance of LPerf in the following experiments. Figure 11 and Figure 12 show the error rate and overhead of profiling results for ATM model component which is the most time-consuming model component in CAS-ESM. These experiments are conducted at the size of 5,10,15,20 days for CAS-ESM. Experimental results demonstrate that LPerf can maintain high precision and accuracy with low overhead when it is applied to large-scale applications. In the best case, the error rates of ATM and CAS-ESM are 0.02% and 0.03% respectively, and the overhead of ATM and CAS-ESM are 1.6% and 2.3% respectively.

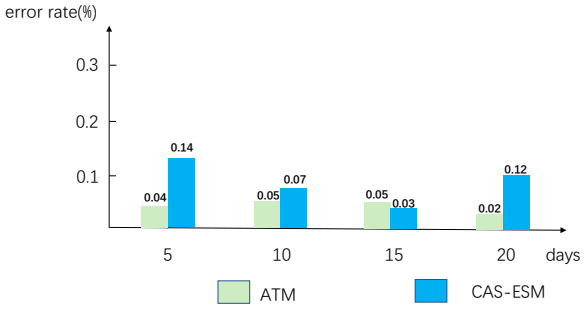

Fig. 11. Error rate of profiling CAS-ESM

2) The Effect of Tuning Granularity: To reduce the overhead of profiling, the method of instrumenting with tunable granularity is adopted. Ignoring to profile some functions with a few instructions can reduce the time overhead of LPerf. The functions whose number of instructions in IR files are lower than a user-defined threshold are not instrumented. Figure 13 shows the results of tuning the granularity. Experimental results show that the running time of CAS-ESM measured by LPerf can be effectively reduced by tuning the granularity.

D. Comprehensive Comparison

Table 3 summarizes the attribute of gprof, TAU and LPerf. From the Table 3, it is obvious that the precision and accuracy

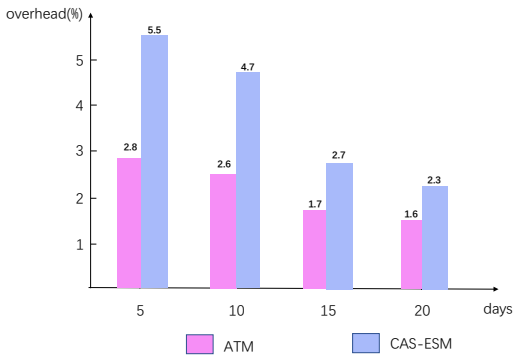

Fig. 12. Overhead of profiling CAS-ESM

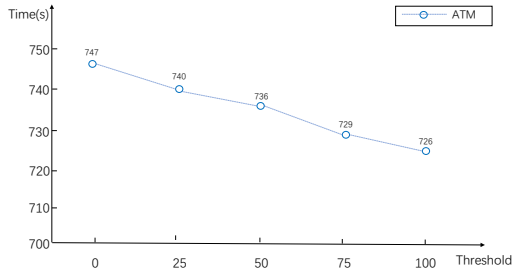

Fig. 13. Trend of running time after tunning the granularity

of LPerf can reach the state of arts with low overhead. By tuning the granularity of instrumentation, the running time of the applications measured by LPerf can be reduced effectively.

VI. Conclusion

In this paper, LPerf is proposed to profile HPC applications with low overhead and high accuracy. Major contributions are listed as follows. (1) To lower the time overhead and improve the applicability of LPerf, preprocessing and automatically instrumenting with tunable granularity during the compilation stage are addressed. (2) To reduce the time and memory overhead, an aggregated caller-callee relationship which is used to locate call relationships of functions efficiently is designed. (3) To obtain the exclusive time of functions precisely, a profiling-aware method of calculating the running time of functions is proposed. Especially, for the HPC applications with multiple MPI communicators, our methods also support modeling different components in an application according to the MPI communicators. It is essential to schedule the number of processes among the components to fully utilize the limited computation resources for HPC applications. The experimental results manifest that the precision, accuracy, and overhead of LPerf have reached the state of the art. In the future, the analysis of LPerf will be enhanced and performance modeling based on LPerf will be conducted.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (2020YFA0607900), Shang-

TABLE III
COMPARISON OF GPROF, TAU AND LPERF

	gprof	TAU	LPerf
Precision	⋆ (ms)	⋆ ⋆ ⋆ (cycle level)	⋆ ⋆ ⋆ (cycle level)
Error rate	⋆ (10% ˜40%)	⋆ ⋆ ⋆(0.05% ˜2%)	⋆ ⋆ ⋆ (0.02%˜2%)
Overhead	⋆ ⋆ ⋆(0.1% ˜0.2%)	⋆(89% ˜261%)	⋆ ⋆(1.6% ˜38%)
Tunable granularity	×	√	√

hai Key Laboratory of Multidimensional Information Processing, East China Normal University (No. MIP202102), and the Fundamental Research Funds for the Central Universities.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] Minwoo Ahn, Donghyun Kim, Taekeun Nam, and Jinkyu Jeong. Scoz: A system-wide causal profiler for multicore systems. *Software: Practice and Experience*, 51(5):1043–1058, 2021.

[3] Maurice Bailleu, Donald Dragoti, Pramod Bhatotia, and Christof Fetzer. Tee-perf: A profiler for trusted execution environments. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 414–421. IEEE, 2019.

[4] Nader Boushehrinejadmoradi, Adarsh Yoga, and Santosh Nagarakatte. A parallelism profiler with what-if analyses for openmp programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 198–211. IEEE, 2018.

[5] Lorenz Braun and Holger Fröning. Cuda flux: A lightweight instruction profiler for cuda applications. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 73–81. IEEE, 2019.

[6] Jon Calhoun, Luke Olson, and Marc Snir. Flipit: An llvm based fault injector for hpc. In *European Conference on Parallel Processing*, pages 547–558. Springer, 2014.

[7] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 727–743, 2018.

[8] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[9] Jay Fenlason and Richard Stallman. Gnu gprof. *GNU Binutils. Available online: http://www. gnu. org/software/binutils (accessed on 21 April 2018)*, 1988.

[10] Keke Gai, Yulu Wu, Liehuang Zhu, and Meikang Qiu. Privacy-preserving data synchronization using tensor-based fully homomorphic encryption. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1149–1156. IEEE, 2018.

[11] Keke Gai, Yulu Wu, Liehuang Zhu, Meikang Qiu, and Meng Shen. Privacy-preserving energy trading using consortium blockchain in smart grid. *IEEE Transactions on Industrial Informatics*, 15(6):3548–3558, 2019.

[12] Keke Gai, Yulu Wu, Liehuang Zhu, Lei Xu, and Yan Zhang. Permissioned blockchain and edge computing empowered privacy-preserving smart grid networks. *IEEE Internet of Things Journal*, 6(5):7992–8004, 2019.

[13] Keke Gai, Yulu Wu, Liehuang Zhu, Zijian Zhang, and Meikang Qiu. Differential privacy-based blockchain for industrial internet-of-things. *IEEE Transactions on Industrial Informatics*, 16(6):4156–4165, 2019.

[14] Peter Garba and Matteo Favaro. Saturn-software deobfuscation framework based on llvm. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 27–38, 2019.

[15] Susan L Graham, Peter B Kessler, and Marshall K McKusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 39(4):49–57, 2004.

[16] Yibo Guo, Qingfeng Zhuge, Jingtong Hu, Meikang Qiu, and Edwin H-M Sha. Optimal data allocation for scratch-pad memory on embedded multi-core systems. In *2011 International Conference on Parallel Processing*, pages 464–471. IEEE, 2011.

[17] Nikita Kataev. Application of the llvm compiler infrastructure to the program analysis in sapfor. In *Russian Supercomputing Days*, pages 487–499. Springer, 2018.

[18] Peter Lammich. Generating verified llvm from isabelle/hol. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[19] Zhen Li, Ali Jannesari, and Felix Wolf. An efficient data-dependence profiler for sequential and parallel programs. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 484–493. IEEE, 2015.

[20] Gangzhao Lu, Weizhe Zhang, Hui He, and Laurence T Yang. Performance modeling for mpi applications with low overhead fine-grained profiling. *Future Generation Computer Systems*, 90:317–326, 2019.

[21] Jonathan R Madsen, Muaaz G Awan, Hugo Brunie, Jack Deslippe, Rahul Gayatri, Leonid Oliker, Yunsong Wang, Charlene Yang, and Samuel Williams. Timemory: Modular performance analysis for hpc. In *International Conference on High Performance Computing*, pages 434–452. Springer, 2020.

[22] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[23] Andre Merzky, Ming Tai Ha, Matteo Turilli, and Shantenu Jha. Synapse: Synthetic application profiler and emulator. *Journal of computational science*, 27:329–344, 2018.

[24] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. Cudaadvisor: Llvm-based runtime profiling for modern gpus. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 214–227, 2018.

[25] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[26] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216. IEEE, 2010.

[27] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun. Online diagnosis of performance variation in hpc systems using machine learning. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):883–896, 2018.

[28] Hui Zhang and Jeffrey K Hollingsworth. Data centric performance measurement techniques for chapel programs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 377–386. IEEE, 2017.

[29] Lei Zhang, Meikang Qiu, Wei-Che Tseng, and Edwin H-M Sha. Variable partitioning and scheduling for mpsoc with virtually shared scratch pad memory. *Journal of Signal Processing Systems*, 58(2):247–265, 2010.

[30] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. Gvprof: a value profiler for gpu-based clusters. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1263–1278. IEEE Computer Society, 2020.