Performance Modeling of Communication and Computation in Hybrid MPI and OpenMP Applications

Laksono Adhianto and Barbara Chapman
Department of Computer Science
University of Houston, TX
{laksono,chapman}@cs.uh.edu

Abstract

Performance evaluation and modeling is a crucial process to enable the optimization of parallel programs. Programs written using two programming models, such as MPI and OpenMP, require an 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 must 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: 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.

1. Introduction

Clustered symmetric multiprocessors (SMP) are the most cost-effective solution for large scale applications. Of the top ten supercomputers listed in the Top500, most (if not all) are clusters of SMPs. A combination of MPI [11] and OpenMP [28] 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.

Some work has shown performance improvement using the mixed mode model [36, 4, 13, 35, 10, 17]. Interestingly, there are also significant reports of poor hybrid performance [7, 21, 6, 8, 16, 22, 34] or ones that show minor benefits to adding OpenMP to an MPI program [27, 3, 23, 13]. The factors which can affect a hybrid MPI and OpenMP 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 (blocking, collective), message size and network related such as network contention. An MPI implementation may also have impact to the performance 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 in 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 runtime 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.

Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS'06)
0-7695-2612-8/06 $20.00 © 2006 IEEE

We have created performance evaluation tools to help us measure the communication and the 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 research 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, enhanced with system profiling.

The remainder of this paper is organized as follows. In the next section, we describe some related works on performance modeling and prediction related to our work. In section 3 we introduce our approach to analyzing hybrid MPI and OpenMP applications and modeling the communication and multithreaded computation. Then, in section 4, we report the experiment of our framework. Finally section 5 concludes the paper with a summary and some interesting research directions.

2. Related Work

There are many performance models for distributed memory such as LogP [9], LogGP [2] and PLogP [18]. LogP [9] predicts the communication performance by assuming 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 [18] is another extension of the LogP model by including major contributing factors such as copying data to and from network interfaces.

Some models have been proposed for shared memory [12, 15]. The Queuing Shared Memory (QSM) model [12] takes into consideration the number of memory accesses and contention at the memory, but does not differentiate between between contiguous versus non-contiguous accesses. On the other hand, Helman and JaJa [15] takes into account contention at both the processors and the memory.

MPI application performance is modeled and predicted using static analysis by [20]. 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 PERC project [32] 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. Marin et al [24] suggest a semi-automatic approach to model and predict 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 on predicting computation performance is carried out by [31, 26]. Shen et al. [31] estimate the locality phases of a program via a combination of locality profiling and run-time prediction, while [26] uses dynamic sampling of trace snippets throughout an application's execution to model performance behavior.

3. Methodology

The main idea of our approach is to exploit and appropriately adapt the ideas of the PERC project to evaluate the performance of an MPI and/or OpenMP program. We follow their definition of *application signature*. However, we retrieve application information such as the memory access pattern and floating point operation from the compiler, rather than tracing program behavior at run time. 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

model analytically parallel performance measurement, how we collect system profile and how we 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] where it is suggested 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_{hyb}^{comm} is the estimated communication time of hybrid application, t_{mpi}^{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 (obviously, $n_t \geq m_t$), then the message passing efficiency e_{mp} can be defined as:

$$e_{mp}(n_t p) = \frac{\sum t_{hyb}^{comm}(m_t p)}{\sum t_{mpi}^{comm}(n_t p)} \quad (1)$$

The parameter $(n_t p)$ in e_{mp} is used as a function of the total number of CPUs used by the program. Therefore if a hybrid application employs $p \times n_t$ processors although only $p \times m_t$ are used to perform communication, we need to compare it with a pure MPI program running with $p \times n_t$ processes. Our formula is more accurate than [8] if a hybrid application uses more than 1 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 (at least) 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 [33] 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 [25]. 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 [38], where the estimated execution time of a loop t_{serial}^{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.

$$t_{serial}^{comp} = t_{machine} + t_{overhead} + t_{cache} \quad (2)$$

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

$$t_{par}^{comp} = \frac{t_{serial}^{comp}}{n_t} + \sum t_{unpar}^{comp} + \sum O \quad (3)$$

Where $\sum O$ is the total multithreading overhead and t_{unpar}^{comp} is the portion of the code that cannot be parallelized. Many OpenMP codes contain not only worksharing directives such as `OMP DO` in Fortran (or pragma `omp for` in C) to mark parallel loops, but also critical regions using `CRITICAL` directive or locks, or even explicit barrier synchronizations. In this case, $\sum O$ is the sum of all overheads and synchronization costs incurred by OpenMP directives. Ideally, t_{par}^{comp} would also have an additional cache penalty to take false sharing into account. Determining a suitable penalty is not trivial, however, 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 $\sum t_{serial}^{comp}$ and its parallel version:

$$e_{mt} = \frac{\sum t_{serial}^{comp}}{n_t \sum t_{par}^{comp}} \quad (4)$$

$$= \frac{\sum t_{serial}^{comp}}{n_t \left(\frac{t_{serial}^{comp}}{n_t} + \sum t_{unpar}^{comp} + \sum O \right)} \quad (5)$$

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. Performance Modeling

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

- Eclipse[37] for the main user interface.
- Benchmarks to construct the system profile: Sphinx [33] to retrieve parallel overheads (MPI, OpenMP and hybrid MPI+OpenMP) and Perfsuite [19] to collect hardware information.
- The OpenUH compiler [29] to analyze source code and determine application signatures.

All parts are based on open source tools. Eclipse [37] is an extensible open source IDE that can be used to create applications as diverse as web sites, embedded Java programs,

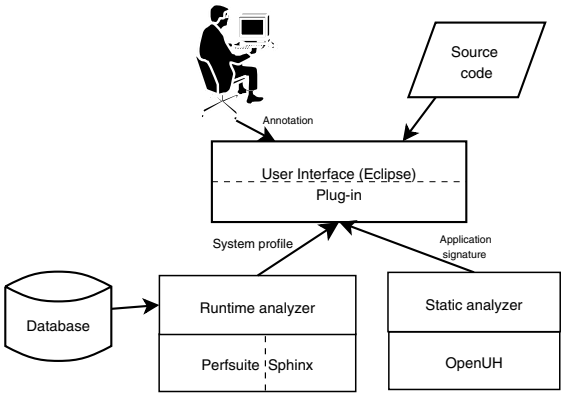

Figure 1. Tool framework

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 equation 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. Many benchmarks (e.g. SKaMPI [30], Sphinx [33] (ar branch of SKaMPI) and Pallas MPI benchmark [14] have been created to obtain more realistic figures for a given target system (machine, MPI implementation, compiler and operating system). We decided to use Sphinx due to 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 [19] is open source software and 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 [25].

Cost factor	Execution time	
	(cycles)	(seconds)
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 1. The estimated execution time of the loop in matrix-multiply function.

Our tool works as follows. First, the OpenUH compiler parses and analyzes a hybrid MPI and OpenMP program. The compiler then extracts for each computational loop: the estimated computation time t_{serial}^{comp} and list of OpenMP directives. The compiler also outputs MPI routines including the type of variable and its message length. 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_{mpi}^{comm} and t_{hyb}^{comm} for target the runtime library and machine. The computation of e_{mt} and e_{mp} is then carried out in our Eclipse plugin.

4. Case study

Due to the space constraint, we consider one simple case study only, 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.

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.

We conduct our experiment on an HP RX8620 with 16 itanium2 1.5GHz processors with 6MB cache and 32x1GB 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 to 32%.

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

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

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

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 Equation 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}. Not surprisingly, the error is relatively small (up to 5.2%), as shown in Table 3.

What we can learn from our performance model is 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.

5. Discussion

In this paper, 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 OpenUH compiler serves to retrieve *application signature* such as computation loops, cache access pattern, MPI routines and OpenMP directives; while runtime benchmarks from Sphinx and Perfsuite are helpful to collect *system profile* such as communication latency, overhead and machine information. This approach has the advantage of being fast, more accurate than the general analytical model and it 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. Last but not least, 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*, when we can reuse to measure the performance of other applications.

The result 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] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, 22(1):94–136, 2004.

[2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. 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*, pages 95–105, New York, NY, USA, 1995. ACM Press.

[3] S. Benkner and V. Sipková. Exploiting distributed-memory and shared-memory parallelism on clusters of smps with data parallel programs. *International Journal of Parallel Programming*, 31(1):3–19, 2003.

[4] S. W. Bova, C. P. Breshears, H. Gabb, B. Kuhn, B. Magro, R. Eigenmann, G. Gaertner, S. Salvini, and H. Scott. Parallel programming with message passing and directives. *Computing in Science and Engineering*, 3(5):22–37, /2001.

[5] J. Bull. Measuring synchronisation and scheduling overheads in openmp. In *European Workshop on OpenMP (EWOMP1999)*, Lund, Sweden, 1999.

[6] I. J. Bush, C. J. Noble, and R. J. Allan. Mixed openmp and mpi for parallel fortran applications. In *European Workshop on OpenMP (EWOMP2000)*, Edinburgh, UK, 2000.

[7] F. Cappello and D. Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *SC2000, Supercomputing 2000, November, Dallas*, 2000.

[8] E. Chow and D. Hysom. Assessing performance of hybrid mpi/openmp programs on smp clusters. Technical Report UCRL-JC-143957, Lawrence Livermore National Laboratory, May 2001.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. 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*, pages 1–12, New York, NY, USA, 1993. ACM Press.

[10] N. Drosinos and 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)*, page 15, Santa Fe, New Mexico, Apr. 2004.

[11] M. P. I. Forum. http://www.mpi-forum.org.

[12] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can shared-memory model serve as a bridging model for parallel computation? In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 72–83, New York, NY, USA, 1997. ACM Press.

[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] P. GmbH. Pallas mpi benchmarks - pmb, http://www.pallas.de/pages/pmbd.htm.

[15] D. R. Helman and J. Jaacute;Já;. Prefix computations on symmetric multiprocessors. *J. Parallel Distrib. Comput.*, 61(2):265–278, 2001.

[16] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing 2000*, pages 50–50, 2000.

[17] M. D. Jones and R. Yao. Parallel osem reconstruction speed with mpi, openmp, and hybrid mpi-openmp programming models. In *IEEE Nuclear Science Symposium and Medical Imaging Conference Record*, Rome, Italy, October 2004.

[18] T. Kielmann, H. E. Bal, and K. Verstoep. Fast measurement of logp parameters for message passing platforms. In *IPDPS Workshops*, pages 1176–1183, 2000.

[19] R. 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.

[20] M. Kühnemann, T. Rauber, and G. Rünger. A Source Code Analyzer for Performance Prediction. In *Proc. of the IPDPS-Workshop on Massively Parallel Processing (CDROM)*. IEEE, 2004.

[21] P. Lanucara and S. Rovida. Conjugate-gradients algorithms: An mpi-openmp implementation on. In *First European Workshop on OpenMP*, pages 76–78, 1999.

[22] G. Mahinthakumar and 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):371–393, 2002.

[23] A. Majumdar. Parallel performance study of monte carlo photon transport code on shared-, distributed-, and distributed-shared-memory architectures. In *IPDPS*, pages 93–, 2000.

[24] G. Marin and J. 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*, pages 2–13, New York, NY, USA, 2004. ACM Press.

[25] K. S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 9(8):769–787, 1998.

[26] J. Odom, J. K. Hollingsworth, L. DeRose, K. Ekanadham, and S. Sbaraglia. Using dynamic tracing sampling to measure long running programs. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 59, Washington, DC, USA, 2005. IEEE Computer Society.

[27] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Rev.*, 44(3):373–393, 2002.

[28] OpenMP. http://www.openmp.org.

[29] OpenUH. http://www.cs.uh.edu/õpenuh.

[30] R. Reussner, P. Sanders, L. Prechelt, and M. Muller. Skampi: A detailed, accurate MPI benchmark. In *PVM/MPI*, pages 52–59, 1998.

[31] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 165–176, New York, NY, USA, 2004. ACM Press.

[32] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[33] SPHINX. http://www.llnl.gov/casc/sphinx/sphinx.html.

[34] C. H. Tai1, Y. Zhao, and 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):3949–3983, 2005.

[35] 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.

[36] H. W. and T. D. K. A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications. In *Parallel CFD99, Wiiliamsburg, VA*, May 1999.

[37] A. 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*, page 3, London, UK, 2003. Springer-Verlag.

[38] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286. IEEE Computer Society, 1996.