Available online at www.sciencedirect.com
ScienceDirect
Simulation Modelling Practice and Theory 15 (2007) 481–491
SIMULATION MODELLING PRACTICE AND THEORY
www.elsevier.com/locate/simpat

Performance modeling of communication and computation in hybrid MPI and OpenMP applications

Laksono Adhianto *, Barbara Chapman

Department of Computer Science University of Houston, TX, United States

Received 24 February 2006; received in revised form 31 July 2006; accepted 24 November 2006
Available online 20 January 2007

Abstract

Performance evaluation and modeling are crucial steps to enabling the optimization of parallel programs. Programs written using two programming models, such as MPI and OpenMP, require analysis to determine both performance efficiency and the most suitable numbers of processes and threads for their execution on a given platform. To study both of these problems, we propose the construction of a model that is based upon a small number of parameters, but is able to capture the complexity of the runtime system. We incorporate measurements of overheads introduced by each of the programming models, and thus need to model both the network and computational aspects of the system.

We have combined two different techniques that includes static analysis, driven by the OpenUH compiler, to retrieve application signatures and a parallelization overhead measurement benchmark, realized by Sphinx and Perfsuite, to collect system profiles. Finally, we propose a performance evaluation measurement to identify communication and computation efficiency. In this paper, we describe our underlying framework, the performance model, and show how our tool can be applied to a sample code.

Keywords: Performance modeling; MPI; OpenMP; Cluster; SMP

1. Introduction

Clustered symmetric multiprocessors (SMP) are the most cost-effective solution for large scale applications. Of the top 10 supercomputers listed in the Top500, most (if not all) are clusters of SMPs. A combination of MPI [11] and OpenMP [32] is regarded as a suitable programming model for such architectures. For instance, developers can employ MPI to communicate between nodes and OpenMP for parallelization within the SMP node.

Previous work has shown performance improvement using the mixed mode model [41,4,13,40,10,19]. There are also significant reports of poor hybrid performance [7,23,6,8,17,24,39] or ones that show minor benefits to adding OpenMP to an MPI program [30,3,25,13]. The factors which can affect a hybrid MPI and OpenMP

* Corresponding author.
E-mail addresses: laksono@cs.uh.edu (L. Adhianto), chapman@cs.uh.edu (B. Chapman).

1569-190X/$ - see front matter
doi:10.1016/j.simpat.2006.11.014

application's performance are numerous, complex and interrelated. We can roughly classify them into three different areas:

- Poor **MPI communication efficiency**: application related problem such as types of MPI routines (e.g., blocking and collective), message size and network related such as network contention. An MPI implementation may also impact performance due to issues such as message buffering and synchronization.
- Poor **OpenMP parallelization efficiency**: critical sections that incur long wait times, OpenMP thread management overheads, and poor cache utilization and false sharing may all reduce performance.
- Poor **MPI and OpenMP interaction**: load imbalance, idle threads during MPI communication, or frequent entering and exiting of OpenMP parallel regions in order to execute MPI constructs reduces parallel efficiency.

The first problem may be inherent to the algorithm or be the result of an inefficient network or runtime library. The second factor may be algorithm-related, the result of suboptimal programming, or be caused by compiler and its run-time system. The third problem can generally be solved by employing the so called OpenMP-SPMD programming technique, which requires extensive array privatization.

In order to obtain an efficient hybrid program, we must:

- Determine whether a program has efficiency problems and develop a strategy to overcome these where possible.
- Determine the most efficient combination of MPI processes and OpenMP threads on a given platform and problem size.

We have created performance evaluation tools to help measure the communication and computation parts of the program in order to carry out these tasks.

There are three major techniques for performance modeling and prediction: *analytical modeling*, *simulation*, and *measurement/instrumentation*. Analytical modeling has the lowest cost as it is mostly based on static analysis. However, it must make assumptions about a program's control flow and values of its data, and may not take the runtime environment properly into account. Simulation enables an automated approach to assessing program performance under a variety of conditions, but take an excessive amount of time. Dynamic measurement is probably the most accurate of the techniques. However, instrumentation overhead may significantly perturb results and huge trace/event files may be generated; a program counter-based approach does not suffer from these problems but has somewhat limited applicability.

Our goal is to model hybrid MPI and OpenMP analytically, to detect communication and parallelization inefficiency, as well as inefficiencies caused by the strategy used to combine the two programming models, and lastly to use our model and additional insights to optimize the performance of the mixed mode model. In this paper, we address the problem of detecting performance problems posed by MPI communication and OpenMP multithreaded computation. We propose a novel and low-cost approach to analyze and model hybrid MPI and OpenMP performance behavior analytically an enhanced with system profiling.

The remainder of this paper is organized as follows. In Section 2, we describe some related works on performance modeling and prediction. In Section 3 we introduce our approach to analyzing hybrid MPI and OpenMP applications and modeling the communication and multithreaded computation. In Section 6, we report the experimental result of our framework. Finally, Section 7 concludes with a summary and some interesting research directions.

2. Performance modeling

2.1. Modeling the communication

Analytical model of message-passing communication has been widely recognized since the emergence of the message-passing programming model in the early of 90s [18,9,2]. Since modeling the communication is a com-

plex problem, most message-passing models are based on basic communication patterns, such as sending–receiving or point-to-point pattern. A more complex communication pattern such as collective communication is constructed based on basic patterns.

There are several performance models for distributed memory such as Log *P* [9], LogGP [2] and PLogP [20]. LogP [9] predicts the communication performance by assuming that only constant-size small messages are communicated between the nodes. LogGP [2] is an extension of the LogP model that additionally allows for large messages by introducing the gap per byte parameter. PLogP [20] is another extension of the LogP model that includes major contributing factors such as copying data to and from network interfaces.

Communication latency can be modeled more accurately using an MPI benchmark. Many benchmarks such as MPPTest [15], MPBench [28], Pallas MPI benchmark [14], MPIBench, SKaMPI [34] and Sphinx [38] (which is a branch of SKaMPI) have been created to obtain more realistic figures for a given target system (machine, MPI implementation, compiler and operating system). MPPTest is an MPI performance measurement benchmark developed by Argonne National Laboratory (ANL) and distributed within the MPICH library. It supports basic MPI routines exposing contention and scalability problems. MPBench evaluates MPI benchmarks on two different CPUs, developed by Philip Mucci from University of Tennessee. It is easy to use and supports critical MPI routines.

Pallas MPI benchmark (PMB) is a more complete benchmark developed by Pallas Gmbh, and is now part of Intel and renamed as the Intel cluster toolkit for Linux. The benchmark is able to not only measure latency, but also bandwidth and collective performance. MPIBench is a complete performance model benchmark for MPI developed by the University of Adelaide in Australia. It uses a highly accurate global clock for measurement and times individual routine calls, for all processes. It is also one of few MPI benchmarks that are able to take into consideration network contention. SKaMPI is a configurable and easy to extend message-passing performance model from the University of Karlrushe. Similar to most performance modeling benchmarks, it does not take into consideration network contention when measuring point to point communication, but instead it takes into account the slowest communication of two processes. Sphinx is a derivative of SKaMPI developed by the Lawrence Livermore National Laboratory (LLNL). It inherits SKaMPI's flexibility and extensibility with additional support for OpenMP and Pthreads. To the best of our knowledge, Sphinx is the only benchmark that is capable of measuring not only message passing communication overhead, but also multi-threading overhead.

2.2. Modeling the computation

Some models have been proposed for shared memory [12,16]. The Queuing Shared Memory (QSM) model [12] takes into consideration the number of memory accesses and memory contention, but does not differentiate between contiguous versus non-contiguous accesses. On the other hand, Helman and JaJa [16] takes into account contention of both the processors and memory.

Marin and Mellor-Crummey [26] suggests a semi-automatic approach to modeling and predicting the characteristics of program behavior. They use a combination of the attributes of applications and the result of simple probes. Although the model shows high accuracy for sequential programs, it is unknown if it will extend to model parallel programs. Other work predicting computation performance is carried out by [36,29]. Shen et al. [36] estimate the locality phases of a program via a combination of locality profiling and run-time prediction, while [29] uses dynamic sampling of trace snippets throughout an application's execution to model performance behavior.

Similar to modeling for the overhead of message-passing communication, multi-threading overhead is also a non-deterministic model and hard (if not impossible) to predict accurately due to dependence of how it is implemented, the underlying architecture and the platform. To the best of our knowledge, there are only two OpenMP overhead benchmarks available: Sphinx [38] and EPCC [5]. Sphinx is based on the SKaMPI infrastructure and enhanced for OpenMP and Pthreads support by Bronis de Supinski from LLNL. EPCC is developed by Mark Bull from the University of Edinburgh, UK. EPCC is a very simple, easy to understand and robust. Both benchmarks employ the same basic approach: they both compare the difference in execution time from the measured parallel routine into a reference routine.

2.3. Modeling the communication and the computation

Modeling mixed communication and computation has been around since the beginning 1990s. MPI application performance is modeled and predicted using static analysis by [22]. The SUIF compiler is employed to parse the source code and retrieve information on MPI communication calls and arithmetical operations from the intermediate representation. The message-passing communication is modeled with a specific formula for each pattern (collective, point-to-point, ...) while computation is based on floating point arithmetic operation. Although the model is simple, it is surprisingly very accurate for real applications.

The PERC project [37] uses the so-called convolution technique to combine *machine profiles* and *application signatures* for both serial and parallel programs. An *application signature* is a summary of the fundamental operations to be carried out by the application, independent of any particular machine. A *machine profile* is a representation of the capability of a resource/system to perform operations. An *application signature* is collected by a dynamic instruction trace and then "mapped" with a *machine profile* by using the "convolution" method.

The POEMS project [1] developed an accurate performance model for parallel applications executing on dedicated, shared memory systems. The model has two levels: a lower-level queuing model to characterize the impact of contention and caching effects, and a higher-level task graph model of the application.

3. Methodology

The main idea in our approach is to exploit and appropriately adapt the ideas of the PERC project to evaluate specifically the performance of an MPI and/or OpenMP program. We follow their definition of *application signature* where a program can be represented by memory access patterns and arithmetic operations. However, we retrieve application information such as memory access pattern and floating-point and integer operation from the compiler, rather than trace program behavior at run time. The advantage of our approach is that instead of using a tracer/profiler that needs program execution, we rely on analytical models of the source code. Since our target application are large scale applications, which have long execution time, it is preferable to detect performance efficiency as early as possible.

Our *system profile* is similar to the *machine profile* defined in PERC, but we extend their definition to include characteristics of the network and the compiler's runtime library. By combining an *application signature* and *system profile*, we can model the behavior of an application on a given target system for different problem sizes without requiring program execution. In this Section, we describe how we analytically model parallel performance measurement, collect system profile and combine the two approaches.

3.1. Performance measurement

The parameters we use to measure performance behavior are based on the work of Chow et al. [8] who suggests that message-passing efficiency can be measured based on the ratio of the communication time of a hybrid application and the communication time of the pure MPI version. Assume $t_{\text{hyb}}^{\text{comm}}$ is the estimated communication time of the hybrid application, $t_{\text{mpi}}^{\text{comm}}$ is the communication time if the application is executed without multithreading (pure MPI), p is the number of MPI processes, m_{t} is the number of threads used in communication, and n_{t} is the total number of OpenMP threads (where $n_{\text{t}} \geqslant m_{\text{t}}$), then the message passing efficiency e_{mp} can be defined as:

$$e_{\text{mp}}(n_{\text{t}}p) = \frac{\sum t_{\text{hyb}}^{\text{comm}}(m_{\text{t}}p)}{\sum t_{\text{mpi}}^{\text{comm}}(n_{\text{t}}p)} \tag{1}$$

The parameter $(n_{\text{t}}p)$ in e_{mp} is used as a function of the total number of CPUs used by the program. Therefore, if a hybrid application employs $p \times n_{\text{t}}$ processors although only $p \times m_{\text{t}}$ are used to perform communication, we need to compare it with a pure MPI program running with $p \times n_{\text{t}}$ processes. Our formula is more accurate than [8] if a hybrid application uses more than one thread to perform communication.

Modeling multithreading efficiency can be very tricky and is more difficult to determine than message-passing efficiency. First, parallelization overhead varies significantly based upon the compiler, machine architecture, operating system and runtime library. We can deal with this by employing microbenchmarks to measure the overheads exclusively. Some benchmarks, such as Sphinx [38] and EPCC [5] are developed specifically for this purpose.

Second, compared to a message-passing based program, cache memory has a bigger impact on multithreading applications. It is known that a cache-optimized multithreaded program can achieve significant speedup compared to an unoptimized program [27]. Most analytic performance models for parallel programs have only limited ability to consider cache behavior such as *cache misses*, while simulation is a more promising technique for capturing *false sharing*.

Our model for a serial computation loop is based on [43], where the estimated execution time of a loop $t_{\text{serial}}^{\text{comp}}$ is defined as the total of the sum of predicted cache misses t_{cache}, loop overhead t_{overhead} and arithmetic processing t_{machine} including pipelining, register pressure and latency.

$$t_{\text{serial}}^{\text{comp}} = t_{\text{machine}} + t_{\text{overhead}} + t_{\text{cache}} \tag{2}$$

where t_{machine} is the total cycles of arithmetic computation, jump branch (such as branch condition and call function) and the estimated time of the calls if interprocedural analysis is activated (otherwise we use the default value). In the implementation, t_{machine} is limited for the analyzable source code. While calls to external libraries are simply set by default which may decrease the accuracy of our prediction.

Without loss of generality, we simplify our model of estimated execution time of a parallel loop $t_{\text{par}}^{\text{comp}}$ to:

$$t_{\text{par}}^{\text{comp}} = \frac{t_{\text{serial}}^{\text{comp}}}{n_{\text{t}}} + \sum t_{\text{unpar}}^{\text{comp}} + \sum O \tag{3}$$

where $\sum O$ is the total multithreading overhead and $t_{\text{unpar}}^{\text{comp}}$ is the portion of the code that cannot be parallelized. Many OpenMP codes contain worksharing directives such as `OMPDO` in Fortran (or pragma `ompfor` in C) to mark parallel loops, critical regions using the `CRITICAL` directive or locks, and explicit barrier synchronizations. In this case, $\sum O$ is the sum of all overheads and synchronization costs incurred by OpenMP directives. Ideally, $t_{\text{par}}^{\text{comp}}$ would also have an additional cache penalty to take false sharing into account. Determining a suitable penalty is not trivial, since it needs to be machine-dependent. Within a cc-NUMA architecture, this penalty may depend on the distance between two nodes. Including false sharing in our model will be a major focus of our future work.

The multithreading efficiency measurement is simply the ratio of the execution time for the sequential program $\sum t_{\text{serial}}^{\text{comp}}$ and its parallel version:

$$e_{mt} = \frac{\sum t_{\text{serial}}^{\text{comp}}}{n_{\text{t}} \sum t_{\text{par}}^{\text{comp}}} \tag{4}$$

$$= \frac{\sum t_{\text{serial}}^{\text{comp}}}{n_{\text{t}} \left(\frac{t_{\text{serial}}^{\text{comp}}}{n_{\text{t}}} + \sum t_{\text{unpar}}^{\text{comp}} + \sum O \right)} \tag{5}$$

The value of e_{mt} will lie between zero and one, and could only be equal to one if there is no serial code and no parallel overheads are incurred. The definition does not take the cumulative effect of multiple caches into account and will thus not be able to predict superlinear speedup, which does occur in practice.

3.2. Modeling framework

As shown in Fig. 1, our framework tool comprises three main parts:

- Eclipse[42] for the primary user interface.
- Benchmarks to construct the system profile: Sphinx [38] to retrieve parallel overheads (MPI, OpenMP and hybrid MPI + OpenMP) and Perfsuite [21] to collect hardware information.
- The OpenUH compiler [33] to analyze source code and determine application signatures.

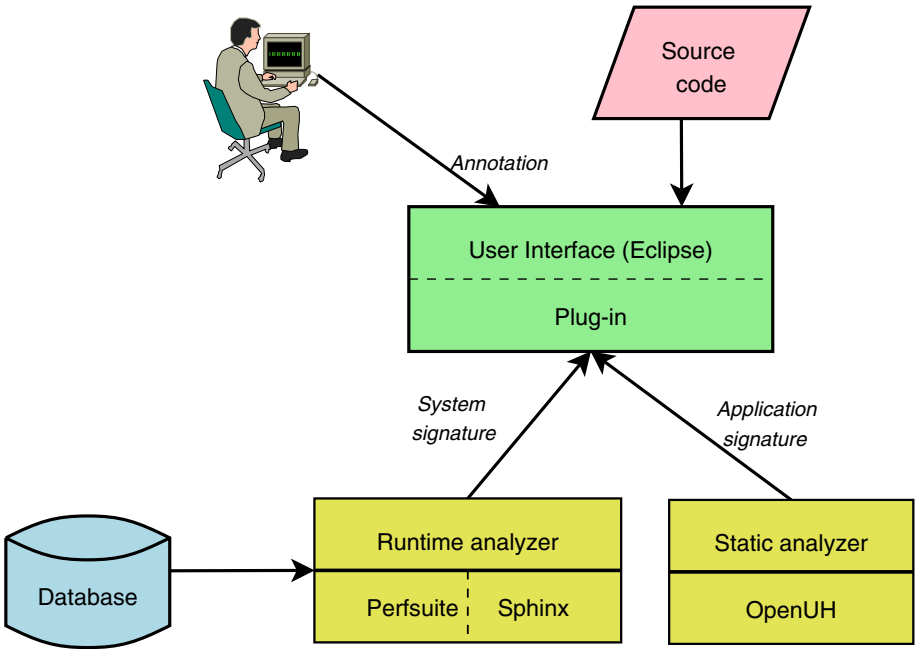

Fig. 1. Tool framework.

All components are based on open source tools. Eclipse [42] is an extensible open source IDE that can be used to create applications as diverse as web sites, embedded Java programs, C++ programs, and Enterprise JavaBeans. We are currently developing a new plug-in to deal with user interaction. Our framework needs user input to provide the number of MPI processes, OpenMP threads and loop iterations.

As shown in Eq. (1), our performance measurement requires that we obtain the value of t_{comm}, the latency of message-passing communication. Existing models typically assess parameters such as message size, latency, transfer time per byte, gap per message and number of nodes. However, an approach that does not take the MPI implementation into account is not sufficiently accurate for our purposes so that here we intend to use measurements instead. We decided to use Sphinx because of its flexibility, ease of configuration and also its support for OpenMP and hybrid MPI + OpenMP. Sphinx (and SKaMPI) is also very accurate: it not only repeats the test based on the distribution of the results, but also supports high timing resolution. A disadvantage was that the OpenMP overhead measurements were not complete and not fully tested. We have accordingly extended Sphinx to measure the overheads of most OpenMP constructs including synchronization (master, ordered, set/unset lock) and variable scoping (such as private, lastprivate and threadprivate).

Perfsuite [21] is open source software that contains a small set of tools, utilities, and libraries for user-level application performance analysis on x86 and ia64 Linux systems. Perfsuite provides access to accurate, high-resolution timers, information about architectural features such as details of the memory hierarchy, and resource usage information such as CPU time consumed or the resident set size of a running application. One of the Perfsuite tools we are interested in is `psinv`. This tool retrieves hardware information such as clock speed, memory size, cache size and cache line size. In conjunction with array usage analysis from the compiler, this information is useful to predict cache reuse, loop cost and false sharing in an application [27].

Our tool operates in the following manner. First, the OpenUH compiler parses and analyzes a hybrid MPI and/or OpenMP program. The compiler then extracts for each computational loop: the estimated computation time $t_{\text{serial}}^{\text{comp}}$ and list of OpenMP directives. The compiler also outputs MPI routines including the type of communication and its message length in bytes. Since in most cases the problem size is undefined during compilation, we need to manually define the value, then pass it to Sphinx to measure the estimated communication time $t_{\text{mpi}}^{\text{comm}}$ and $t_{\text{hyb}}^{\text{comm}}$ for the target runtime library and machine. The computation of e_{mt} and e_{mp} is then carried out in our Eclipse plugin.

4. OpenUH compiler

OpenUH [33] is an adoption of the open source Open64 compiler infrastructure [31]. The Open64 Project is focused on compilers for the C/C++ /Fortran95 languages based on technology originally developed by SGI and released under open source GPL license in 1999. OpenUH extends Open64 features such as portability, OpenMP support, autoscoping, OpenMP for cluster extension and enhanced Fortran front-end. It supports rich and powerful analysis such as dependence analysis, alias analysis, cache modeling, machine modeling and interprocedural analysis.

As shown in Fig. 2, the compiler consists of mainly six components: Fortran and C/C++ front-end (FE), interprocedural analysis (IPA), loop nest optimization (LNO), global optimizer (WOPT), code generation (CG) and lastly an assembler (AS). The FE module is divided into to sub modules: one for parsing C/C++ code based on the GNU GCC front-end, and the other parses Fortran 90 code based on the Cray F90 front-end. The output of the front-end is an intermediate representation (IR), which in Open64 is called WHIRL for Winning Hierarchical Intermediate Representation Language. WHIRL consists of five level hierarchies:

(1) Very high level (VHL) serves as the interface between the front-end and the middle-end,
(2) High level (HL) is used as the common interface among middle-end components such as LNO and IPA,
(3) Mid level (ML) is mainly the representation for WOPT components,
(4) Low level (LL) is also used by the global optimizer, and
(5) Very low level (VLL) for the code generation.

The *application signature* module is implemented in the LNO component. There are some reasons for this choice: first, LNO is carried out after IPA, which can take advantage of higher analysis accuracy due to interprocedural analysis (IPA). Second, LNO uses HL WHIRL which contains rich information about the loop, array index, loop index and OpenMP directives (or pragmas in C/C++). Lastly, loop based analysis such as data dependence analysis and array region (to some extent) is performed in this level, which enables us to easily manipulate and analyze at the loop level. The only disadvantage of using this approach is that the analysis is performed after some general optimization such as constant propagation and code hoisting. Fortunately, there does not significantly harm the accuracy of the model.

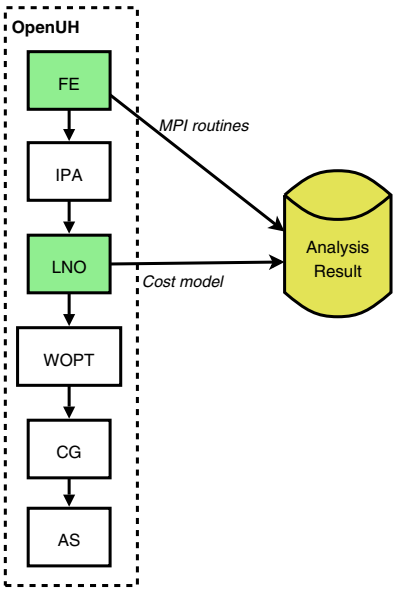

Fig. 2. OpenUH infrastructure.

```
/* computation part */
void MatrixMultiply() {
   int i, j, k;          /* loop indices */
   for(i=0; i<bs; i++)  /* bs is the block size */
      for(j=0; j<bs; j++)
         for(k=0; k<bs; k++)
            c[i,j] += a[i,k]*b[k,j];
}
/* communication part */
void RotateArrays() {
      /* shift "a" to the left */
      MPI_Send(a, bs*bs, MPI_FLOAT, west, ...);
      MPI_Recv(a, bs*bs, MPI_FLOAT, east, ...);
      /* shift "b" upwards */
      MPI_Send(b, bs*bs, MPI_FLOAT, north, ...);
      MPI_Recv(b, bs*bs, MPI_FLOAT, south, ...);
}
/* main algorithm*/
    . . . .
   /* main loop of Cannon's algorithm */
   for(l=0; l<nb_neigbours; l++) {
      MatrixMultiply();
      RotateArrays();
   }
```

Fig. 3. Main algorithm of cannon matrix multiply.

5. Sphinx

We have seen in Section 4 that modeling realistic analytical communication or multi-threading overhead is almost impossible without the help of a benchmark. Our research goal not only focuses on message passing communication, but also OpenMP parallelization. And one of the most suitable benchmark that is capable of catering to these two issues is Sphinx. In most analytical communication models there are several parameters to consider such as bandwidth, latency, gap per message and the message length. By using Sphinx, we consider only two parameters: communication type and the message length (in bytes). Reduced parameters help us to simplify our framework and allow us to model the communication relatively easier.

The communication latency is calculated as follows: first, Sphinx reads a parameter file passed by the OpenUH compiler to determine which communication type to measure. Sphinx then computes the full message round-trip of the MPI communication. In timing the communication, Sphinx also considers *statistical error* and *semantic error* [35]. In order to control *statistical error*, timing of the communication is repeated until it meets the lower bound of the defined standard deviation. If the network is perturbed where the standard deviation is always bigger than the tolerated error, the repetition is stopped until it reaches the maximum number of iterations.

In order to reduce *systematic error*, Sphinx will first perform dummy communication to refresh the cache memory before it performs the real test. A *systematic error* also occurs when the overhead of the timer subroutine is significant. To eliminate this error, Sphinx utilizes `MPI_Wtime` where the overhead is usually negligible.

6. Case study

Due to the space constraint, we consider one simple case study, a parallel matrix multiply $C = A \times B$ using Cannon algorithm. The code is an interesting problem for two reasons. First, the communication is performed outside the computation, which simplifies the explanation of our methodology. Second, matrix multiplication is not easily identified as parallelizable due to a data dependency in the innermost loop.

As shown in Fig. 3, the main loop of the program consists of two functions: local matrix multiplication and message-passing communication to rotate the matrix A and B. The matrix rotation is based on two point-to-point message-passing communication patterns of `MPI_Send` and `MPI_Recv`, while the matrix multiply computation is parallelized with an OpenMP `parallel for` directive.

Table 1
The estimated execution time of the loop in matrix-multiply function

Cost factor	Execution time	
	(cycles)	(s)
Machine cycles	130000.00	0.050544
Loop overhead cycles	525252.00	0.204217978
Cache cycles	85799.43	0.033358818
Total	741051.43	0.289514255

Table 2
Estimated communication efficiency e_{mp} vs. real e_{mp}

CPUs	Estimated e_{mp}	Real e_{mp}	Error (in %)
4	1	1	0
9	1.832	2.731	32.916
16	3.139	2.758	12.132

Table 3
Estimated multithreading efficiency e_{mt} vs. real e_{mt}

n_t	Overhead O	Estimated e_{mt}	Real e_{mt}	Error (in %)
1	0.013	0.956	1	4.402
2	0.144	0.499	0.500	0.184
3	0.179	0.349	0.334	4.609
4	0.203	0.262	0.249	5.214

We conduct our experiment on an HP RX8620 with 16 itanium2 1.5 GHz processors with 6 MB cache and 32×1 GB DIMMs of RAM, running on Linux kernel 2.6 with Intel compiler version 9.0 and MPI library from ScaMPI v 1.5.

The result of predicted communication efficiency can be seen in Table 2. As we can see, our estimated e_{mp} is not very accurate where the error ranges between 12% and 32%. This inaccuracy is understandable because measured communication times vary widely. Sphinx report that the standard deviation of the real communication is up to 151%. Thus we believe that a large error margin is tolerable.

Our current multithreaded computation model can only be accurate if false sharing is not significant. In our matrix multiply case, we expect few occurrences of false sharing. Our compiler first estimates the serial execution time of the multiplication loop as shown in Table 1. Then we model the parallel version according to Eq. (3) where the overhead O is measured by Sphinx. Next, we measure the predicted multithreading efficiency e_{mt} and compare with the real e_{mt}. As expected, the error is relatively small (up to 5.2%), as shown in Table 3.

We can learn from our performance model that for the given problem size, it is not beneficial to increase the number of threads due to significant OpenMP overhead introduced by the Intel OpenMP compiler. On the other hand, communication efficiency can be increased by adding the number of MPI processes. However, since the proportion of communication time is much smaller than the computation time, there is no significant benefit in increasing the number of MPI processes.

7. Conclusion

We have proposed a novel and cost efficient approach to model and evaluate parallel OpenMP, MPI and hybrid MPI + OpenMP with reasonable accuracy. Our approach is based on a combination of static analysis and feedback from a runtime benchmark for both communication and multithreading efficiency measurement. The static analysis performed by the OpenUH compiler serves to retrieve *application signature* such as computation loops, cache access pattern, MPI routines and OpenMP directives. Runtime benchmarks from Sphinx and Perfsuite are helpful to collect *system profile* such as communication latency, overhead and machine information.

Our approach has the advantage of being fast, more accurate than the general analytical model and does not require program execution. Moreover, we also allow user interaction to define unknown variables such as the number of MPI processes and OpenMP threads. This feature enables greater flexibility for users to model application behavior for different problem sizes, different numbers of threads and processes and different target machines without running the application. Lastly, another advantage compared to other methodologies is that we can reuse the same *application signature* to predict the performance on another machine with a different problem size. The same holds for the *machine profile*, which we can reuse to measure the performance of other applications.

The results of this performance evaluation and modeling is used for program analysis and optimization. For instance, in the example of matrix multiplication, our tool is able to identify that the communication efficiency can be increased by using another MPI communication routine such as `MPI_Sendrecv` instead of using the combination of `MPI_Send` and `MPI_Recv`.

References

[1] Vikram S. Adve, Mary K. Vernon, Parallel program performance prediction using deterministic task graph analysis, ACM Transactions On Computer Systems 22 (1) (2004) 94–136.
[2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, Chris Scheiman, Loggp: incorporating long messages into the logp model: one step closer towards a realistic model for parallel computation, in: SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures, ACM Press, New York, NY, USA, 1995, pp. 95–105.
[3] Siegfried Benkner, Viera Sipkov'a, Exploiting distributed-memory and shared-memory parallelism on clusters of smps with data parallel programs, International Journal of Parallel Programming 31 (1) (2003) 3–19.
[4] Steve W. Bova, Clay P. Breshears, Henry Gabb, Bob Kuhn, Bill Magro, Rudolf Eigenmann, Greg Gaertner, Stefano Salvini, Howard Scott, Parallel programming with message passing and directives, Computing in Science and Engineering 3 (5) (2001) 22–37.
[5] J.M. Bull, Measuring synchronisation and scheduling overheads in openmp, in: In European Workshop on OpenMP (EWOMP1999), Lund, Sweden, 1999.
[6] I.J. Bush, C.J. Noble, R.J. Allan, Mixed openmp and mpi for parallel fortran applications, in: In European Workshop on OpenMP (EWOMP2000), Edinburgh, UK, 2000.
[7] F. Cappello, D. Etiemble, Mpi versus mpi + openmp on ibm sp for the nas benchmarks, in: In SC2000, Supercomputing 2000, November, Dallas, 2000.
[8] Edmond Chow, David Hysom. Assessing performance of hybrid mpi/openmp programs on smp clusters. Technical Report UCRL-JC-143957, Lawrence Livermore National Laboratory, May 2001.
[9] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, Thorsten von Eicken, Logp: towards a realistic model of parallel computation, in: PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM Press, New York, NY, USA, 1993, pp. 1–12.
[10] N. Drosinos, N. Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium 2004 (IPDPS 2004), Santa Fe, New Mexico, April 2004, p. 15.
[11] Message Passing Interface Forum. <http://www.mpi-forum.org>.
[12] Phillip B. Gibbons, Yossi Matias, Vijaya Ramachandran, Can sharedmemory model serve as a bridging model for parallel computation?, in: SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures, ACM Press, New York, NY, USA, 1997, pp 72–83.
[13] L. Giraud, Combining shared and distributed memory programming models on clusters of symmetric multiprocessors: some basic promising experiments. Working Note WN/PA/01/19, CERFACS, Toulouse, France, 2001.
[14] Pallas GmbH. Pallas mpi benchmarks - pmb. <http://www.pallas.de/pages/pmbd.htm>.
[15] William D. Gropp, Ewing Lusk, Reproducible measurements of MPI performance characteristics, in: Jack Dongarra, Emilio Luque, Tom'as Margalef (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, vol. 1697, Springer Verlag, 1999, pp. 11–18, 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 1999.
[16] David R. Helman, Joseph Jaacute, Prefix computations on symmetric multiprocessors, Journal of Parallel and Distributed Computing 61 (2) (2001) 265–278.
[17] D.S. Henty, Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling, in: Supercomputing 2000, 2000.
[18] Roger W. Hockney, The communication challenge for mpp: Intel paragon and meiko cs-2, Parallel Computation 20 (3) (1994) 389–398.
[19] M.D. Jones, R. Yao, Parallel osem reconstruction speed with mpi, openmp, and hybrid mpi-openmp programming models, in: In IEEE Nuclear Science Symposium and Medical Imaging Conference Record, Rome, Italy, October 2004.
[20] Thilo Kielmann, Henri E. Bal, Kees Verstoep, Fast measurement of logp parameters for message passing platforms, in: IPDPS Workshops, 2000, pp. 1176–1183.

[21] Rick Kufrin, Perfsuite: an accessible, open source, performance analysis environment for linux, in: 6th International Conference on Linux Clusters (LCI- 2005), Chapel Hill, NC, April 2005.
[22] M. Kühnemann, T. Rauber, G. Rünger, A source code analyzer for performance prediction, in: Proceedings of the IPDPS-Workshop on Massively Parallel Processing (CDROM), IEEE, 2004.
[23] Piero Lanucara, Sergio Rovida, Conjugate-gradients algorithms: an mpiopenmp implementation on, in: First European Workshop on OpenMP, 1999, pp. 76–78.
[24] G. Mahinthakumar, F. Saied, A hybrid MPI-OpenMP implementation of an implicit finite-element code on parallel architectures, International Journal of High Performance Computing Applications 16 (4) (2002) 371–393.
[25] Amitava Majumdar, Parallel performance study of monte carlo photon transport code on shared-, distributed-, and distributed-shared-memory architectures, in: IPDPS, 2000, p. 93.
[26] Gabriel Marin, John Mellor-Crummey, Cross-architecture performance predictions for scientific applications using parameterized models, in: SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems, ACM Press, New York, NY, USA, 2004, pp. 2–13.
[27] Kathryn S. McKinley, A compiler optimization algorithm for shared-memory multiprocessors, IEEE Transactions on Parallel and Distributed Systems 9 (8) (1998) 769–787.
[28] P. Mucci, K. London, The Mpbench Report, 1998.
[29] Jeffrey Odom, Jeffrey K. Hollingsworth, Luiz DeRose, Kattamuri Ekanadham, Simone Sbaraglia, Using dynamic tracing sampling to measure long running programs, in: SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2005, p. 59.
[30] Leonid Oliker, Xiaoye Li, Parry Husbands, Rupak Biswas, Effects of ordering strategies and programming paradigms on sparse matrix computations, SIAM Rev. 44 (3) (2002) 373–393.
[31] Open64. <http://sourceforge.net/projects/open64/>.
[32] OpenMP. <http://www.openmp.org>.
[33] OpenUH. <http://www.cs.uh.edu/~openuh>.
[34] Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Muller. Skampi: A detailed, accurate MPI benchmark. In PVM/MPI, 1998, pp. 52–59.
[35] Ralf Reussner, Peter Sanders, Jesper Larsson Träff, Skampi: a comprehensive benchmark for public benchmarking of mpi, Scientific Programming 10 (1) (2002) 55–65.
[36] Xipeng Shen, Yutao Zhong, Chen Ding, Locality phase prediction, in: ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, ACM Press, New York, NY, USA, 2004, pp. 165–176.
[37] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, Avi Purkayastha, A framework for performance modeling and prediction, in: Supercomputing'02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, IEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 1–17.
[38] SPHINX. <http://www.llnl.gov/casc/sphinx/sphinx.html>.
[39] Chin Hoe Tai1, Yong Zhao, K.M. Liew, Parallel-multigrid computation of unsteady incompressible viscous flows using a matrix-free implicit method and high-resolution characteristics-based scheme, Computer Methods in Applied Mechanics and Engineering 194 (36–38) (2005) 3949–3983.
[40] M.B. van Gijzen. Two level parallelism in a stream-function model for global ocean circulation. Technical Report TR/PA/03/09, CERFACS, Toulouse, France, 2003.
[41] Huang W. and Tafti D.K.A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications. In Parallel CFD99, Wiiliamsburg, VA, May 1999.
[42] Andre Weinand, Eclipse – an open source platform for the next generation of development tools, in: NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, Springer-Verlag, London, UK, 2003, p. 3.
[43] Michael E. Wolf, Dror E. Maydan, Ding-Kai Chen, Combining loop transformations considering caches and scheduling, in: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, IEEE Computer Society, 1996, pp. 274–286.