



Using hardware counter-based performance model to diagnose scaling issues of HPC applications

Nan Ding^{1,2} · Shiming Xu³ · Zhenya Song^{2,4} · Baoquan Zhang⁵ · Jingmei Li⁶ · Zhigao Zheng⁷

Received: 5 December 2017 / Accepted: 18 April 2018
© The Author(s) 2018

Abstract

Performance diagnosing for HPC applications can be extremely difficult due to their complicated performance behaviors. One hand, developers used to identify the potential performance bottlenecks by conducting detailed instrumentation, which may introduce significant performance overheads or even performance deviations. On the other hand, developers can only conduct small numbers of application runs for profiling the performance with the limitations on both computing resources and time duration. Meanwhile, the performance bottlenecks of HPC applications may vary with the degree of parallelism. To address these challenges, our paper proposes a systematic performance diagnosing method focusing on building an accurate and interpretable performance model with performance counters. Our method is able to diagnose the HPC application scaling issues by predicting its runtime and performance behaviors in different functions. After applying this modeling method on three real-world HPC applications, HOMME, CICE and OpenFoam, our evaluations show that our diagnosing method based on the performance model has the ability to diagnose the potential scaling issues, which is typically missed by the traditional performance diagnosing method and achieves about 10% prediction errors in a scale of 4096 MPI ranks on two problem sizes.

Keywords Performance modeling · Hardware counter · Performance issue

✉ Zhenya Song
songroy@fio.org.cn

Nan Ding
dingnan0701@gmail.com

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

² Laboratory for Regional Oceanography and Numerical Modeling, Qingdao National Laboratory for Marine Science and Technology, Qingdao 266237, China

³ Department of Earth System Science, Tsinghua University, Beijing 100084, China

⁴ First Institute of Oceanography, State Oceanic Administration, Qingdao 266061, China

⁵ Department of Computer Science and Technology, University of Minnesota, Twin Cities, Minneapolis 55454, USA

⁶ Department of Computer Science and Technology, Harbin Engineering University, Harbin 150001, China

⁷ Department of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

1 Introduction

Ever-growing supercomputers lead to more and more processing units. The scalability is regarded as one of the most important design objectives of HPC applications [1–5]. To diagnose an application's scaling performance, developers have to profile the applications several times on a given number of processes with instrumentation, which may introduce more than 10% overhead for each profiled application run [6, 7] as well as the possible performance deviation. At the same time, the detected performance issues using a specific number of processes may not still be issued when scaling the application. Even conducting laborious analysis on a few selected kernels of the whole application may lead to high risk of missing crucial bottlenecks. Besides, developers may not diagnose applications on real systems because they only have limited opportunities to access the large-scale machine.

Performance modeling is an effective way to understand the scaling performance behaviors and identify the scaling issues earlier. Compared to the traditional diagnose method, performance modeling method is able to predict

the scaling performance behaviors by profiling on small-scale parallelisms. Alexandru et al. [1, 2] proposed a performance model to find potential kernels may become bottlenecks when scaling to a large number of processes. However, the kernels they detected do not separate communications and computations. The detected kernels all contain communications. Such results are easily predicted since the time cost of communication usually increases with the growing number of processes. Thus, the computation code sections that do not scale well are still hidden in the complex code.

Analytical performance modeling [8–11] has ever been widely used to predict application execution time by analyzing the source code and algorithms. It utilizes algorithm details such as iteration times and the number of key variables, to diagnose the scaling runtime of each domain knowledge-oriented kernels. However, such method is highly depended on the algorithm and implementation. Such case-by-case performance modeling approach is very time- and labor-consuming, as well as lacking the portability across applications.

Statistical model [12, 13] tries to overcome the disadvantages by predicting the scaling performance according to a large number of sampled application runs without digging into the source codes. However, it introduces a large number of application runs to train the performance model for achieving satisfying accuracy.

To address the challenges, we propose a resource-based performance modeling method which has the relatively good portability across applications and low diagnose overhead. We diagnose applications' scaling issue using the performance model-based method with consideration of both resource usages and interactions between applications and the hardware.

To summary, our contributions are:

1. *Employing resource-based modeling method to predict large-scale/problem size application runtime using small configuration application profiling efficiently and effectively* We propose a resource-based performance modeling technique to predict the scaling runtime and architecture-oriented behaviors of each function. The general idea of our model is to build the performance model from the architecture and system perspective and identify the key and basic performance factors. For compute kernels, these factors are the different types of operations such as executed instructions, loads and stores. We predict these basic operations of each function and combine these predictions into an performance model for the whole application on the target system. For the communication part, the factors are the number and the size of messages. We measure them using the standardized PMPI interface, which is portable across all MPI

implementations. And then use LogGP model [14] to predict the communication time. We can build the performance model for different HPC applications with the same modeling methodology and give a comprehensive timing breakdown of the target application as well as the architectural behaviors of each function, without introducing more overhead and potential performance deviation.

2. *Employing performance model-based guidance for performance diagnostics* Given the fact that users are usually interested in the causes and positions of performance bottlenecks, for example, it is less helpful to tell that a program execution takes square time than pointing to the set of most expensive kernels in the program, and their scaling issues. We use performance modeling technique to predict the application runtime with key performance characters and to discover the performance issue beforehand on three real applications, HOMME [15], CICE [16] and OpenFoam [17]. By using our model, we apply (1) strong scaling diagnostics, (2) weak scaling diagnostics, (3) non-scalable compute kernel diagnostics on HOMME and CICE. And we apply (1) strong scaling diagnostics, and (2) load imbalance diagnostics on OpenFoam.

With these efforts, our performance diagnosing method is feasible to provide scaling performance insights and identify the potential performance issues in production-level quality with interpretability and low overhead. We believe our method can be further elaborated to conduct a variety of deeper analyses and possible optimizations for the complex code base.

2 The performance diagnostic framework

The primary objective of our method is to identify the potential scaling issues as well as the corresponding code functions. We use a resource-based modeling alongside time approach for the scaling diagnostics. In our work, we split the parallel application into computation and communication two parts and then predict the computation and communication performance separately and the accumulation of these two part is the final runtime prediction.

The computation runtime equals the accumulation of all compute kernels' runtime. We use hardware counters to profile the computation scaling characteristics to overcome the large overhead of instrumentations [18]. The communication time is the accumulation of point-to-point (p2p) MPI operations and collective MPI operations. We measure MPI operations using the standardized PMPI interface, and then use LogGP model [14] to conduct the communication time prediction. Such method is portable across applications. The framework includes the following four

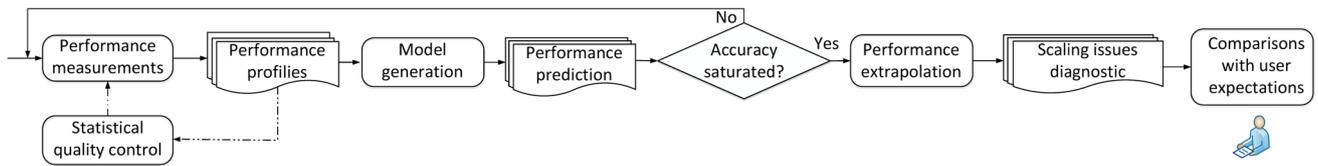


Fig. 1 Workflow of scaling issue diagnostics. Solid boxes represent actions, and banners are their inputs and outputs. Dashed arrows are optional actions taken by user decisions

key modules. Figure 1 gives an overview of the key steps necessary to predict runtime and potential performance issues.

1. Performance measurements We generate the hardware counter profiles for computation using the driver of Intel VTune, which records the executions times plus various hardware counters, such as the number of CPU cycles waiting for memory, the number of executed instructions. All metrics are broken down by call path and MPI ranks. Here, we define each element on the call path is a function without including its children. We use precise event-based sampling (PEBS) [19] for hardware counter profiling. The PEBS mechanism is armed by the overflow of the counter with a precise program counter address at granularity of functions. Thus we can obtain the function list after the profiling runs, together with the hardware counter information. However, manual instruction can be added to profiling a lower-level performance, such as loops. We choose functions as kernels because it is clearly to separate communication and computation in parallel applications to enable us to predict them separately. Assume that we use two MPI ranks in one compute node. The hardware counter profiling results of each function are the total amount of the two MPI ranks. We generate communication traces, namely the message size, the message count, the source and destination, by using the standardized PMPI interface. Figure 2 illustrates how we generate the performance measurements on hardware counters and MPI operations.

To summary, the metrics we collected include resource-based metrics and time-based metrics, as Tables 1 and 2 show. Resource-based metrics, such as number of instructions, or the number of messages sent/received, are usually a function of application configuration, and therefore deterministic. We define them as resource-based metrics because they can reflect the resource utilization of a program on the target platform. The resource-based metrics play an important role in our method. Because they can be used to determine the model functions easily, and can be used to predict large-scale performance easily. The model functions are usually a set of pre-defined functions that potentially can be used as a good approximation. Time-based metrics, such as communication time and

computation time, are used to determine the coefficients of our model functions. We conduct the performance measurements on several parallelisms. The profiling results of these parallelisms are used to determine the model functions and verify the approximation.

2. Profiling validity 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.

Hardware counter profiling validity can be controlled by its overflow intervals. Our observations show that the default overflow interval, namely 2000 thousands CPU cycles can have 96% confidence intervals on identifying the functions in the program for a 100 ms application run.

3. Resource-based modeling alongside time After performance measurements, a performance model framework is generated according to Table 2 to convolute the communication and computation performance (Eq. 1). In this paper, we assume that applications are implemented following the Bulk Synchronous Parallel (BSP) programming model [20]. Follow the BSP programming model, the total runtime can be calculated by the accumulation of computation runtimes and communication runtimes, which can be found correct in most of today's real-world HPC applications.

$$T_{app} = \sum_{i=1}^n (T_{comp_i} + BF_{mem_i} * T_{mem_i}) + BF_{comm} * T_{comm} + T_{others} \quad (1)$$

The computation part consists of a number of kernels. The computation time of a kernel can be estimated by two components. The first part is the time taken to execute the computation instructions. The second part is the time associated with fetching the data from storage to compute [21]. There is typically some degree of overlapping 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 variable i is the index of the detected functions. The accumulation of the p2p and the collective communication times is used to predict the total

Fig. 2 An example of performance measurement

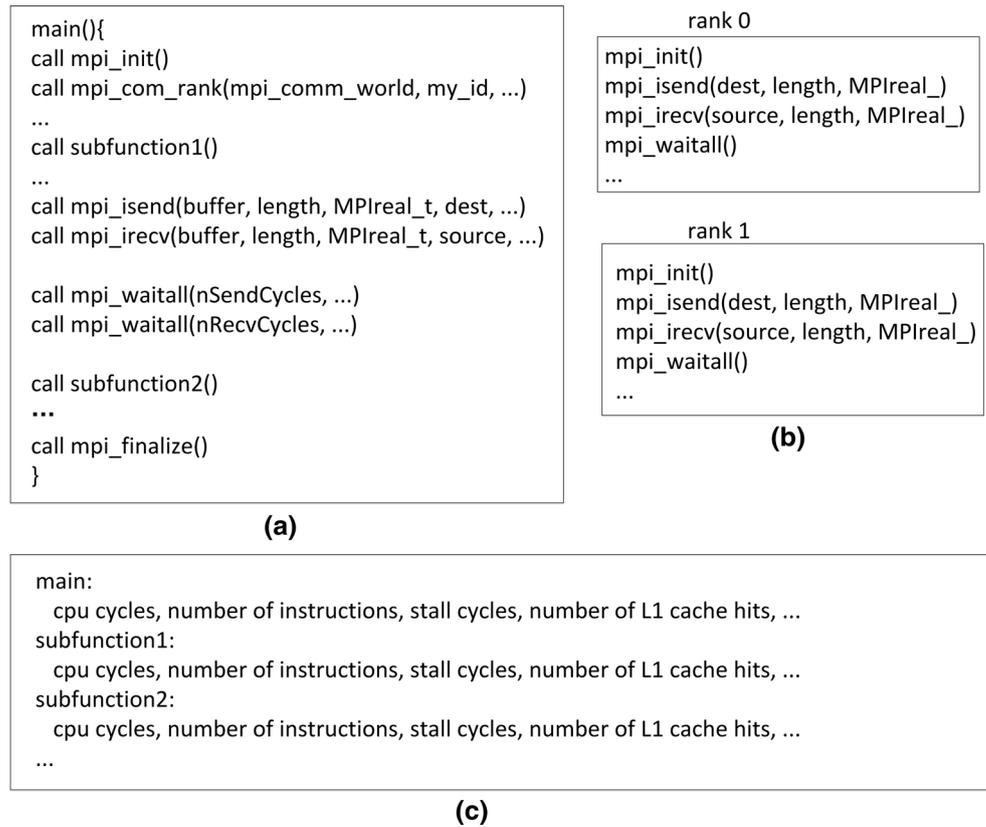


Table 1 Resource-based metrics and how they are derived from hardware counters

Resource-based metrics	Description	How are they derived?
T_{stall_i}	Waiting time for memory	Fitting from RESOURCE_STALLS.LB(SB) counter
T_{L1_i}	L1 cache access time	Fitting from MEM_LOAD_UOPS_RETIRED.L1_HIT_PS counter
T_{L2_i}	L2 cache access time	Fitting from MEM_LOAD_UOPS_RETIRED.L2_HIT_PS counter
T_{LLC_i}	Last-level cache access time	Fitting from MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS counter
$T_{mainmemory_i}$	Main memory time	Fitting from MEM_UOPS_RETIRED.ALL_LOADS(STORES)_PS counter
Instructions	Executed instructions	Fitting from INST_RETIRED.ANY_P counter
$T_{collective}$	Collective MPI communication	Fitting from P and S
T_{p2p}	Point-to-point MPI communication	Fitting from S
T_{others}	Initialization and finalization time	Fitting from P
Definitions	Description	How are they derived?
S	Total communication volume	Measured
P	Number of processes	Input
D	Problem size	Input
n	Total number of kernels	Detected

communication time. A variable named BF_{comm} is measured to evaluate the non-overlapped communication and computation time. Below, we explain how we build the

resource-based performance models, and match them with time-based metrics in detail. CPI is regarded as a comprehensive indicator for the performance of kernels, but it

Table 2 Time-based metrics used to derive the model

Time-based metrics	Description	How are they derived?
T_{comp_i}	Calculation time of each kernel	$T_{comp_i} = \frac{instructions_i * CPI_i}{CPUfrequency * P}$
T_{mem_i}	Total memory time	$T_{mem_i} = T_{L1_i} + T_{L2_i} + T_{LLC_i} + T_{mainmemory_i}$
BF_{mem_i}	Ratio of non-overlapped memory time	$BF_{mem_i} = \frac{T_{stall_i}}{T_{mem_i}}$
BF_{comm}	Ratio of non-overlapped communication time	$BF_{comm} = \frac{T_{mapp} - T_{mcomp}}{T_{mcomm}}$
T_{comm}	Average communication time	$T_{comm} = \sum_{i=1}^r T_{p2p} + \sum_{i=1}^l T_{collective}$
CPI_i	Cycles per instruction	Measured
T_{mapp}	Total application time	Measured
T_{mcomp}	Total computation time	Measured
T_{mcomm}	Total communication time	Measured

alone cannot help us to identify the different performance patterns of kernels since the similar CPIs of kernels can be gained from lower instruction efficiency and better memory traffic, or better instruction efficiency and poorer memory traffic.

The general idea of a resource-based performance model is to account for all key cost factors. For the computation, as Table 1, model terms, such as T_{stall_i} , T_{L1_i} , T_{L2_i} , T_{LLC_i} and $T_{mainmemory_i}$ are derived from different types of hardware counters listed in the third column.

Take T_{stall_i} term as an example, we show how to use hardware counters derived it. In an ideal case, all CPU cycles should be spent to produce useful work. However, there always exists certain part of CPU cycles is spent stalling. At hardware level, the stalling time (T_{stall_i}) represents the waiting time for memory, namely the result of unavailability of processing units or data. Such scenarios often degrade the performance of an application as the number of cores increases [22]. We use the accumulation of two hardware counters, namely *RESOURCE_STALLS.LB* and *RESOURCE_STALLS.SB* to calculate the stall cycles. In this paper, we use five different process configurations for profiling set, and the sixth parallelism is used to verify the regression results. We do the regression for the hardware counters with the best model function in our pre-defined model function set, including one order of polynomial, exponent and logarithm [1, 2]. Because based on massive analysis and experiments, those three kinds of functions and their combinations can cover most of the situation.

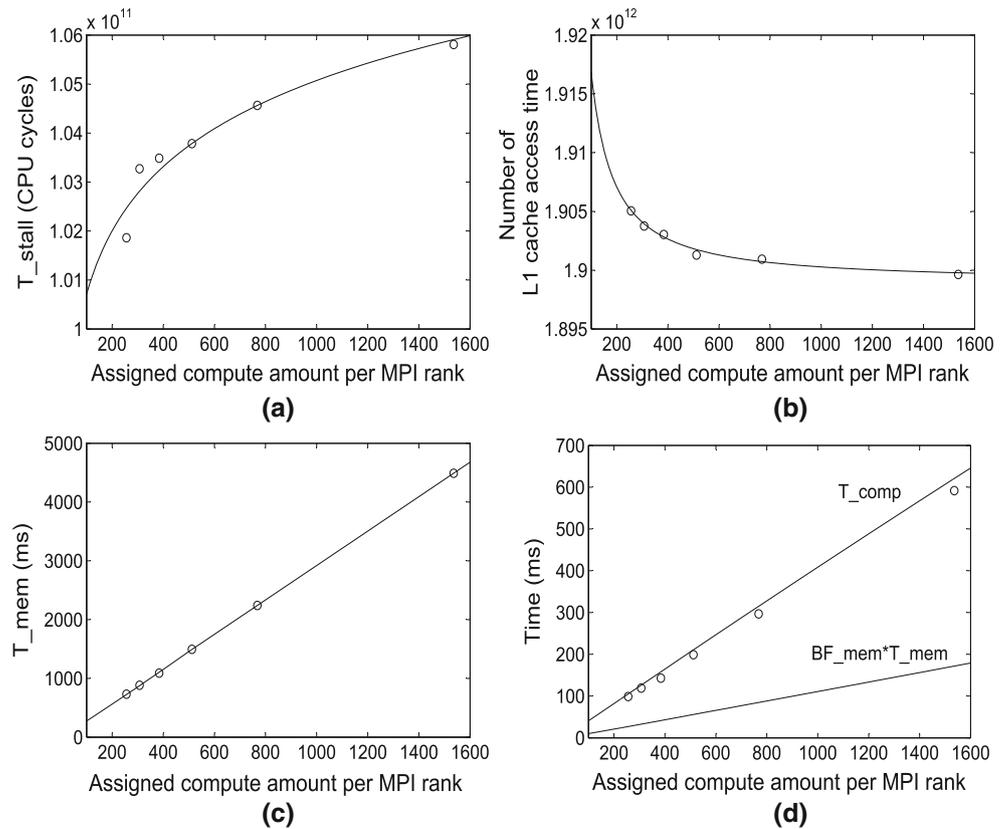
Figure 3 is the model results of a kernel called *laplace_sphere_wk* from HOMME. Figure 3a is the accumulation of *RESOURCE_STALLS.LB* and *RESOURCE_STALLS.SB* (equals to T_{stall}), and Fig. 3b is the number of L1 cache access time. The x -axis is the average compute amount per MPI rank. We do the

regression fitting according to the average compute amount because it is easier to predict the performance for another problem size. We will explain it later in the paper. We use one-order power equation, $a \cdot x^b + c$, for the T_{stall} and number of L1 access time. The model parameter b is positive for T_{stall} while is negative for the number of L1 access time. It represents that as we assign more MPI ranks in one node, the waiting time for memory (T_{stall}) decreases, and we can fetch more data from L1 cache. Such performance insights can help the users draw a conclusion that the kernel is computation driven in the present cases. According to Table 1, we can model the T_{mem} and T_{comp} as Fig. 3c and d shows. Through the model term BF_{mem} , users can obtain the non-overlapped memory time. Therefore, if users want to assign more MPI ranks in the node, they should put their effect on optimizing the memory. Because the total computation time T_{comp} becomes closer to $BF_{mem} \cdot T_{mem}$.

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 m of size s equals to $t = m \cdot (a \cdot s + b)$. We model the p2p communication time t with total communication size ts as $a \cdot ts + b$. For the collective communications, the time cost t of a broadcasts 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 modeling each individual message sizes, we use an average message size among processes.

To summary, we first model the resource-based metrics separately for each function. We then match the resource-based metric of the functions with time-based metrics using the best one in our pre-defined linear fitting functions. The accumulation time of the compute kernels plus the communication time is the total estimated application runtime. The model inputs are the problem size and number of

Fig. 3 An example of performance modeling the resource-based metrics along time. 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



processes. The outputs are the predicted runtime of each kernel and estimated scaling resource-based metrics.

4. Scaling performance prediction Our model predicts: (1) the execution time T_{app} of a given parallel application on a target scale P_t by using several applications runs with q processes, where $q \in \{2, \dots, P_0\}$, $P_0 < P_t$; (2) the execution time T_{app} of a given parallel application on a target scale P_k with a different problem size D_s . Once the model coefficients are determined, we can use the equations in Table 1 to predict the runtime in Fig. 3.

Our motivation is to identify the kernels from a small problem size D , and build the performance models. We then predict the application runtime for large problem size D_s without running the application of D_s .

The prediction for another problem size D_s is conducted according to the average compute amount per MPI rank. Of course, the memory resource contention plays an important role in runtime performance, especially the number of main memory access time. In this paper, we define a threshold $E = \left\| \frac{N_{LLCmiss_{i_p}}}{N_{totalmem_{i_p}}} \right\|$ to evaluate the effectiveness for D_s runtime prediction from the harm of memory contention. $N_{LLCmiss_{i_p}}$ represents the number of last-level cache misses of function i with number of process p , and $N_{totalmem_{i_p}}$ is the total number of memory access. Thus

by using the profiling results of the five parallelism mentioned above, if $E \leq 1e - 4$, it indicates that the performance of last-level cache does not fluctuate much when assigns different compute amount to each MPI rank. Otherwise if $E > 1e - 4$, it indicates that the affect of memory contention has already 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 big performance fluctuation of last-level cache rarely happens. Besides, it is a common sense in HPC applications that one should choose a suitable number of processes for an application run following certain rule. For example, the number of processes should follow $6 \cdot n^2$ when we run an atmosphere model with the spectral element dynamical core.

Take HOMME as an example, we predict the runtime of problem size $ne256$ (grid number: $256 \cdot 256 \cdot 6$, vertical level: 128) on a target scale 1536 MPI ranks from $ne32$ ($32 \cdot 32 \cdot 6$, vertical level: 128). We do not predict the performance per function because our observation shows that the compute amount increment of each function is inconsistent to the $ratio = \frac{D_s}{D}$, but highly depends on its own 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 overall runtime for large problem size. The average compute amount of 1536 MPI ranks (problem size $ne256$) is 16 grids. Thus, we can find how the resource-based metrics behave when we assign 16 grids in one MPI rank by using the models built from $ne32$. For the communication, we estimate the total message size among processes according to its data decomposition which can be learned from the application's technical report and its building scripts.

3 Evaluation

The experiments are carried out on the Intel cluster in National Supercomputing Center in Wuxi of China (NSCC-Wuxi). 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 is Intel MPI 15.0, and the network is Mellanox[®] FDR InfiniBand. The time of I/O part isn't taken into consideration in our experiments.

For all the experiments, the profiling overhead of our framework is 3% on average. As we know, the overhead of instrument-based profiling tools, such as Score-P [6], and Scalasca [7], usually exceeds 10%.

3.1 HOMME

HOMME [15] is the dynamical core of the Community Atmosphere System Model (CAM) being developed by the National Center for Atmosphere Research (NCAR). The two cases we used are as listed in Table 3.

3.1.1 Strong scaling diagnostics

Figure 4 presents the communication performance of $ne32$ problem size based on six profiling and evaluation data points with the number of processes $P = \{12\ 24\ 36\ 48\ 96\ 120\}$. The sub-figure on the left is the communication time, and the right one is the total communication volume. We first extrapolate these data up to 768 processes for the same

problem size ($ne32$). In this paper, model error is defined as $(measuredtime - predictedtime)/(measuredtime * 100)$. The error of our model is 12% on average. The communication model is built with resource-based metric: total communication size. Point-to-point communication time consumes the most communication time followed by the communication of MPI_Allreduce. According to the right sub-figure, we can infer that the p2p time suffers from the large communication volume. The boundary updating occurs in the halo regions for each process in every time step. Every process has to communicate to its eight neighbors. The total communication volume grows with \sqrt{P} . MPI_Allreduce is ranking the second in MPI time. By using our performance model, we can find that MPI_Allreduce comes from the diagnostics in *prim_driver_mod.F90*, which can be turned off during application runs. Therefore, to achieve a better performance, users can focus on reducing p2p communications volume by reducing the p2p communication invoking count or reducing the size directly.

As mentioned in Sect. 2, Fig. 3 is the strong scaling computation model results. The computation model error is around 6%.

3.1.2 Weak scaling diagnostics

For the large problem size $ne256$, our motivation is to predict its performance using the model built from $ne32$ instead of running and profiling $ne256$ cases. *Ratio* of total communication volume of different communication patterns is decided by domain knowledge in our model. For p2p, the increment of communication volume is $ratio = \frac{256}{32}$ (the right sub-figure in Fig. 5), therefore the p2p time of $ne256$ equals to $a \cdot (ratio \cdot size_{ne32}) + b$. Similarly, the collective communication is predicted with $a \cdot \log(P) + b \cdot (ratio \cdot size_{ne32}) + c$. Different from p2p communications, *ratio* of MPI_Bcast is 1 and for MPI_Allreduce it is $\frac{2 \cdot 256}{32}$. Figure 5 is the communication performance of $ne256$. The most expensive communication kernel is still p2p communication and the maximum prediction error for large problem size is 12%. With the guidance of our model, the p2p communication should be put in the first place when tuning the code.

Figure 6 is the predicted overall application runtime of $ne256$. As mentioned in Sect. 2, we conduct the prediction based on the compute amount assigned to each MPI rank. The validation data points in Fig. 6 are $P = \{512\ 768\ 1,024\ 1,536\ 2,048\ 3,072\}$. The compute amount assigned to each MPI rank is $\{768\ 512\ 384\ 256\ 192\ 128\}$. Then we can find the time cost of the certain compute amount assigned to each MPI by using the performance model built from $ne32$. We can keep the model error of large problem size prediction

Table 3 Configurations of the two HOMME cases

Case name	$ne32$	$ne256$
Horizontal grids	$32 \cdot 32 \cdot 6$	$256 \cdot 256 \cdot 6$
Numerical method	Spectral element	
Simulation method	J & W baroclinic instability	
Vertical layer	128	
Simulation time	2-h simulation	

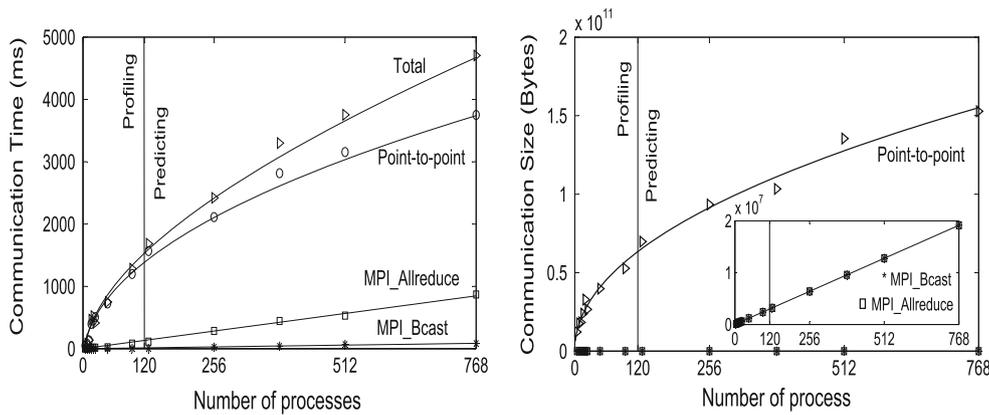
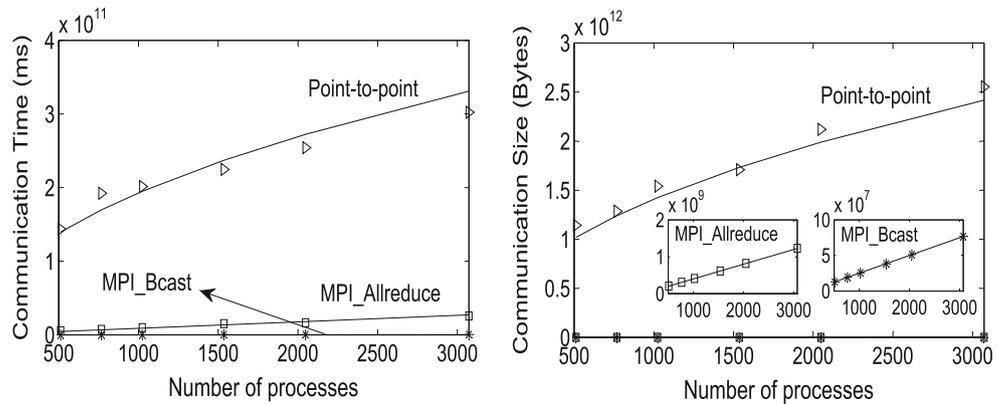


Fig. 4 Measured versus Predicted MPI time and total communication volume of the *ne32* problem size in HOMME on the Intel cluster of NSCC-Wuxi. The left sub-figure is communication time in which p2p communication consumes most of the total communication time. The right sub-figure is the communication sizes in which p2p

communication is the largest. The zoom-in figure is the communication size of MPI_Allreduce and MPI_Bcast. Thus with the model report, we can infer that p2p time is a potential performance issue and it suffers from the large communication volume

Fig. 5 Measured versus predicted MPI time and communication volume of the *ne256* problem size of HOMME. The solid lines are predicted results using the model built from *ne32*. The marked dots are measured data



10.6% on average. The prediction error comes from the load imbalance caused by the ill-considered number of processes. Because *ne256* has a total of $6 \cdot 256 \cdot 256 =$

393,216 ways of MPI parallelism in the problem. Therefore, process count $1536 = 6 \cdot 16 \cdot 16$ in Fig. 6 has a better load balance workload thus better prediction accuracy.

3.1.3 Kernel ranking diagnostics

Usually users pay much attention to optimize the performance of kernel `compute_rhs` and `euler_step`. Because they are also the key computation parts based on the domain knowledge and they truly contain a lot of floating-point operations. However, there are five other kernels that should be taken into account. They are `edgevpack`, `edgevunpack`, `laplace_sphere`, `divergence_sphere` and `vertical_remap`. Function `vertical_remap` is embarrassingly parallel. The functions `edgevpack` and `edgevunpack` are used to pack/unpack for one or more vertical layers into an edge buffer. Kernel `laplace_sphere` and `divergence_sphere` compute the gradient after communications. Figure 7 lists the top seven time-consuming kernels at different scales. Take `edgevunpack` as an example, it is compute-driven

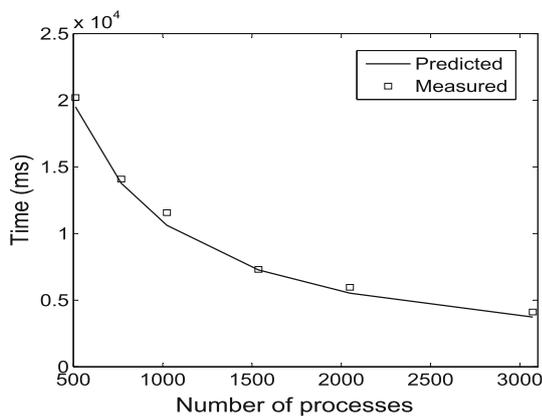


Fig. 6 Measured versus predicted total computation time of the *ne256* problem size of HOMME. The solid lines are predicted results using the model built from *ne32*. The marked dots are measured data. The overall model error is 10.6% on average

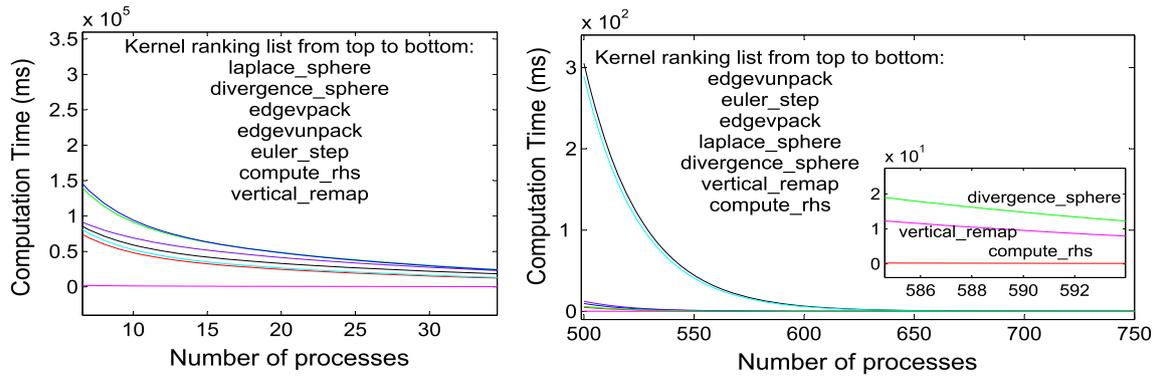


Fig. 7 Kernel ranking list of the top seven time-consuming functions at different scales of HOMME. Different colors represent different kernels. We can see that the kernel ranking list is changing according

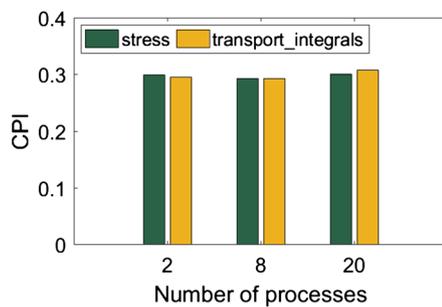
when $P < 40$ processes, and its time does not decrease anymore after the number of processes is larger than 300. That is to say, we cannot find the `edgevunpack` function is the performance critical one at large scale if we don't conduct the application run with over 300 processes. With the help of our performance model, we don't need to conduct large-scale application run but can efficiently predict the performance variation of different functions and give the correct candidates for tuning at scale.

3.2 CICE

The Los Alamos sea ice model (CICE) [16] is a widely used sea ice model in the well-known CESM project [23]. The configurations are listed in Table 4.

Table 4 Configurations of the two CICE cases

Case name	gx3	gx1
Horizontal grids	116 · 100	384 · 320
Grid decomposition method	slenderX2	
Simulation time	1 month	



to different parallelisms. The zoom-in figure is the fifth–seventh functions for a scale of 500–700

3.2.1 Strong scaling diagnostics

Function `stress` is the most time-consuming function, and `transport_integrals` is less time-consuming functions. The two functions have similar *CPIs* (as shown in the left sub-figure of Fig. 8). However, *CPI* alone cannot distinguish such performance that comes from lower instruction efficiency and better memory traffic, or better instruction efficiency and poorer memory traffic. Normally users would measure the *Flops / Bytes* to see whether it is a memory-driven program or a compute-driven program. Compared to the ratio of theoretical *Flops / Bytes*, if the measured ratio is smaller than the theoretical one, it indicates that the code is memory intensive, and then users would focus on memory optimization of `stress` (Table 5). However, with the help of our model, the memory blocking factor of `stress` scales well ($4.2e-4$ to $0.2e-4$ for a scale of 2 to 20 processes, while `transport_integral` changes from $0.9e-4$ to $0.3e-4$). The memory has a more obvious impact on kernel `transport_integral`. It is the reason that the speedup (shown in the right sub-figure of Fig. 8) of

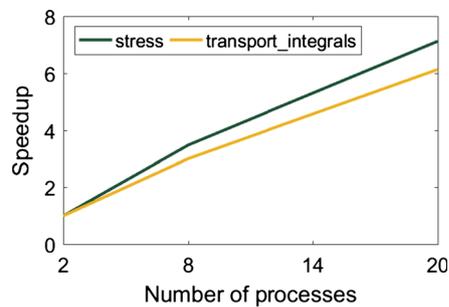


Fig. 8 The *CPIs* of the function `stress` and `transport_integral` of CICE using gx3 resolution are very similar. However, the similar *CPIs* of kernels can be gained from lower instruction efficiency and better memory traffic, or better instruction efficiency and poorer memory traffic. With the help of our model, the memory

blocking factor of `stress` scales well ($4.2e-4$ to $0.21e-4$) while `transport_integral` changes from $0.9e-4$ to $0.3e-4$. The memory has a more obvious impact on kernel `transport_integral`. It is the reason that the speedup (right sub-figure) of `stress` is better than `transport_integral`

Table 5 Theoretical flops/bytes versus measured flops/bytes

Number of cores	Theoretical	Measured
2	0.29	0.01
8	1.18	0.05
20	2.94	0.07

stress is better than `transport_integral`. By further looking up the source code, we find that `stress` is mainly a stencil computation with `add` and `multiply` operations which is considered having a relatively good overlap between computation and memory. Kernel `transport_integral` is mainly the assignment operations. Therefore, the memory impact of `stress` may not be an issue when scales to large number of processes.

3.2.2 Week scaling diagnostics

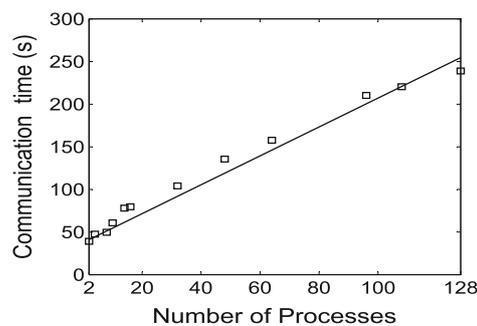
Figure 9 is the predicted results for $384 \cdot 320$ problem size using the model from $116 \cdot 100$. In CICE, p2p communication is the key part to slow down the performance while computation scales well for a scale to 128 cores. Because p2p communication happens in boundary data exchange, and every `evp` sub-cycling has to communicate the data for calculating the sea ice mask.

In addition, we find that there are two functions (`ice_timer_stop` and `ice_timer_start`) do not decrease with the number of processes. These two functions are used to control the timer start and stop to measure the timing information of each function, which are not scalable. Therefore, to achieve a better performance for large-scale runs, users may turn off the timer.

3.3 OpenFOAM

OpenFOAM [17] is an open source computation fluid dynamics (CFD) solver. The test configurations are listed in Table 6. In this example, we show how we detect the load imbalance and improve its performance with our performance diagnostic framework.

Fig. 9 Large problem size ($384 \cdot 320$) runtime prediction of CICE. The solid line is the predicted runtime by our model. The marked dots are the measured validation data

**Table 6** Configurations of the OpenFoam cases

Case name	motorBike
Cell number	$20 \cdot 8 \cdot 8$
Decomposition method	ptscotch
Simulation time	200 steps

3.3.1 Load imbalance diagnostics

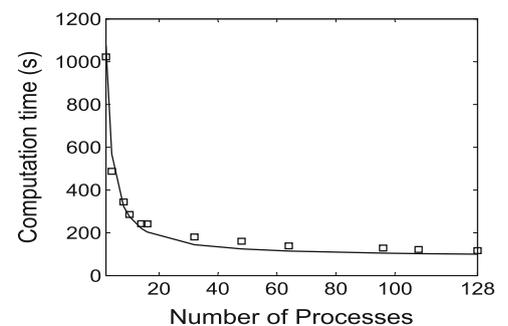
The default data decomposition is to try best to decompose the data in each dimension evenly. However, such decomposition is not always good for time-to-solution performance. For example, the default data decomposition of case `motorBike` using 12 MPI ranks is 3, 2, 2 in x, y, z directions. Its profile results show that the idle time is up to 15% of the total application runtime. According to our model, it is because in each time step the MPI ranks have to wait until all communications among three directions are done. We then change the data decomposition into 4, 3, 1 to cancel the communication with z direction which reduces 6% p2p MPI communication invoking counts. And we can achieve 15% performance improvement by changing the data decomposition with user decisions (Fig. 10).

3.3.2 Strong scaling diagnostics

With the manually decided data decomposition method, we profile the case `motorBike` with $P = \{48, 64, 96, 120, 160, 200\}$. As Fig. 11 shows, the p2p communication prediction error is 13.6% at the most, and the overall model error is 7% on average. By using the user-decided data decomposition, we can achieve 25% at the most (13.7% on average) p2p communication performance improvement, and 6% performance benefits for the overall performance.

4 Related work

Scaling performance prediction of parallel applications has numerous prior work. There are two well-known approaches. One is using a trace-driven simulation to capture detailed



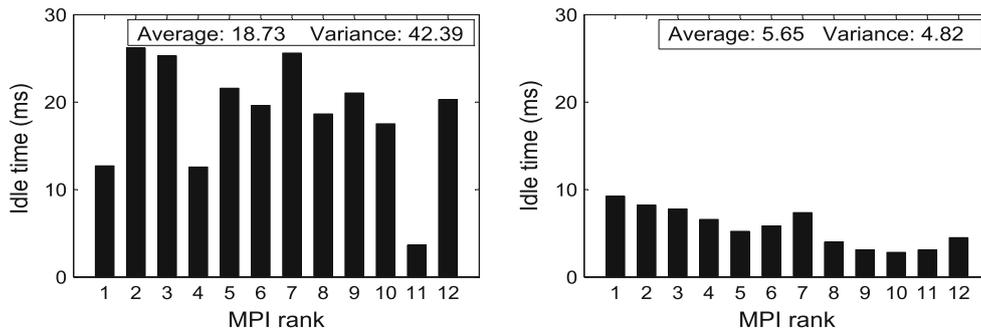


Fig. 10 We find that the idle time of data decomposition 3, 2, 2 is up to 15% of the total runtime of OpenFoam. According to our model, such long idle time comes from waiting for communication between directions x and z . We then change the data decomposition to 4, 3, 1

to reduce the communication with z direction which reduces 6% p2p MPI communication invoking counts. As shown in the right figure, the average idle time is reduced to 5.65ms, and the idle time variance is reduced to 4.82 from 42.30 with average idle time 18.72 ms

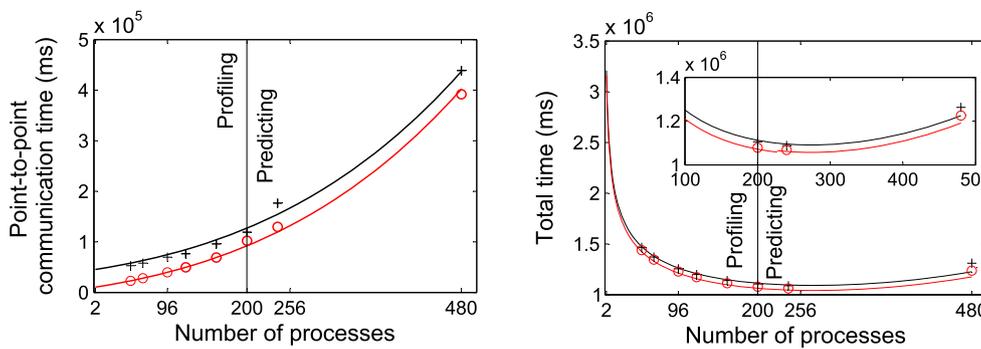


Fig. 11 The user-decided data decomposition has gained 25% at the most (13.7% on average) for p2p communication performance compared to the default data decomposition of OpenFoam. The model prediction error is 13.6% at the most. The right sub-figure is the total runtime of motorBike, the communication time reveals

gradually after 100 processes. The overall model error is 7% on average. The zoom-in figure is the total runtime from 100 processes to 480 processes. The user-decided data decomposition can improve 6% performance of the total runtime

performance behavior at a required level, such as MPI-SIM and BigSim [24, 25]. However, it is extremely expensive, not only in terms of time cost of the simulation, but especially in the memory requirement. Zhai et al. [26] extrapolate single-node performance by using representative process to reduce the memory requirements. Wu et al. [27] predict communication performance by extrapolating traces to large-scale application runs. Engelmann et al. [28] develop a simulator permits running an HPC application with millions of threads while observing its performance in a simulated extreme-scale HPC system using architectural models and virtual timing. However, all above works aim to accurately performance prediction but not provide scaling issues for the high-end users beforehand.

Another approach is using analytical performance modeling technique. It is well understood that an analytical performance model has the ability to provide performance insights of a complex parallel program [29, 30]. Hoefler et al. [31] use a simple six-step process to build a performance model for applications with detailed insights.

However, it requires tedious code analysis and not being used to predict scalability issues. Calotoiu et al. [1] use performance models to find performance scalability bugs for parallel applications in aspects of communication and floating-point operations. This is probably the most similar work with ours. The main differences are that (1) they aim to report the kernel rankings while they do not separate the computation and communication. However, such results are easily predicted since the time cost of communication usually increases with the growing number of processes. Thus, the computation code sections that do not scale well are still hidden in the complex code while in our work, we separate the communication and computation; and (2) they use communications and floating-point operations as metrics to evaluate the large-scale performance issues, while we provide the possible causes of the potential scaling issues by separating the memory effect from computations. To better understand the fine-grained performance, Bhat-tacharyya et al. [32] break the whole program into several loop kernels with the assumption that kernels can have

simpler performance behaviors. However, this loop-level kernel identification will introduce as many kernels to be instrumented and modeled as there are loops. This can be hundreds even for the NAS parallel benchmarks, and it is not effective to handle the complex loops and functions in real applications. Besides, it lacks the insights for resource consumption of each kernel. Chatzopoulos et al. [22] use the hardware counters to extrapolating the scalability of in-memory applications. There is a consensus that performance modeling technique can be an effective approach for understanding the resource consumption and scalability.

Other approaches focus less on general purpose models but rather on modeling for a specific purpose [33–35]. Martinasso et al. [36] develop a congestion-aware performance model for PCIe communication to study the impact of PCIe topology. Mondragon et al. use both simulation and modeling technique to profile next-generation interference sources and performance of the HPC benchmarks [12]. Yang et al. [37] performance modeling the applications by running kernels on the target platform and then conduct the prediction cross-platform based on relative performance between the target platforms.

5 Conclusions

Our work demonstrates that performance modeling technique with hardware counters can be used to help users to understand the potential performance bottlenecks of large-scale runs and large problem size runs. Compared to the laboriously detailed performance models by hand, our performance model is competitive to give an earlier report on the potential performance issues as well as their causes and code positions. Compared to the traditional performance diagnostic method, our method can predict the scaling performance behaviors efficiently and effectively by profiling the application runs on small-scale parallelisms rather than profiling large-scale runs several times. Our model is currently used to conduct the predictions on the same system. The projection across architectures are not considered in the current version of our model. However, the resource-based modeling alongside time makes it easy to build a new model for different architectures. Besides, our resource-based metrics can be used as a preliminary suggestion on how a system is designed to achieve the best performance for a given application.

Acknowledgements We would like to thank all the anonymous reviewers for their insightful comments and suggestions. This work is partially supported by the National Key R&D Program of China (Grant Nos. 2016YFA0602100 and 2017YFA0604500), National Natural Science Foundation of China (Grant Nos. 91530323 and 41776010). Song Zhenya's work is supported by AoShan Talents Cultivation Excellent Scholar Program of Qingdao National

Laboratory for Marine Science and Technology (No. 2017ASTCP-ES04), the Basic Scientific Fund for National Public Research Institute of China (No. 2016S03), and China-Korea Cooperation Project on the Trend of North-West Pacific Climate Change.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Calotoiu A, Hoefler T, Poke M, Wolf F (2013) 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. ACM, p 45
2. Bhattacharyya A, Kwasniewski G, Hoefler T (2015) Using compiler techniques to improve automatic performance modeling. In: Proceedings of the 24th international conference on parallel architectures and compilation. ACM
3. Wang H, Jingchao LI, Guo L, Dou Z, Lin Y, Zhou R (2017) Fractal complexity-based feature extraction algorithm of communication signals. *Fract Complex Geom Patterns Scaling Nat Soc* 25(5):1740008
4. Lin Y, Wang C, Wang J, Dou Z (2016) A novel dynamic spectrum access framework based on reinforcement learning for cognitive radio sensor networks. *Sensors* 16(10):1675
5. Lin Y, Wang C, Ma C, Dou Z, Ma X (2016) A new combination method for multisensor conflict information. *J Supercomput* 72:2874–2890
6. Knüpfer A, Rössel C, an Mey D, Biersdorff S, Diethelm K, Eschweiler D, Geimer M, Gerndt M, Lorenz D, Malony A et al. (2012) Score-P: a joint performance measurement run-time infrastructure for periscope, Scalasca, TAU, and Vampir. In: Tools for high performance computing 2011. Springer, pp 79–91
7. Geimer M, Wolf F, Wylie BJ, Abraham E, Becker D, Mohr B (2010) The Scalasca performance toolset architecture. *Concurr Comput Pract Exp* 22(6):702
8. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. *Commun ACM* 52(4):65
9. Stengel H, Treibig J, Hager G, Wellein G (2015) Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In: Proceedings of the 29th ACM on international conference on supercomputing. ACM, pp 207–216
10. Kerbyson DJ, Jones PW (2005) A performance model of the parallel ocean program. *Int J High Perform Comput Appl* 19(3):261
11. Bauer G, Gottlieb S, Hoefler T (2012) Performance modeling and comparative analysis of the MILC lattice QCD application `su3_rmd`. In: 12th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid), 2012. IEEE, pp 652–659
12. Mondragon OH, Bridges PG, Levy S, Ferreira KB, Widener P (2016) Understanding performance interference in next-generation HPC systems. In: High performance computing, networking, storage and analysis, SC16: international conference for IEEE, pp 384–395
13. Jayakumar A, Murali P, Vadhiyar S (2015) Matching application signatures for performance predictions using a single execution.

- In: Parallel and distributed processing symposium (IPDPS), 2015 IEEE International. IEEE, pp 1161–1170
14. Alexandrov A, Ionescu MF, Schauser KE, Scheiman C (1995) LogGP: incorporating long messages into the LogP model-one step closer towards a realistic model for parallel computation. In: Proceedings of the seventh annual ACM symposium on parallel algorithms and architectures. ACM, pp 95–105
 15. Dennis JM, Edwards J, Evans KJ, Guba O, Lauritzen PH, Mirin AA, St-Cyr A, Taylor MA, Worley PH (2012) CAM-SE: a scalable spectral element dynamical core for the Community Atmosphere Model. *Int J High Perform Comput Appl* 26(1):74
 16. Hunke EC, Lipscomb WH, Turner AK et al. (2010) CICE: the Los Alamos sea ice model documentation and software users manual version 4.1 LA-CC-06-012. T-3 Fluid Dynamics Group, Los Alamos National Laboratory 675
 17. Jasak H, Jemcov A, Tukovic Z, et al. (2007) OpenFOAM: A C++ library for complex physics simulations. In: International workshop on coupled methods in numerical dynamics, IUC Dubrovnik, Croatia, vol 1000, pp 1–20
 18. Uh GR, Cohn R, Yadavalli B, Peri R, Ayyagari R (2006) Analyzing dynamic binary instrumentation overhead. In: WBIA workshop at ASPLOS
 19. Weaver VM (2016) Advanced hardware profiling and sampling (PEBS, IBS, etc.): creating a new PAPI sampling interface
 20. Stewart A (2011) A programming model for BSP with partitioned synchronisation. *Form Asp Comput* 23(4):421
 21. Clapp R, Dimitrov M, Kumar K, Viswanathan V, Willhalm T (2015) Quantifying the performance impact of memory latency and bandwidth for big data workloads. In: IEEE international symposium on workload characterization (IISWC), 2015. IEEE, pp 213–224
 22. Chatzopoulos G, Dragojević A, Guerraoui R (2016) Estima: extrapolating scalability of in-memory applications. In: Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, p 27
 23. Craig AP, Mickelson SA, Hunke EC et al (2015) Improved parallel performance of the CICE model in CESM1. *J High Perform Comput Appl* 29(2):154–165
 24. Prakash S, Bagrodia RL (1998) MPI-SIM: using parallel simulation to evaluate MPI programs. In: Proceedings of the 30th conference on Winter simulation. IEEE Computer Society Press, pp 467–474
 25. Zheng G, Kakulapati G, V. Kalé L (2004) Bigsim: a parallel simulator for performance prediction of extremely large parallel machines. In: Parallel and distributed processing symposium, 2004. Proceedings. 18th International. IEEE, p 78
 26. Zhai J, Chen W, Zheng W (2010) Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In: ACM sigplan notices, vol 45. ACM, pp 305–314
 27. Wu X, Mueller F (2011) Scalaextrap: trace-based communication extrapolation for spmd programs. In: ACM SIGPLAN notices, vol 46. ACM, pp 113–122
 28. Engelmann C (2014) Scaling to a million cores and beyond: using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Gener Comput Syst* 30:59
 29. Yin H, Hu Z, Zhou X, Wang H, Zheng K, Nguyen QVH, Sadiq S (2016) Discovering interpretable geo-social communities for user behavior prediction. In: IEEE international conference on data engineering, pp 942–953
 30. Yin H, Cui B, Zhou X, Wang W, Huang Z, Sadiq S (2016) Joint modeling of user check-in behaviors for real-time point-of-interest recommendation. *ACM Trans Inf Syst (TOIS)* 35(2):11
 31. Hoefler T, Gropp W, Kramer W, Snir M (2011) Performance modeling for systematic performance tuning. In: State of the practice reports, ACM, p 6
 32. Bhattacharyya A, Hoefler T (2014) Pemogen: Automatic adaptive performance modeling during program runtime. In: Proceedings of the 23rd international conference on parallel architectures and compilation. ACM, pp 393–404
 33. Wang S, Wang S, Wang S, Wang S, Wang S (2016) Learning graph-based POI embedding for location-based recommendation. In: ACM international on conference on information and knowledge management, pp 15–24
 34. Yin H, Wang W, Wang H, Chen L, Zhou X (2017) Spatial-aware hierarchical collaborative deep learning for POI recommendation. *IEEE Trans Knowl Data Eng* PP(99):1
 35. Yin H, Cui B, Sun Y, Hu Z, Chen L (2014) LCARS:aA spatial item recommender system. *Acm Trans Inf Syst* 32(3):11
 36. Martinasso M, Kwasniewski G, Alam SR, Schulthess TC, Hoefler T (2016) A PCIe congestion-aware performance model for densely populated accelerator servers. In: Proceedings of the international conference for high performance computing, networking, storage and analysis. IEEE Press, p 63
 37. Yang LT, Ma X, Mueller F (2005) Cross-platform performance prediction of parallel applications using partial execution. In: Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 conference. IEEE, pp 40–40