CCF Transactions on High Performance Computing (2020) 2:135–148
https://doi.org/10.1007/s42514-020-00035-8

REGULAR PAPER

APMT: an automatic hardware counter-based performance modeling tool for HPC applications

Nan Ding[1,2] · Victor W. Lee[3] · Wei Xue[1,4] · Weimin Zheng[1]

Received: 2 November 2019 / Accepted: 29 April 2020 / Published online: 24 June 2020

Abstract
The ever-growing complexity of HPC applications and the computer architectures cost more efforts than ever to learn application behaviors. In this paper, we propose the *APMT*, an Automatic Performance Modeling Tool, to understand and predict performance efficiently in the regimes of interest to developers and performance analysts while outperforming many traditional techniques. In APMT, we use hardware counter-assisted profiling to identify the key kernels and non-scalable kernels and build each kernel model according to our performance modeling framework. Meantime, we also provide an optional refinement modeling framework to further understand the key performance metric, cycles-per-instruction (CPI). Our evaluations show that by only performing a few small-scale profiling, APMT is able to keep the average error rate around 15% with average performance overheads of 3% in different scenarios, including NAS parallel benchmarks, dynamical core of atmosphere model of the Community Earth System Model (CESM), and the ice component of CESM on commodity clusters. APMT improve the model prediction accuracies by 25–52% in strong scaling tests comparing to the well-known analytical model and the empirical model.

Keywords Performance Modeling · Automatic Modeling · Hardware Counter · Kernel Clustering · Parallel Applications

1 Introduction

The ever-growing complexity of HPC applications, as well as the computer architectures, cost more efforts than ever to learn application behaviors by massive analysis of applications' algorithms and implementations. To make projections of applications' scaling run-time performance, designing performance models (Marathe et al. 2017; Balaprakash et al. 2016; Xingfu et al. 2014; Jones et al. 2005; Craig et al. 2015; Bhattacharyya et al. 2014; Bauer et al. 2012; Nan et al. 2014; Pallipuram et al. 2015, 2014) has long been an art only mastered by a small number of experts. Nevertheless, we can still see that performance models can be used to quantify meaningful performance characteristics across applications (Balaprakash et al. 2016; Xingfu et al. 2014) and to provide performance bottlenecks associated with their implementations (Williams et al. 2009); to offer a convenient mechanism for users and developers to learn the scaling performances (Calotoiu et al. 2013); and even to guide the optimization decisions (Nan et al. 2014).

Recently, several methods for performance modeling have been developed to simplify and streamline the process of modeling. Techniques range from traditional expert (analytical) modeling (Bauer et al. 2012; Nan et al. 2014), through compiler-assisted modeling Bhattacharyya et al. (2014) and domain language-based modeling Spafford and Vetter (2012) to fully automatic (analytical and empirical) modeling (Jones et al. 2005; Craig et al. 2015; Knüpfer et al. 2012; Hong and Kim 2009; Barnes et al. 2008; Balaprakash et al. 2016; Xingfu et al. 2014). However, these modeling techniques are either inadequate to capture the functional relationships between applications performance and the target architecture or suffer from manual high-efforts to learn the algorithm and implementation case by case.

✉ Wei Xue
xuewei@mail.tsinghua.edu.cn

1 Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

2 Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

3 Intel Corporation, Santa Clara, CA 95054, USA

4 National Supercomputing Center in Wuxi, Wuxi 214000, China

The main advantage of analytical models is interpretability which allows users to reason about performance while the disadvantage is the high manual efforts of portability across applications. Empirical models are built according to statistical methods which is good at portability across applications but it is challenging to achieve high model accuracy while keeping low overheads. So far, tools only supported either analytical (Craig et al. 2015; Bauer et al. 2012) with several prior knowledge or empirical modeling (Jones et al. 2005; Craig et al. 2015; Knüpfer et al. 2012) with regression-based method. Machine learning techniques have been introduced to improve the model accuracy when predicting the execution time from observed performance results of a large number of application runs (Balaprakash et al. 2016; Xingfu et al. 2014). It usually takes a long time to profile and train the model, especially for the cost-expensive applications.

Different from the previous work, we propose APMT, an offline performance modeling tool with hardware counter-assisted profiling, to overcome the disadvantages of high manual efforts (analytical models) and unwarrantable model accuracy (empirical models). Hardware counter-assisted profiling is widely used in state-of-the-art performance tools, such as HPCtoolkit (Adhianto et al. 2010), LIKWID (Treibig et al. 2010), Intel Vtune (Malladi 2009), PAPI (Mucci 1999), and perf (Weaver 2013), to learn application behaviors rather than apply massive analysis to the application. Meanwhile, hardware counters are supported on various processors, such as Intel processors (sps22 sps22), IBM processors (Liang 2009) and AMD processors (Zaparanuks et al. 2009).

APMT starts from a simple analytical model framework, predicts the computation and communication performance separately. We use hardware counter-assisted technique to predict applications' computation performance using our pre-defined functions. We instrument the PMPI interface Keller et al. (2003) to profile communication performance, and then use the well-known Hockney model Chou et al. (2007) to predict the communication performance.

To summary, the key contributions of this paper are:

1. A hardware counter-assisted technique to identify expensive and non-scalable kernels Three types of kernels are detected in APMT: key kernels (large run time proportion), non-scalable kernels, and the sum of rest functions. Such a method allows us to reduce the number of kernels by more than an order of magnitude compared to the loop-level kernels (Bhattacharyya et al. 2015, 2014).

2. A low-overhead performance modeling framework A common problem in modeling large-scale applications is the modeling overhead, which often exceeds 10% (Malony et al. 2004). Yet, machine time, especially at scale, is very valuable. Moreover, large overheads may cause applications' performance deviations as well. Our hardware counter-assisted method can keep the overhead to 3% on average.

3. A novel scheme to understand performance through model parameters 'Cycles per instruction' (CPI), blocking factor for computation and blocking factor for communication are proposed to understand the kernel's instructions throughput, memory traffic, and computation/communication overlap.

4. Evaluations on real applications We deploy the APMT on two real-world applications and one proxy application: HOMME (Dennis et al. 2012), the dynamical core of the Community Atmosphere System Model (CAM Dennis et al. 2012), and the Los Alamos CICE model (Hunke et al. 2010), a full simulation of sea ice which is used in coupled global climate models, and the NAS parallel benchmarks (Bailey et al. 1991). APMT improves the model accuracy up to 52% compared to previous models (Nan et al. 2014; Worley et al. 2011).

APMT characterizes the performance of applications more concisely than previous approaches. With these efforts, we intend to develop an automatic and practical use performance modeling tool with the capabilities of interpretability, low-overhead, and portability. Thus, we demonstrate that APMT can bridge the gap between high manual effort performance modeling and automated modeling approaches.

2 Performance profiling

To guarantee lightweight analysis, APMT leverages a hardware counter-assisted method for the computation profiling, and instrument the PMPI interface Keller et al. (2003) to profile the MPI performance.

Hardware Counter-assisted Profiling. Although the instrumentation is a usual way to obtain the functions' timing information (Bhattacharyya et al. 2015, 2014), the hardware counter-assisted profiling can also get the timing information provided by the hardware counter (CPU cycles), but in a light-weight way (Weaver 2013; Bitzes and Nowak 2014). The hardware counter-assisted profiling has 3% overhead on average when profiling different hardware counters whilst the overhead of instrument-based profiling tools, such as Score-P (Knüpfer et al. 2012), and Scalasca (Geimer et al. 2010), usually exceeds 10%.

Moreover, hardware counters can also provide more architecture-oriented information of application runs, such as CPI (cycles per instruction), the number of L1/L2/LLC data cache accesses/misses and the waiting time for memory, to help further identifying and understanding the kernel performance characteristics (Gamblin et al. 2011; Zaparanuks et al. 2009; Browne et al. 2000).

To the best of our knowledge, most existing performance models that use hardware counter-assisted profiling focus on coarse-grained performance insights, such as flops and memory bandwidth (Williams et al. 2009). Those models can be very useful to tell users the kernel is compute bound or memory bound. Thus, kernels with a low algorithmic operations and low flop rate should be the ones need to be optimized. However, a very important golden standard of performance metrics, timing, is missing which can mislead future optimization decisions.

APMT differs from the established approaches, we use hardware counter-assisted profiling to provide a finer-grained performance insights such as memory and compute overlap, and scaling time-to-solution performance estimation. Such kinds of the insights can guide developer with future optimization with kernels should be optimized in next steps and optimazation directions such as communication, memory and compute overlap, and data reuse.

One important information from performance profiling is to know where the time goes. Here, we use `CPU_CLK_UNHALTED.THREAD_P` which refers to the CPU cycles to profile the timing of each function. The accuracy of the hardware counter-assisted method (sampled)for finding hotspots has been validated (Wu and Mencer 2009; Merten et al. 1999). In this paper, we use the normalized root mean square error (NRMSE, Eq. 1) to evaluate the time percentage differences between the hardware counter-assisted method (sampled)and real values (instrumented). In Eq. 1, K is the total number of kernels, t_i and r_i refer to the time proportions of sampled time proportions and the instrumented time proportions. The instrumented values are measured by using the well-known call graph execution profiler (GPROF Garcia et al. 2011). We use two-million-cycles intervals as sampling intervals, and NRMSE is around 10^{-4}, which indicts good matches between sampled results and the real ones.

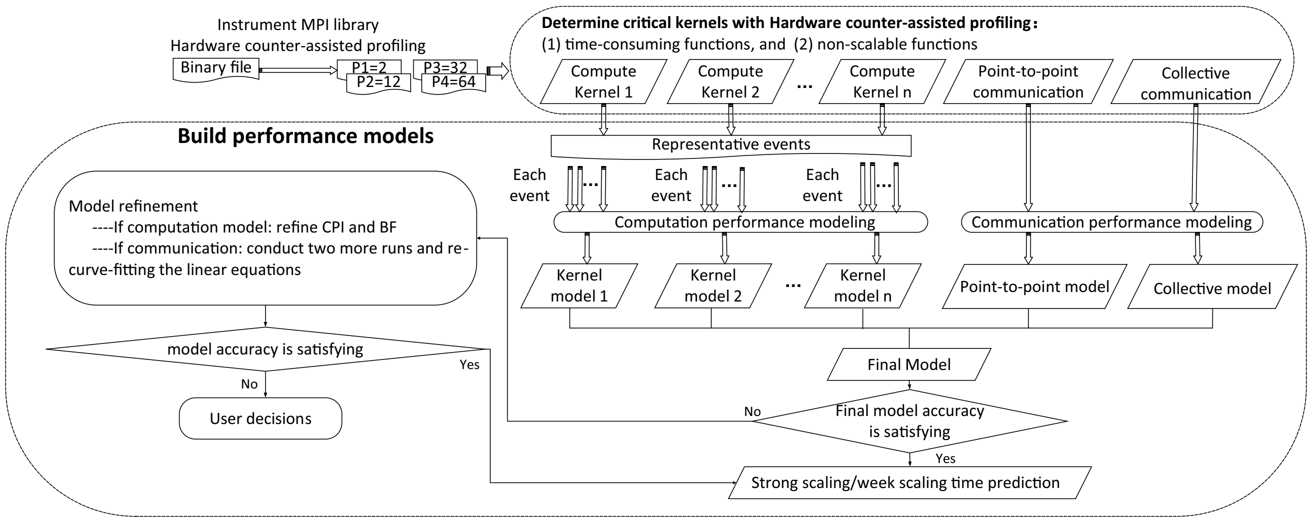
$$NRMSE = \frac{\sqrt{\sum_{i=1}^{K} t_i \cdot (t_i - r_i)^2}}{\max\limits_{1<i<K}(t_i \cup r_i) - \min\limits_{1<i<K}(t_i \cup r_i)} \quad (1)$$

Instrumented MPI Profiling. We generate communication traces, namely, the message size, the message count, the source, and destination, by using the standardized PMPI interface (Keller et al. 2003; Gamblin et al. 2011). Performance measurements may have serious run-to-run variation because of OS jitter, network contention, and other factors. To ensure the profiling validity, users can repeat measurements until the variance stables.

3 Performance model framework

In this section, we present the overview of APMT, and then we illustrate how to build computation and communication models for each detected kernels.

Figure 1 shows the overview of the APMT. We first breakdown the applications into a set of compute kernels by conducting several profiling runs. We then model the performance of each compute kernel with the representative hardware counters according to our model framework. For communications, we construct the model in terms of point-to-point communication and collective communication. We also conduct a closed-loop model refinement to meet the user-defined model accuracy. In the end, the model reports strong/weak scaling run-time performance. User decisions have to be made when the results after model refinement can not meet the accuracy requirements.



Instrument MPI library
Hardware counter-assisted profiling
Binary file
P1=2
P2=12
P3=32
P4=64
Determine critical kernels with Hardware counter-assisted profiling:
(1) time-consuming functions, and (2) non-scalable functions
Compute Kernel 1
Compute Kernel 2
...
Compute Kernel n
Point-to-point communication
Collective communication
Build performance models
Representative events
Each event
Each event
Each event
Computation performance modeling
Communication performance modeling
Model refinement
----If computation model: refine CPI and BF
----If communication: conduct two more runs and re-curve-fitting the linear equations
Kernel model 1
Kernel model 2
...
Kernel model n
Point-to-point model
Collective model
Final Model
model accuracy is satisfying
Yes
No
User decisions
No
Final model accuracy is satisfying
Yes
Strong scaling/week scaling time prediction

Fig. 1 Overview of APMT. Three key steps to build performance models: (1) profiling, (2) determine critical kernels, and (3) build computation and communication performance model for each critical kernels

3.1 Model inputs and outputs

We take the number of processes and the problem sizes as our model inputs because they are the most commonly-used settings for performance evaluation. Model outputs are the application run-time of the given problem size and the given parallelism, as well as the timing breakdown of the model. The problem size cannot be determined only by grid size in structured grid problem because the iteration number may not remain the same due to convergence. Therefore, the problem size has to take both grid size (nx, ny, nz) and the number of iterations ($iter$) into account. Those inputs can be obtained by either from domain experts or the application configurations.

3.2 Critical Kernel identification

We identify three kinds of kernel candidates by using execution cycle counter (`CPU_CLK_UNHALTED.THREAD_P`) from at least three profiling runs using different parallelisms. Figure 2 shows an example of how to identify critical kernels. We choose functions as a set of kernel candidates because it is natural to separate communication and computation in parallel applications to enable us to predict them separately. We can use the same idea to handle kernel identification for different input size.

The three kinds of critical kernel candidates are listed below.

1. Functions that whose share of the application execution cycles are larger than a user-defined threshold (default is 5% in this paper) .
2. Functions that whose run-time is not decreasing. The non-expensive functions may become expensive ones when we run applications using different parallelisms or different inputs. For example, the functions in the sequential part can turn into hotspots when running the application with more processes. As a result, we need to find these potential hotspot functions for different numbers of processes and inputs.
3. The remaining functions after 1 and 2. The reason is that the aggregated kernel can reduce the overhead of building performance models while maintaining good accuracy. Moreover, the entire run-time would be slightly affected even if we consider those small functions individually.

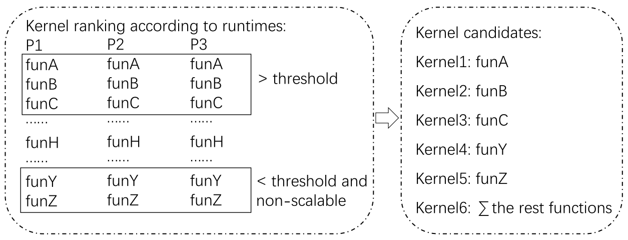


Fig. 2 An example of critical kernel candidate identification. $P1$, $P2$ and $P3$ refer to different number of processes. *funA–funC* represent the kernels which the time percentages are larger than the user-defined threshold (default is 5% in this paper). *funY-funZ* represent the kernels which the time percentages are not decreasing with growing number of processes. The rest of the functions are aggregated as one big kernel

3.3 Model construction

The performance model framework is shown as Eq. 2. The computation and communication times are taken into account separately in APMT because they have different performance characteristics.

$$T\_app = \sum_{i=1}^{n}(T\_comp_i + BF\_mem_i * T\_mem_i) + BF\_comm * T\_comm + T\_others \tag{2}$$

The computation time of a kernel can be estimated by two components. The first part is the time it takes to execute the computation instructions (T_comp_i, i is kernel index, n is the total number of kernels). The second part is the time associated with fetching the data from storage to compute Clapp et al. (2015) (T_mem_i). There is typically some degree of overlap between these two components. We introduce a variable named memory blocking factor (BF_mem_i), which measures the non-overlapping part for loading data from local memory.

The communications (T_comm) can be categorized into two groups: point-to-point (p2p) communication and collective communication. They have different performance according to the number of processes and the communication volume. The summation of the p2p and the collective communication times is used to predict the total communication time.

Different from the previous works that fit the kernel model directly (Bhattacharyya et al. 2014), we introduce hardware counter-assisted performance models for each kernel. With architecture-level modeling, our framework is slightly more complicated than previous ones but offers more insights on performance.

Computation model construction. Table 1 lists all model parameters in APMT, and Table 2 summarizes how they are derived. Instead of using high order equations to cover the non-linearity of performance models (Van den et al. 2015; Jayakumar et al. 2015), we pre-define a set of linear fitting functions including polynomial, exponential and logarithm for each hardware counter. We do not apply high degree equations in the first place for two reasons. The first one

Table 1 Performance models descriptions

	Descriptions
T_comp_i	Calculation time of each kernel
T_mem_i	Total memory time
BF_mem_i	Ratio of non-overlapped memory time
BF_comm	Ratio of non-overlapped communication time
T_comm	Average communication time
T_stall_i	Waiting time for memory
T_L1_i	L1 cache access times
T_L2_i	L2 cache access times
T_LLC_i	Last level cache access times
T_mm_i	Main memory times
$instructions$	Executed instructions
$T\_collective$	Collective MPI communication
T_p2p	Point to point MPI communication
T_others	Initialization and finalization time
CPI_i	Cycles per instruction (measured)
s_{tot}	Total communication volume (measured)
T_mapp	Total application time (measured)
T_mcomp	Total computation time (measured)
T_mcomm	Total communication time (measured)
r	Total number of p2p MPI operations (measured)
l	Total number of collective MPI operations (measured)
P	Number of processes (input)
D	Problem size (input)
n	Total number of kernels (detected)

is that high degree equations often suffer from over-fitting. The second reason is that more model parameters will result in additional performance modeling overhead. Each extra model parameter requires at least one more application run in order to solve the multivariate equations.

We use the highest rsquare Calotoiu et al. (2013) one among the pre-defined fitting functions for each hardware counter. That is to say, each hardware counter is a function of parallelism, and then we assemble the corresponding hardware counters according to Table 2 to describe the non-linear run-time performance of each kernel. Figure 3 shows an example of how to derive model parameters using a kernel from HOMME. Model parameter BF_mem_i consists of five model variables T_stall_i, T_L1_i, T_L2_i, T_LLC_i and T_mm_i.

$$BF\_mem_i = \frac{T\_stall_i}{T\_L1_i + T\_L2_i + T\_LLC_i + T\_mm_i} \tag{3}$$

where T_L1_i represents to the number of L1 hits in kernel i, and T_L1_i can be described as a function of P using hardware counter $MEM\_LOAD\_UOPS\_RETIRED.L1\_HIT\_PS$. We then multiplied the function by the L1 cache latency. The latency is measured using Intel's Memory Latency Checker (Doweck 2006). The remaining model variables are derived similarly. Here, we only consider the worst case of T_mem_i in the model and do not account for concurrency in the memory subsystem.

One may notice that the non-overlap memory time can be directly calculated by T_stall_i. However, knowing the scaling estimated runtime is not sufficient to understand what constraints the performance. Therefore, we keep T_L1, T_L2, T_LLC in the model to provide more performance insights such as the compute and memory overlap ratio (BF_mem) and the cache performance. Those can be good guidance for developers to decide what optimization should be performed in next steps (c.f. Sect. 4.1).

For each kernel's computation time, we use number of instruction multiplied by cycles per instruction (CPI). One

Table 2 How is the model item derived from hardware counter?

	How is the model item derived?
T_comp_i	$\frac{instructions_i * CPI_i}{CPUfrequecy * P}$
T_mem_i	$T\_L1_i + T\_L2_i + T\_LLC_i + T\_mm_i$
BF_mem_i	$\frac{T\_stall_i}{T\_mem_i}$
BF_comm	$\frac{T\_mapp - T\_mcomp}{T\_mcomm}$
T_comm	$\sum_{i=1}^{r} T\_p2p + \sum_{i=1}^{l} T\_collective$
T_p2p	$a \cdot s_{tot} + b$
$T\_collective$	$a \cdot log(P) + b \cdot s_{tot} + c$
T_stall_i	Fitting from RESOURCE_STALLS.LB(ST) counter
T_L1_i	Fitting from MEM_LOAD_UOPS_RETIRED.L1_HIT_PS counter
T_L2_i	Fitting from MEM_LOAD_UOPS_RETIRED.L2_HIT_PS counter
T_LLC_i	Fitting from MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS counter
T_mm_i	Fitting from MEM_UOPS_RETIRED.ALL_LD(ST)_PS counter
$instructions_i$	Fitting from INST_RETIRED.ANY_P counter
T_others	Fitting from P

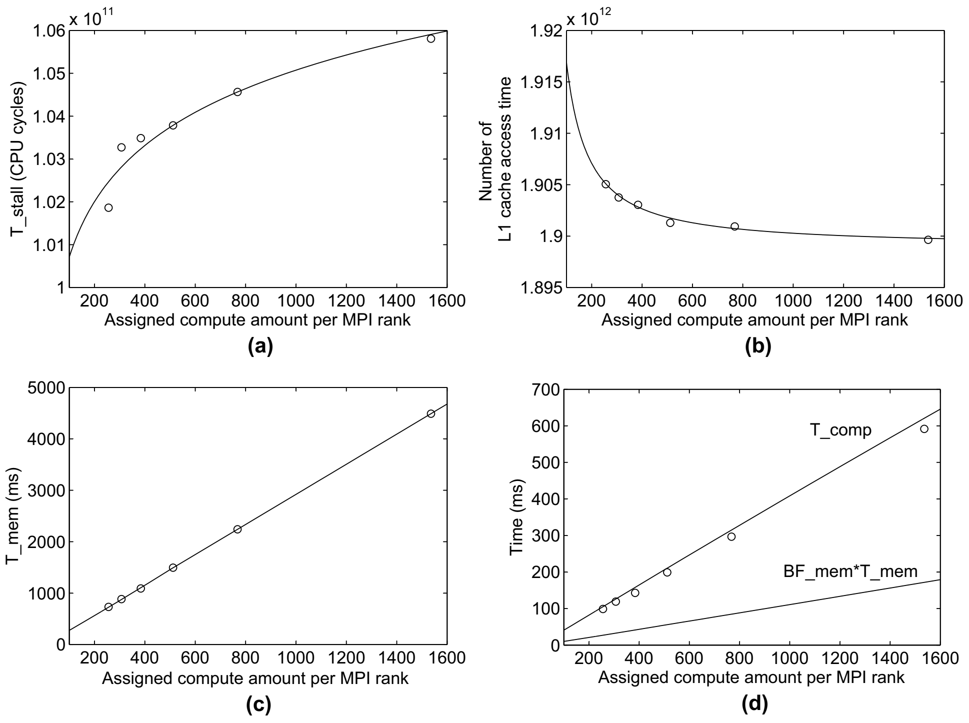


Fig. 3 An example of how to derive model parameters. The example is a kernel (`laplace_sphere_wk`) from HOMME with $32 \cdot 32 \cdot 6$ grids. We use the average compute amount assigned to each MPI rank ($\frac{32 \cdot 32 \cdot 6}{P}$) as the x-axis, therefore as the values of x-axis become bigger, the smaller number of MPI ranks we use. The profiling process number $P = \{4\ 8\ 12\ 16\ 20\ 24\}$

may note that CPI of one kernel may change rapidly because the cache misses/stalled memory instructions may lower the instruction throughput. Therefore, the CPI in this paper refers to the computation time and computation cycles only. The *CPI_i* is the average CPI from the three profiling runs using different number of processes of one kernel. We use a standard deviation (Eq. (4)) to evaluate the data volatility.

$$deviation = \frac{\sum_{p=1}^{n}(CPI\_i_p - \bar{CPI}\_i_p)^2}{n-1} \quad (4)$$

Communication model construction. For the point-to-point (p2p) communication, we assume a linear relationship because all processes can carry on their operations in parallel. The time cost t of sending a certain number of message n of size s equals to $t = n \cdot (a \cdot s + b)$ according to the well-known Hockney model (Chou et al. 2007). We modeling the p2p communication time t with total communication size ($s\_tot = n \cdot s$) as $a \cdot s_{tot} + b$ (m=1, k=0 of our pre-defined fitting function in Table 2).

For collective communications, we consider the *MPI_Bcast*, *MPI_Alltoall*, and *MPI_Allreduce* in the subset of MPI collective operations. Take *MPI_Bcast* as an example, the time cost t of a broadcast a message of size s among all processes P equals to $t = a \cdot log(P) + b \cdot s + c$. For a sake for simplicity, we do not model each message sizes, and we use an average message size among processes.

3.4 Model refinement

The model results sometimes can be far away from the realities. Such error either stems from the communication contention or the computation part. Therefore, if the model error comes from communication, we refine the total communication volume by conducting two more profiling runs using different process numbers. One profiling run is used to re-fitting the functions, and the other one is used for validation. The overhead of refining communication is two more application profiling runs.

If the model error comes from the computation part, we believe it results from the model parameters CPI_i and BF_mem_i. Let's think the model parameter CPI represents the effective cycles per instruction. We then define CPI_core that stands for the CPI if all memory references are served by cache (Clapp et al. 2015), as Eq. (5) shows. If we take the cache as an infinite cache, CPI equals to CPI_core. If we add cache misses, the memory blocking factor will increase to reflect the impact of memory latency.

However, refining CPI_i and BF_mem_i can not be easily resolved due to the complexity of the shared memory systems and applications. Here we provide a methodology to refine CPI_i and BF_mem_i which requires changing the CPU frequency and repeating the test using one parallelism in the previous profiling runs. Thus, we can make the memory faster compared to the speed of executing instructions by lowering the CPU frequency. After that, we can get two sets of CPI_i and T_mem_i. We then can calculate the CPI_core_i and BF_mem_i by solving the linear equation (Eq.(5)). As shown in Fig. 10, the y-intercept of the line is CPI_core_i and the gradient is BF_mem_i. Therefore, the overhead of refining computation parameters are higher than refining communication. In addition, the computation refinement may only work for the machines who can reboot parts of the compute nodes without interrupting other ongoing jobs.

$$CPI_i = CPI\_core_i + BF\_mem_i * T\_mem_i \tag{5}$$

Finally, we get a hierarchical and fine-grained performance modeling framework that combines the advantages of analytical and empirical methods. APMT combines easily interpretable linear and logarithmic functions into robust and accurate non-linear application performance models.

4 Evaluation

In this section, we first describe the experiment platforms and configurations. We then apply APMT to two real-world applications and one proxy application, and show how to use APMT to understand run-time performance.

4.1 Experiment platforms and configurations

The experiments are carried out on two platforms, a 4-node Intel Xeon cluster and an Intel cluster in National Supercomputing Center in Wuxi of China (NSCC-Wuxi). Each node of the 4-node Intel Xeon cluster contains two Intel Xeon E5-2698v3 processors running at 3.0 GHz with 64 GB of DDR3-1600 memory. The operating system is CentOS 6.7. Each NSCC-Wuxi node contains two Intel Xeon E5-2680v3 processors running at 2.5 GHz with 128 GB memory. The operating system is RedHat 6.6. The MPI version of two clusters is Intel MPI 15.0, and the network of both clusters is FDR InfiniBand.

We use APMT to predict the run-time performance of two real-world applications: HOMME Dennis et al. (2012) and CICE Hunke et al. (2010), and one proxy application: NPB Bailey et al. (1991). HOMME Dennis et al. (2012) is the dynamical core of the Community Atmospheric Model (CAM) being developed by the National Center for Atmosphere Research (NCAR). The Los Alamos sea ice model (CICE) Hunke et al. (2010) is a widely used sea ice model in the famous CESM project (Craig et al. 2015). The NPB is a well-known suite of benchmark that proxy scientific applications Asanovic et al. (2006) by mimicing the computation and communication characteristics of large scale computational fluid dynamics (CFD) applications. The applications all run on a single node to better understand the memory contention. We perform the predictions across nodes to prove the effectiveness and robustness of APMT. Furthermore, we evaluate CICE (up to 1024 processes) and HOMME (up to 3000 processes) on NSCC-Wuxi for large scale runs. I/O is not considered in our evaluation.

4.2 Results

In this section, we show how APMT breaks down the applications into kernels, provided performance insights, and conduct the model refinement. In the CICE case, we show how to use APMT to predict the strong-scaling run-time and how APMT can help the users and developers to learn the performance characteristics. In HOMME, we present how we estimate the time-to-solution performance for large problem size, and how non-scalable kernels behave when conduct strong scaling application runs. In NPB, we focus on how to apply the model refinement technique to the kernel.

Kernel identification. Table 3 lists our experiment configurations, the number of kernels detected by our hardware counter-assisted profiling, the number of loop-level kernels Bhattacharyya et al. (2014) of the small problem size. Compared to the number of loop-level kernels ("Lker" in Table 3), we reduce the number of kernels by more than an order of magnitude.

CICE We detect seven kernels on the 4-node cluster while five kernels on NSCC-Wuxi. Due to the different run-time performance on the two computing platforms, `shortwave_compute_dEdd` and `transport_remap_gradient` on the 4-node cluster are not taken as kernels on NSCC-Wuxi. The function `shortwave_compute_dEdd` computes the transports across each edge by integrating the mass and tracers over each departure triangle, which only costs 0.52% time proportion on NSCC-Wuxi while it takes 9.57% on the 4-node cluster. The function `transport_remap_gradient` computes a limited gradient of the scalar field phi in scaled coordinates which costs 1.87% time on NSCC-Wuxi while it takes 5.00% on the 4-node cluster. The sum of all the kernel's run-time can cover 99% computation time of CICE on both of the two computing platforms.

The profiling process number of CICE are $P = \{4\ 8\ 10\ 16\ 20\ 24\}$ on NSCC-Wuxi and $P = \{2\ 8\ 16\ 20\ 24\}$ on the 4-node cluster. As Fig. 4 shows, each CICE kernel's CPI deviation (Eq. (4)) using different number of processes is around 10^{-3} - 10^{-2} on the 4-node cluster and NSCC-Wuxi. Therefore, we use a constant average CPI as default option during the performance modeling for low overhead.

We observe that `Limited_gradient` and `transport_integrals` have similar CPIs on the two computing platforms in Fig. 4, respectively. However, by looking into its memory behavior (Fig. 5), we can see

Table 3 Number of functions (Func), total number of kernels (ker), number of non-scalable kernels (nonk), LOCs, number of kernels from loop-level (Lker) modeling work (Bhattacharyya et al. 2014)

	Func	ker	nonk	LOCs	Lker	Size
CICE	109	7	2	75,000		$116 \cdot 100$
						$384 \cdot 320$
HOMME	210	11	6	113,095		$32 \cdot 32 \cdot 6 \cdot 128$
						$256 \cdot 256 \cdot 6 \cdot 128$
EP	5	3	0	359	12	2^{28}
						2^{30}
MG	16	5	1	2,568	98	$256 \cdot 256 \cdot 256 \cdot 4$
						$512 \cdot 512 \cdot 512 \cdot 20$
FT	10	5	2	2,034	39	$256 \cdot 256 \cdot 128 \cdot 6$
						$512 \cdot 256 \cdot 256 \cdot 20$
SP	15	5	1	4,902	229	$64 \cdot 64 \cdot 64 \cdot 400$
						$102 \cdot 102 \cdot 102 \cdot 400$
LU	11	6	2	5,957	165	$64 \cdot 64 \cdot 64 \cdot 250$
						$102 \cdot 102 \cdot 102 \cdot 250$
BT	14	8	2	9,162	211	$64 \cdot 64 \cdot 64 \cdot 200$
						$102 \cdot 102 \cdot 102 \cdot 200$
CG	3	6	1	1,901	30	$14,000 \cdot 15$
						$75,000 \cdot 75$

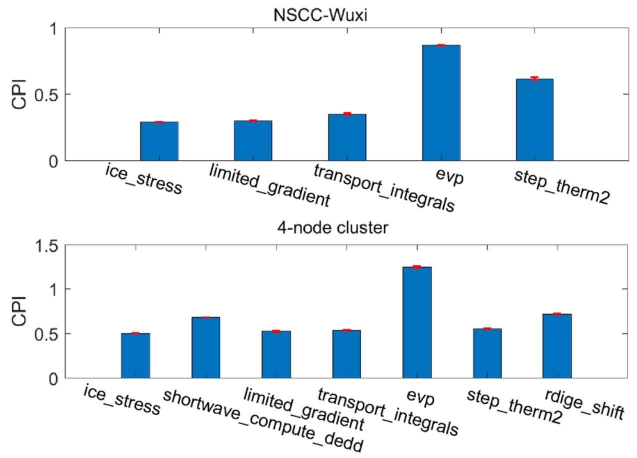


Fig. 4 CPIs and their deviations of each kernel with CICE runs using different number of processes on the 4-node cluster and NSCC-Wuxi

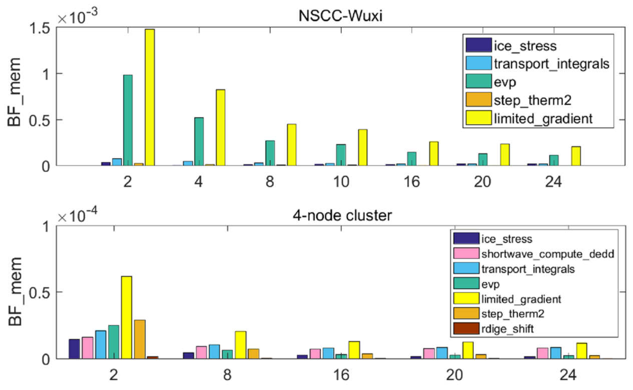


Fig. 5 Memory blocking factor of each CICE kernel on NSCC-Wuxi using problem size gx3. The horizontal resolution of gx3 is $116 \cdot 100$

that `limited_gradient` and `transport_integrals` have different memory performance characteristics (*BF_mem*). `limited_gradient` has a higher *BF_mem* than `transport_integrals` which indicts that `limited_gradient` suffers from a lower memory traffic.

Recall that *BF_mem* is estimated by using $\frac{T\_stall}{T\_L1+T\_L2+T\_LLC+T\_mainmemory}$ with seven hardware counters (Table 2). Take the the *BF_mem* of `limited_gradient` and `transport_integrals` on NSCC-Wuxi as an example, we can see that most of the memory access happen in L1 cache for both kernels (orange line is close to the green line) in Fig. 6. However, the stall cycles due to store operation (*RESOURCE_STALLS.ST*) of `limited_gradient` is much higher than the kernel `transport_integrals` which denotes the differences of *BF_mem*. The large store stall cycles indict that there exists a relatively strong data dependency in `limited_gradient` than `transport_integrals`. Based on the above insights from the model, reducing data dependencies of `limited_gradient` should be performed in next optimization step.

For communication, we focus on inter-node communication. Therefore, we profile inter-node communication on NSCC-Wuxi using a 32-process run, a 96-process run and a 128-process run as shown in Fig. 7. We see that there exists a lot of p2p communications. These communications are used to update halo regions (ghost cells) using `MPI_Send` and `MPI_Recv`.

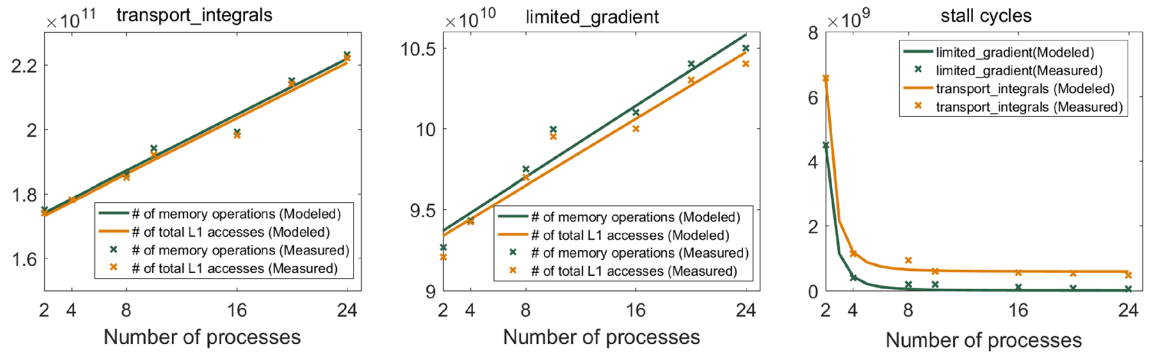


Fig. 6 Hardware counter-assisted profiling of two CICE kernels on NSCC-Wuxi using problem size gx3. The horizontal resolution of gx3 is $116 \cdot 100$. The *R_square* Hu et al. (1999) of our hardware counter-assisted profiling is around 0.9. This indicates a good quality of fitting between the model and the measured performance data. The range of *R_square* is [0, 1], and the higher, the better

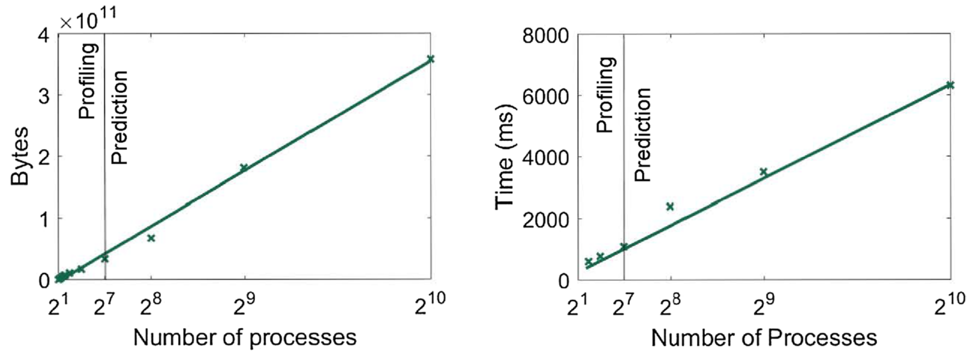


Fig. 7 Communication time of CICE with problem size (gx3) of $116 \cdot 100$ on NSCC-Wuxi. The dotted points are the measured data, and the (green) lines are plotted with our model. The *y*-axis are the accumulated total communication volume (Bytes) and accumulated total communication time of all MPI ranks

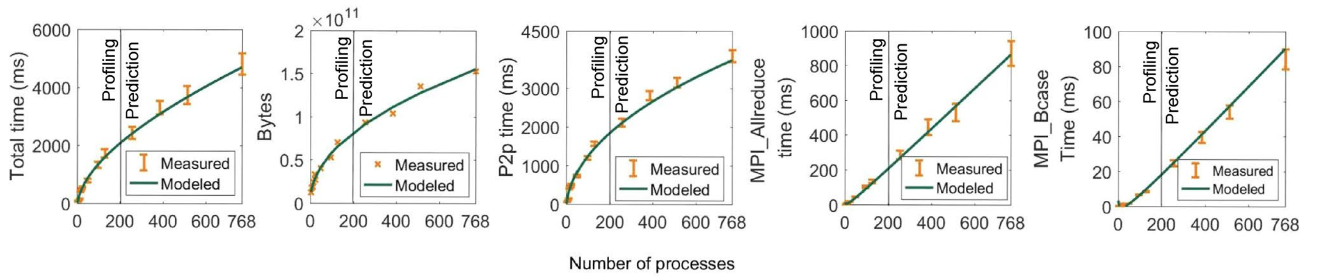


Fig. 8 Measured vs. Predicted communication time of the *ne*32 problem size in HOMME on the Intel cluster of NSCC-Wuxi

HOMME. We profile the small problem size *ne*32 (grid number: $32 \cdot 32 \cdot 6$, vertical level: 128). Fig. 3 shows the computation performance, and Fig. 8 shows the communication performance. For a strong scaling evaluation, p2p communication contributes the most in its total communication time. With the help of our model, we can see that the total p2p communication volume increases with a growing number of processes. On the contrary, the total communication volume remains the same (24KB) for `MPI_Allreduce` and `MPI_Bcast`, respectively.

The run-time performance prediction for a larger problem size *D_large* is according to the average compute amount per MPI rank. This is because most of the current HPC applications follows the well-known BSP model (Stewart 2011). Of course, the memory resource contention plays an important role in run-time performance, especially the main memory access time. In this paper, we define a threshold $E = \| \frac{N\_LLCmiss\_i_p}{N\_totalmem\_i_p} \|$ to evaluate the effectiveness for D_large run-time prediction from the harm of memory contention. $N\_LLCmiss\_i_p$ represents the number of last level cache misses of kernel i with a number of processes p, and $N\_totalmem\_i_p$ is the total number of memory access of p. Our experiments show that last level cache does not play an important role if $E \leq 1e-4$. Otherwise if $E > 1e-4$, it indicts that the effect of memory contention has already

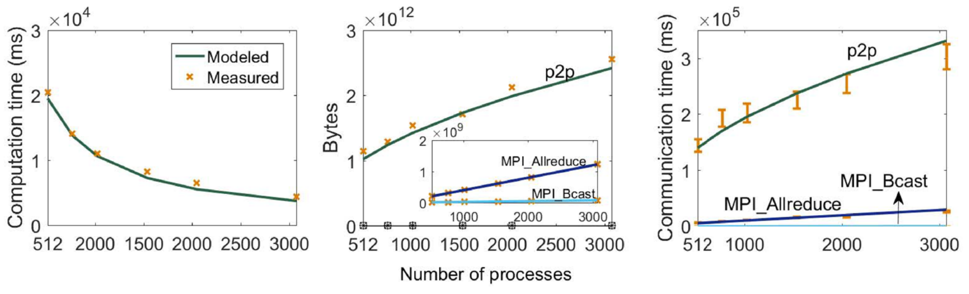


Fig. 9 Measured vs. Predicted total computation time of the *ne*256 problem size in HOMME on the Intel cluster of NSCC-Wuxi. The solid lines are predicted results using the model built from *ne*32. The marked dots are measured data

revealed in the profiling cases. Thus we have to choose suitable numbers of processes to conduct the profiling runs. We exploit the observation that the significant performance fluctuation of last level cache rarely happens.

We then use performance models built using *ne*32 to predict the run-time of problem size *ne*256 (grid number: $256 \cdot 256 \cdot 6$, vertical level: 128). We do not predict the performance per kernel because our observation shows that the compute amount increment of each kernel is inconsistent to the $ratio = \frac{ne256}{ne32}$, but highly depends on its inputs. Users with little domain knowledge are hard to estimate the increment without profiling. Therefore, we predict the computation time for the overall computation rather than functions' time to give a total run-time for large problem size. Taking the 1536 MPI rank prediction as an example, the average computation amount of 1536 MPI ranks (problem size *ne*256) is $\frac{256 \cdot 256 \cdot 6}{1536} = 16$ grids. Thus we can find the model prediction using the *ne*32 computation performance model with 16 grids (Fig. 3). With this method, the estimated computation run-time for *ne*256 is in Fig. 9, which shows a satisfying accuracy of our proposed performance model.

For the communication part, we estimated the total communication volume by a factor of $\frac{256}{32}$. Thus, the p2p communication model for large problem size is equal to $t = a \cdot ratio \cdot s + b$, where s represents the total communication volume of small problem size. The collective communication model is $t = a \cdot log(P) + b \cdot ratio \cdot s + c$. As Figs. 8 and 9 show, the communication time is measured twenty times and we can see that our model can capture the key performances. Therefore, one may focus on reducing p2p communication volume, and improving computation and communication overlap in the next step.

NAS Parallel Benchmarks. We use a 4-process run, a 9-process run, and a 16-process run to collect the profiling data for BT and SP because they need a square number of processes. We use a 2-process run, a 4-process run, and an 8-process run to collect the profiling data for the rest of NPB benchmarks.

BF_comm in NPB equals to 1 because they follows the well-known BSP model (Stewart 2011). We take the

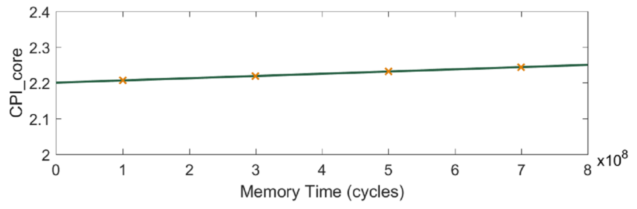


Fig. 10 Refined CPI_core_i and BF_mem_i of CICE on 4-node cluster

Table 4 Model refinement of BT benchmark on CPI_core_i and BF_mem_i

	CPI_core_i		B_mem_i	
Kernels	Before	After	Before	After
1	2.26	2.20	10^{-2}	10^{-5}
2	1.58	1.50	10^{-2}	10^{-7}
3	1.62	1.56	10^{-3}	10^{-6}

timing analysis of a typical HPC application, BT, as an example. The BT benchmark has eight steps in one super time-step, four for computation and the others for communication. In the communication steps, the communication along the x, y and z axes are MPI p2p communications and the communications across all grids performs MPI collective communications (MPI_Allreduce and MPI_Bcast). Following its timelines, the total run-time of each process in one super time-step can be estimated from the accumulation of computation and communication times, which works for a lot of today's real-world HPC applications.

BT. The largest model error is originally 18% for the BT kernel for the scales less than 40 cores. The computation time takes 80% of the total run-time. We refine the *CPI_i* and BF_mem_i for three kernels in BT. We change the CPU frequency from 3GHz to 2GHz and measure the *CPI* and *T_mem*. Here, we use *newCPI*_1, and $newT\_mem_1$ to represent the newly measured data at 2GHz. *CPI*_1 and T_mem_1 are the data at 3GHz. According to Eq. 5, we can calculate

$$CPI\_core_1 = \frac{CPI\_1 - newCPI\_1}{T\_mem_1 - newT\_mem_1}$$
$$BF\_mem_1 = \frac{CPI\_1 - CPI\_core_1}{T\_mem_1} \tag{6}$$

As Fig. 10 shows, the y-intercept of the line is the refined CPI_core_1 and the gradient is the refined BF_mem_1. Table 4 lists our refined results. And the model errors can be reduced to less than 7% (Table 3).

EP. EP has a better scaling performance compared to the other kernels. Because the communication time is extremely small and the magnitudes of BF_mem_i for EP is 10^{-7}.

FT. There are numerous MPI_Bcast and MPI_Reduce calls in FT. Although the impact of MPI communication is not significant for the scales less than 128 cores, the communication time is increasing as the number of processes grows. This indicates that when FT is scaled, collective MPI may become its performance bottleneck.

MG and SP. There are numerous p2p communications in MG and SP, and they become communication-bound when the number of processes is larger than 20. It indicates that the optimization of p2p communications for MG and SP can improve their run-time performance.

LU. LU is not as sensitive as MG and SP to communications. Although there are many p2p communications in LU, their total communication volumes are smaller than MG and SP. For example, the total communication volume of LU is $(6.52E+07)\cdot P+(5.92E+09)$ and $(3.05E+08)\cdot P+(3.15E+09)$ for SP. Therefore, the total run-time of LU first decreases and then increases more slowly than MG's and SP's run-time performance.

CG. BF_mem_i remains at 10^{-7}, and the calculation time decreases rapidly with the increased parallelism. Both make the total computation time decrease. When the number of processes is larger than 50, the point at which computation time approaches zero, then communication dominates performance.

5 Comparisons and discussions

A typical work is the domain language-based performance modeling Spafford and Vetter (2012) which requires domain experts write the performance model for each compute kernel using domain language ASPEN. Such performance model can capture the detailed implementation of applications. However, for most of the users who want to build the performance models, it makes no difference compared to building the models by learning the complex source code. Besides, users has to learn a new domain language to build the model.

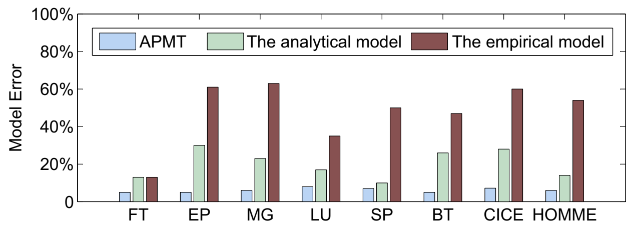


Fig. 11 Strong-scaling model error comparisons among our model, the analytical model Nan et al. (2014), and the empirical model Worley et al. (2011) on the 4-node cluster using 128 processes

By learning the algorithms and implementation of the application, a manual analytic performance model is built (Nan et al. 2014). Their computation model is built with problem size and number of processes: $T\_compute = a\cdot(\frac{problem\ size}{P})+b$. To formulate the equation, they manually analyzed the source code of the application and building this kind of performance model is very labor- and time-intensive. In addition, such case-by-case modeling suffers from portability across applications. As Fig. 11 shows, we reduce the model error up to 25% for CICE's (small problem size) strong scaling tests compared to the manual analytic performance model (Nan et al. 2014).

With less domain knowledge taken into account, empirical performance modeling profiles applications with varying configurations and uses regression to build performance models from the desired fitting functions (Bauer et al. 2012; Worley et al. 2011; Lee et al. 2007). However, choosing the correct fitting functions is critical for the performance model accuracy. The best set of such functions is hard to determine due to the increasing complexity of real-world applications running on modern hardware platforms. As one solution for this issue, machine learning techniques have been introduced to predict the execution time from observed performance results of a large number of application runs (Lee et al. 2007). It usually takes a long time to profile and train the model, especially for the cost-expensive applications (Craig et al. 2015).

An empirical model Worley et al. (2011) is built based on a variation of Amdahl's law, $T\_app = a/P + b\cdot P^c + d$, to estimate the whole application run-time. The term a/P represents the time spent in the perfectly scalable portion of the application, The term $b\cdot P^c$ is the time spent in a partially parallelized portion, such as initialization, communication, and synchronization. The term d indicates the time spent in the non-parallelized portion. This model generates accurate predictions, but it lacks performance insights. For example, when the application time does not decrease anymore, the model cannot tell the leading cause (calculation, memory, or communication). As Fig. 11 shows, we reduce the model error by up to 52% compared to the empirical model (Worley

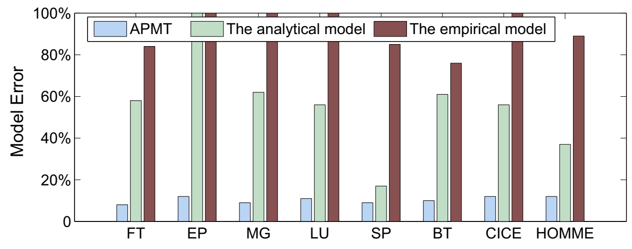


Fig. 12 Large problem size model error comparisons among our model, the analytical model (Nan et al. 2014), and the empirical model Worley et al. (2011) on the 4-node cluster using 128 processes

et al. 2011). We provide a relatively fine-grained performance model with the consideration of architecture-level details for critical kernels.

Making accurate run-time predictions for different problem size is even more challenging since the performance behavior may change obviously with different input size. Compared to the analytic performance model (Nan et al. 2014), we improve large problem size run-time prediction accuracies by 50% on average with CICE and NPB. The empirical model Stewart (2011) gets even larger model errors for large problem size than the analytic model (Nan et al. 2014), as shown in Fig. 12.

Another feasible solution is to break the whole program into several kernels with the assumption that kernels can have simpler performance behaviors (Bhattacharyya et al. 2014). There are two methods for kernel identification according to the information required Arenaz et al. (2008): application-level kernel detection and compiler-assisted kernel detection. Application-level kernels Craig et al. (2015) are often picked manually by domain experts, which cannot be used for automatic performance modeling. Craig et al. manually identified at least nine kernels of CICE with domain knowledge (Craig et al. 2015). These manual kernels do not have different representative performance behaviors due to two reasons. The first reason is that they do not separate the computation and communication parts. However, the computation and communication have different performance behaviors for modeling. The second reason is that each of the nine manual kernels contains many functions. The certain function may be invoked by different parent functions from different kernels. With our kernel identification method, we can only use seven kernels to represent the performance behavior of CICE accurately.

The compiler-assisted kernel detection usually comes with online profiling. Bhattacharyya et al. use on-line profiling and automatic performance modeling to reduce the temporal and spatial overhead of fine-grained kernel identification and modeling (Bhattacharyya et al. 2014). However, this loop-level kernel identification will introduce as many kernels to be instrumented and modeled as there are loops. This can be hundreds for the NAS parallel benchmarks as listed in column "Lker" Table 3. Bhattacharyya et al. (2015) tried to use static code analysis to detect and fuse similar kernels. But it is not effective to handle the complex loops and functions. Moreover, the fine-grained kernel modeling with statistical regression for each kernel (loop or function) makes the performance results hard to interpret. As shown in Table 3, we can achieve the required model accuracy (around 10%) while using up to an order of magnitude fewer kernels, compared to loop-level kernel modeling (Bhattacharyya et al. 2014).

Pallipuram et al. (2015; 2014) uses regression-based method combined with an analytical model framework: $T\_comp = a * FLOPS + b * BYTES + c$, where $FLOPS$ and $BYTES$ are derived from hardware counters. They argue that the hardware counters profiling technique is not accurate. The large errors come from the short execution times of the target program. Our experiments show that we can get 96% confident intervals when the application runtime is larger than 100ms with precise event-based sampling (Sprunt 2002).

6 Conclusions

We present APMT, an automatic performance model building tool, with hardware counter-assisted profiling. The output from APMT is easy-to-understand and reasonably accurate while keeping low overheads. We believe that APMT can offer a convenient mechanism for users and developers to learn the scaling performances of complex real-world applications. Hardware counter-assisted profiling is used to quantify important performance characteristics. The combination of these hardware counter can be easily interpreted to motivate future code optimization. We demonstrate that APMT not only can improve the performance model's productivity but also successfully accelerate the performance model building procedure. Our performance model errors are 15% on average. Compared with the well-known analytical model and the empirical model, APMT can improve the model accuracy of performance predictions by 25% and 52% respectively in strong scaling tests and can get even more accuracy improvements for the run-time predictions with larger problem size. In the future, we will apply our method to understand the load imbalance and I/O performance in HPC applications.

Acknowledgements This work is partially supported by National Key R&D Program of China (Grant Nos. 2016YFA0602100 and 2017YFA0604500), and Center for High Performance Computing and System Simulation of Pilot National Laboratory for Marine Science and Technology (Qingdao).

References

Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: Hpctoolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**(6), 685–701 (2010)

Arenaz, M., Touriño, J., Doallo, R.: Xark: an extensible framework for automatic recognition of computational kernels. ACM Trans. Program Langu. Syst. (TOPLAS) **30**(6), 32 (2008)

Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., et al. The landscape of parallel computing research: a view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)

Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The nas parallel benchmarks. Int. J. High Perform. Comput. Appl. **5**(3), 63–73 (1991))

Balaprakash, P., Tiwari, A., Wild, S.M., Carrington, L., Hovland, P.D.: Automomml: Automatic multi-objective modeling with machine learning. In International Conference on High Performance Computing, pp. 219–239 (2016)

Barnes, B.J., Rountree, B., Lowenthal, D.K., Reeves, J., De Supinski, B., Schulz, M.: A regression-based approach to scalability prediction. In Proceedings of the 22nd annual international conference on Supercomputing, pp. 368–377. ACM (2008)

Bauer, G., Gottlieb, S., Hoefler, T.: Performance modeling and comparative analysis of the milc lattice qcd application su3_rmd. In Cluster, Cloud and Grid Computing (CCGrid), pp. 652–659. IEEE (2012)

Bhattacharyya, A., Hoefler, T.: Pemogen: automatic adaptive performance modeling during program runtime. In Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 393–404. ACM, (2014)

Bhattacharyya, A., Kwasniewski, G., Hoefler, T.: Using compiler techniques to improve automatic performance modeling. In Proceedings of the 24th International Conference on Parallel Architectures and Compilation. ACM (2015)

Bitzes, G., Nowak, A.: The overhead of profiling using pmu hardware counters. CERN openlab report (2014)

Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In Supercomputing, ACM/IEEE 2000 Conference, IEEE, pp. 42–42 (2000)

Calotoiu, A., Hoefler, T., Poke, M., Wolf, F.: Using automated performance modeling to find scalability bugs in complex codes. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 45. ACM (2013)

Chou, C.-Y., Chang, H.-Y., Wang, S.-T., Huang, K.-C., Shen, C.-Y.: An improved model for predicting hpl performance. In International Conference on Grid and Pervasive Computing, pp. 158–168. Springer (2007)

Clapp, R., Dimitrov, M., Kumar, K., Viswanathan, V., Willhalm, T.: Quantifying the performance impact of memory latency and bandwidth for big data workloads. In Workload Characterization (IISWC), pp. 213–224. IEEE (2015)

Craig, A.P., Mickelson, S.A., Hunke, E.C., Bailey, D.A.: Improved parallel performance of the cice model in cesm1. Int. J. High Perfor. Comput. Appl. **29**(2), 154–165 (2015)

Dennis, J.M., Edwards, J., Evans, K.J., Guba, O., Lauritzen, P.H., Mirin, A.A., St-Cyr, A., Taylor, M.A., Worley, P.H.: Cam-se: a scalable spectral element dynamical core for the community atmosphere model. Int. J. High Perform. Comput. Appl. **26**(1), 74–89 (2012)

Doweck, J.: Inside intel® core microarchitecture. In Hot Chips 18 Symposium (HCS), pp. 1–35. IEEE (2006)

Gamblin, T., Schulz, M., de Supinski, B.R., Wolf, F., Wylie, B.J.N. et al. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of mpi programs. In Parallel & Distributed Processing Symposium (IPDPS), 2011. IEEE (2011)

Garcia, S., Jeon, D., Louie, Christopher M., Taylor, Michael B.: Kremlin: rethinking and rebooting gprof for the multicore age. In ACM SIGPLAN Notices, vol. 46, pp. 458–469. ACM (2011)

Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurr. Comput. Pract. Exp. **22**(6), 702–719 (2010)

Hong, S., Kim, H.: An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In ACM SIGARCH Computer Architecture News, vol. 37, pp. 152–163. ACM (2009)

Hu, P.J., Chau, P.Y.K., Sheng, O.R.L., Tam, K.Y.: Examining the technology acceptance model using physician acceptance of telemedicine technology. J. Manag. Inf. Syst. **16**(2), 91–112 (1999)

Hunke, E.C., Lipscomb, W.H., Turner, A.K., et al. Cice: the los alamos sea ice model documentation and software user's manual version 4.1 la-cc-06-012. T-3 Fluid Dynamics Group, Los Alamos National Laboratory, pp. 675 (2010)

Jayakumar, A., Murali, P., Vadhiyar, S.: Matching application signatures for performance predictions using a single execution. In Parallel and Distributed Processing Symposium (IPDPS), pp. 1161–1170. IEEE (2015)

Jones, P.W., Worley, P.H., Yoshida, Y., White, J.B., Levesque, J.: Practical performance portability in the parallel ocean program (pop). Concurr. Comput. Pract. Exp. **17**(10), 1317–1327 (2005)

Keller, R., Gabriel, E., Krammer, B., Mueller, M.S., Resch, M.M.: Towards efficient execution of mpi applications on the grid: porting and optimization issues. J. Grid Comput. **1**(2), 133–149 (2003)

Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A. et al.: Score-p: a joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In Tools for High Performance Computing, pp. 79–91. Springer, Amsterdam (2012)

Lee, Benjamin C., Brooks, David M., de Supinski, B.R., Schulz, M., Singh, K., McKee, S.A.: Methods of inference and learning for performance modeling of parallel applications. In Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 249–258 (2007)

Liang, Q.: Performance monitor counter data analysis using counter analyzer. IBM developerWorks (2009)

Malladi, R.K.: Using intel® vtune^{TM} performance analyzer events/ratios & optimizing applications. http:/software.intel.com. (2009)

Malony, A.D., Shende, S.S.: Overhead compensation in performance profiling. In European Conference on Parallel Processing, Springer, pp. 119–132 (2004)

Marathe, A., Anirudh, R., Jain, N., Bhatele, A., Thiagarajan, J., Kailkhura, B., Yeom, J.-S., Rountree, B., Gamblin, T.: Performance modeling under resource constraints using deep transfer learning. In Proceedings of the: ACM/IEEE International Conference for High Performance Computing, p. 2017. Networking, Storage and Analysis (SC), Denver, Colorado (2017)

Merten, M.C., Trick, A.R., George, C.N., Gyllenhaal, J.C., Hwu, W.W.: A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In ACM SIGARCH Computer Architecture News, vol. 27, pp. 136–147. IEEE Computer Society (1999)

Mucci, P.J., Browne, S., Deane, C., Ho, G.: Papi: A portable interface to hardware performance counters. In Proceedings of the department of defense HPCMP users group conference, vol 710 (1999)

Nan, D., Wei, X., Xu, J., Haoyu, X., Zhenya, S.: Cesmtuner: an autotuning framework for the community earth system model. In High Performance Computing and Communications (HPCC), pp. 282–289, Washington, DC. IEEE Computer Society (2014)

Pallipuram, V.K., Smith, M.C., Raut, N., Ren, X.: A regression-based performance prediction framework for synchronous iterative algorithms on general purpose graphical processing unit clusters. Concurr. Comput. Pract. Exp. **26**(2), 532–560 (2014)

Pallipuram, V., Smith, M., Sarma, N., Anand, R., Weill, E., Sapra, K.: Subjective versus objective: classifying analytical models for productive heterogeneous performance prediction. J. Supercomput. **71**(1) (2015)

PMU Intel. Profiling tools. https://github.com/andikleen/pmu-tools

Spafford, K.L., Vetter, J.S.: Aspen: a domain specific language for performance modeling. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp 84. IEEE Computer Society Press (2012)

Sprunt, B.: Pentium 4 performance-monitoring features. Micro IEEE **22**(4), 72–82 (2002)

Stewart, A.: A programming model for bsp with partitioned synchronisation. Formal Aspects Comput. **23**(4), 421–432 (2011)

Treibig, J., Hager, G., Wellein, G.: Likwid: a lightweight performance-oriented tool suite for x86 multicore environments. In Parallel Processing Workshops (ICPPW), pp. 207–216. IEEE (2010)

Van den Steen, S., De Pestel, S., Mechri, M., Eyerman, S., Carlson, T., Black-Schaffer, D., Hagersten, E., Eeckhout, L.: Micro-architecture independent analytical processor performance and power modeling. In Performance Analysis of Systems and Software (ISPASS), pp. 32–41. IEEE (2015)

Weaver, V.M.: Linux perf_event features and overhead. In: The 2nd International workshop on performance analysis of workload optimized systems, FastPath, vol 13 (2013)

Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)

Worley, P.H., Craig, A.P., Dennis, J.M., Mirin, Arthur A., Taylor, M.A., Vertenstein, M.: Performance of the community earth system model. In High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–11. IEEE (2011)

Wu, Q., Mencer, O.: Evaluating sampling based hotspot detection. In International Conference on Architecture of Computing Systems, Springer, pp. 28–39 (2009)

Xingfu, W., Lively, C., Taylor, V., Hung, C.C., Chun Y.S., Katherine, C., Steven, M., Dan, T., Vince, W.: Multiple metrics modeling infrastructure. Springer, MuMMI (2014)

Zaparanuks, D., Jovic, M., Hauswirth, M.: Accuracy of performance counter measurements. In Performance Analysis of Systems and Software, ISPASS (2009)