A Framework for Matching Applications with Parallel Machines

J. In, C. Jin, J. Peir, S. Ranka, and S. Sahni

Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611
{juin,chjin,peir,ranka,sahni}@cise.ufl.edu

Abstract. This paper presents a practical methodology for performance estimation of parallel applications on various supercomputers. Instead of measuring the execution time of the applications with different number of processors on each machine, we estimate the time based on characterization of the workloads. Benchmarking computation and communication primitives in the applications is also performed on parallel computer systems for the characterization. Finally, a performance model is constructed for performance estimation, and verified with different number of processors on each target system.
Our results show that accurate performance estimation is possible with the model constructed using the workload characterization method. With the result figures, we discuss the reasons why over or under estimations occur on each target machine.

1 Introduction

A typical high-performance computing environment normally consists of a suite of high-performance parallel machines and workstations connected by a high-speed interconnection network. Efficient scheduling and utilizing the available computing power are essential to maximize the overall performance in such an environment. Good performance for a given application on a parallel machine requires a good mapping of the problem onto the machine. Getting this mapping implies choosing an appropriate machine based on the application requirements. Further, many applications require integrating algorithms from diverse areas such as image processing, numerical analysis, graph theory, artificial intelligence, and databases. These problems may not be able to solve efficiently on one parallel machine because they consist of several parts, each of which requires differing types and amounts of parallelization. For such applications it may be required that different parts of the application code are executed on different machines available in a given computing environment.

There is a need for the development of an expert system tool to assist the user in effective matching of his/her application to the appropriate parallel computer(s). Ideally, we would like to design and develop a performance modeling and prediction tool that will allow users to obtain code fragments (if any) with performance or scalability bottlenecks, and derive performance for different target machines for a range of number of processors. Such a tool will allow users to develop scalable code without actually executing codes on parallel machines to derive performance.

Our goal is to be able to model the performance of a given software on a variety of different architectures. Towards the above goal, we targeted the performance assessment of a set of parallel applications on various supercomputers. Instead of measuring the run time of the applications on each machine with different number of processors, we estimated the execution time based on workload characterization. Performance measurement was used only for validation of the modeling and prediction results.

The literature abounds with work on benchmarking computation and communication primitives and estimating the performance of parallel algorithms on supercomputers. An accurate static performance estimation of parallel algorithms is possible by using basic machine properties connected with computation, vectorization, communication and synchronization [3]. Trace-driven [6, 1] and execution-driven [2] simulations are also popular for studying detailed processor performance on both uniprocessor and multiprocessor environment. Particularly, this simulation methodology is used to investigate how to improve the performance of the shared-bus, shared-memory systems [4]. In order to avoid complexities with trace-driven simulation, workload characterization techniques are used which can derive behavior of systems like inter-clustered structure, and finally estimate the performance without detailed trace simulations [8].

The rest of the paper is as follows. In Section 2, we describe the modeling of communication overhead. Section 3 describes the workload characterization. Section 4 presents experimental results. We conclude in Section 5.

2 Benchmarking and Characterization of Workload

Under the SPMD model of computation, a parallel program is partitioned into a set of *computation* and *communication* regions. A computation region can be defined as a program segment that is separated by proper synchronization and data communication primitives, while the communication region contains data communication instructions to send and receive data among different processors to satisfy the data dependence. There are several steps involved in modeling and projecting performance of parallel applications on parallel systems (Fig. 1). In order to estimate the execution time of a program on different parallel systems, we first need to benchmark the program on a base parallel system, and to characterize the workload that includes the amount of computation required for each processor in each computation region as well as the data communication requirements throughout the execution of the program. The second step is to build the performance model that consists of extrapolation of the execution time of the computation regions on the target parallel systems and estimation of the communication times based on the empirical communication model built for each target system. Thirdly, the performance models are verified by comparing the

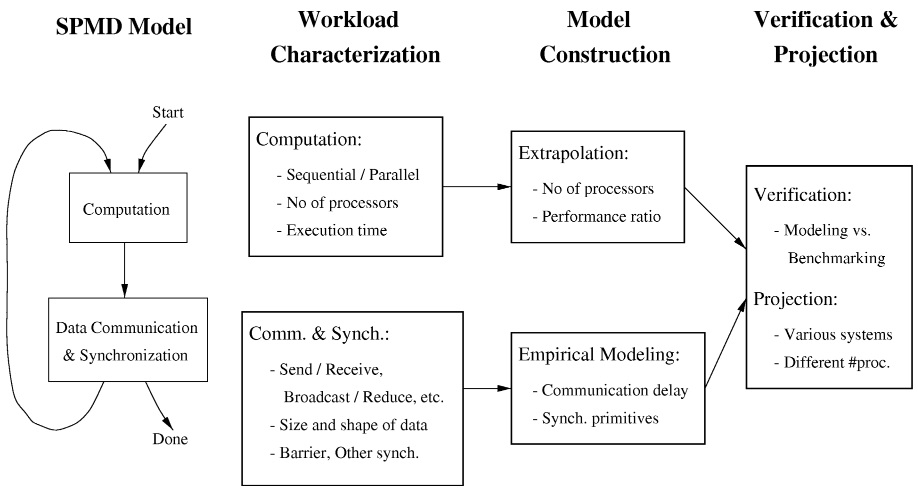

Fig. 1. Method of workload characterization and performance projection

estimated execution time with the real measured time on the target systems. Finally, we can use the model to project the performance of the selected application on different parallel systems with a range of number of processors.

2.1 Example Application

We use a finite-difference program for modeling and projection. [1] In order to estimate the execution time of this program on a CRAY T3E, IBM-SP and ORIGIN 2000, we first benchmark and characterize the workload on an 8 processor Sun/Enterprise system.

Three systems, IBM-SP2, CRAY-T3E and SGI-Origin2000, are used as target systems in this study. The IBM-SP2 system consists of 256 135 MHZ IBM RS/6000 POWER2 Super Chip (P2SC) processors, and peak computational speed of each processor is 540 MFLOPS. The system has 256 nodes, 256 gigabytes(GB) of memory in total and 2 high-Performance Parallel Interfaces(HiPPI) [5]. The CRAY T3E has 544 Processing Elements(PEs), and each PE includes a 600-MHZ DEC Alpha 21164 CPU and 256 MB of memory. The system at CEWES has 300 GFlops peak performance [9]. The SGI Origin 2000 supports up to 512 nodes which can be interconnected by a scalable network. Each node has one or two R10000 processors. A node contains up to 4GB of coherent memory, its directory memory and a connection to IO subsystem [7].

The selected program uses MPI primitives to implement parallel computation. The program divides the computational grid across the processors. The number of processors is provided as an input argument. Each processor computes its

[1] This code was provided to us By Dr. Fred Tracy at CEWES/MSRC. The main computational routine takes above 98% of the total execution time. This was the routine used in our modeling effort.

portion of the grid, and communicates with other processors to send and receive necessary results. At the end of each time step, processor zero collects the results from other processors. The program can be partitioned into three parts: the initialization phase, the main computation phase, and the output and housekeeping phase. The MPI-based time routine, MPI_WTIME(), is inserted between phases to measure their execution time.

The main computation routines are partitioned into into 19 computation regions and 4 communication regions, and each region is primarily separated by barrier synchronizations (MPI_Barrier). For modeling, each region may be further partitioned into sub-regions depending on the structure of the program. In the communication region, four MPI primitives: broadcast (MPI_Bcast), reduce (MPI_Reduce), send (MPI_Send), and receive (MPI_Recv) are used for data communication among processors.

2.2 Modeling Communication

In our study, several MPI primitives are benchmarked and modeled. The selected application includes the following four major primitives; MPI_Send, MPI_Recv, MPI_Bcast, and MPI_Reduce. Benchmarking the primitives is considered with 2, 4, 8, 16, 32, or 64 processors respectively, and input data type is fixed with floating point type. We use our benchmarking results to derive formulas for empirical modeling in communication. In the modeling, we first derive a formula for each given number of processors, then derive a simplified formula for an arbitrary number of processors.

2.3 Modeling Computation

Loop iteration is the basic structure used for modeling. We categorize loops into two types, simple and complex, based on the difficulty of modeling. Simple loops require execution time proportional to the total number of iterations. All other loops are considered to be complex.

Simple Loops There are two main scenarios for modeling simple loops on multiprocessors. In the first case, the number of iterations on each processor is inversely proportional to the number of processors. Given the measured execution time on a uniprocessor environment, we can model the execution on multiple processors based on the following simple formula.

$$M = E \times \lceil I/P \rceil \ . \tag{1}$$

where M is the execution time with multiprocessors, E is the execution time of one loop iteration on one processor, I is the number of iterations with one processor, and P is the number of processors. Note that when the total number of iterations cannot be evenly divided by the number of processors, the smallest integer which is greater than the (I/P) is used.

In the second case, each processor executes the same number of iterations as the computation is replicated on all the processors.

Complex Loops In the selected finite-difference program, a complex loop structure is encountered in several subroutines. In some cases, the loop iteration goes up with the number of processors. For a more accurate modeling, we insert counters into the complex loops. It helps measure the precise iteration numbers for different number of processors.

Avoiding timers to measure the execution time of small computation regions inside an iteration loop, we actually measure the time for the entire loop. We can then model the execution time by solving a set of equations based on the number of iterations of these small regions.

The following equations show an example. Suppose the execution times of an iteration loop are A, B, C and D measured on four different processors, and the execution times of small loops and conditional statements in the loops are T_ngh, T_iroot, T_true, and T_false when the loop iterates one time. Also we assume the numbers of iterations for the small regions are equal to Xs, Ys, Zs, and Ws. Then we can solve the equations to obtain T_ngh, T_iroot, T_true, and T_false.

$$A = X1 * T\_ngh + Y1 * T\_iroot + Z1 * T\_true + W1 * T\_false\ . \tag{2}$$

$$B = X2 * T\_ngh + Y2 * T\_iroot + Z2 * T\_true + W2 * T\_false\ . \tag{3}$$

$$C = X3 * T\_ngh + Y3 * T\_iroot + Z3 * T\_true + W3 * T\_false\ . \tag{4}$$

$$D = X4 * T\_ngh + Y4 * T\_iroot + Z4 * T\_true + W4 * T\_false\ . \tag{5}$$

Note that the actual number of equations can be very large due to the fact we can measure the execution time of the outside loop on different processors with various number of total used processors. In order to increase the estimate accuracy, we use the statistical analysis tool, SAS to obtain more accurate timing for these small regions.

Compiler optimizations The three systems, CRAY T3E IBM-SP2 and SGI Origin 2000 have several levels of compiler optimization respectively, and the default level of each is different. When the performance of a program is compared on the systems, the compiler optimization level is required to be same.

In the selected program, A few complex loops are restructured by the compiler optimization. As a result, the total number of the execution of the loops may be much smaller than the number which the high-level code indicates. This is especially true with a large number of processors.

The situation is further accentuated by the fact that the number of iteration in a loop in a ‘measured’ execution with inserted iteration counters does not match with the actual number in the ‘real’ execution. This is due to the fact that when counters are inserted, the compiler can no longer restructure the loop. For the purpose of this study, instead of rewriting the loop code, we insert dummy counters inside the loops to prevent the compiler from restructuring the loop.

Table 1 shows the differences in execution time with and without dummy counters inserted in a loop statement of a selected application. We can observe that the difference becomes bigger as the number of processors grows. The values in table 1 result from compiling the application program with default optimization level.

No. of Procs	Cray T3E		IBM-SP		Origin 2000	
	W/O (s)	With (s)	W/O (s)	With (s)	W/O (s)	With (s)
1	17.04	17.06	110.5	110.6	108.3	106.2
2	9.002	9.008	57.52	57.76	55.93	54.86
4	4.544	4.559	29.77	29.92	28.63	28.09
8	2.459	2.534	16.21	16.45	15.20	15.37
16	1.516	1.854	11.98	12.01	10.01	10.09
32	1.182	2.672	16.77	18.86	12.77	14.39
64	3.078	7.382	50.87	56.74	34.20	38.95

Table 1. The execution time with/without dummy counter (optimization: default)

3 Model Verification and Performance Projection

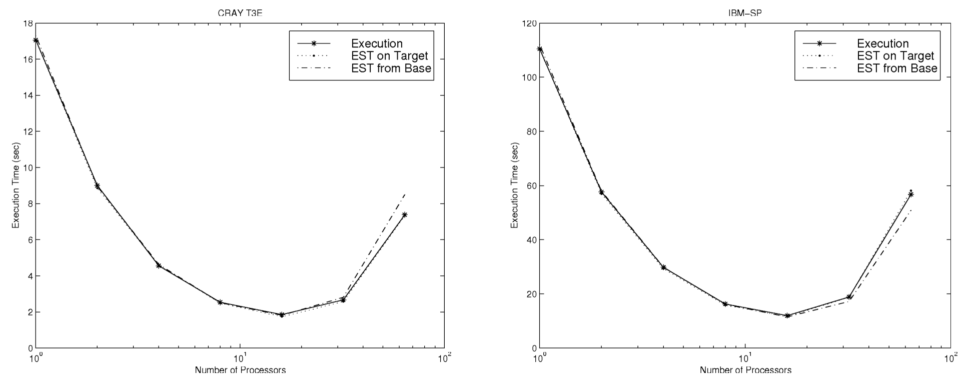

Fig. 2. Performance estimation on CRAY-T3E and IBM-SP2 (EST: Estimation)

We first develop and verify our performance model on a local SUN/Enterprise system with 8 processors. This is used as the base machine for modeling and extrapolating the time to other machines. We show the results of two approaches for the targeted machines. First, we estimate the execution times of a subroutine based on the performance model built on the base system. In order to extrapolate the execution times on the target systems, we used a *performance ratio* between the base and the target systems by comparing the execution time of the selected application running on a uniprocessor environment. Second, we also estimate the performance of the subroutine based on the model built on each target system. Basically, it is the same as constructing the model on the base and the target systems. However, the second approach should provide more accurate results by factoring out the performance ratios between two different systems.

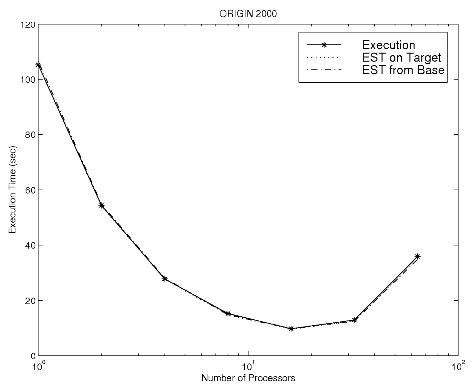

Fig. 3. Performance estimation on SGI-Origin2000 (EST: Estimation)

Figures 2 and 3 summarize the estimated and measured execution times on the three target systems with both approaches. We can make the following observations.

1. The CRAY-T3E provides about 5 times faster execution than the other two machines. This is due to the fact that we use default optimization in compiling the program in all three systems. It turns out that the default optimization level on Cray-T3E is level 2, while level 0 is default on the other two machines.
2. The estimated execution times based on the model built on the SUN/Enterprise are very accurate up to 32 processors on all three target systems. However, we observe about 15%, -10%, and -3% inaccuracy with the 64-processor CRAY-T3E, IBM-SP2, and SGI-Origin2000 respectively. This can be partially attributed to the fact that the original program was not designed for a large number of processors as can be seen by the increase in execution time beyond 16 processors. Another possible reason for the 15% overestimation on the 64-processor CRAY-T3E system is because of the presence of complex loop structures. Furthermore, due to the diversity of system architectures, the simple performance ratio to extrapolate performance between two systems can create additional misprediction.
3. Our modeling is much more accurate when the performance models are built on each target system. As shown in the above figures, the under/over estimation of the 64 processors can be reduced to about 2.5% in the worst case. The small over-estimation is due mainly to the fact that the potential misprediction factor across different machines is eliminated, which exists in the estimation from the base system. In addition, the execution time equations established from the MATLAB model provide very good estimates of the measured execution times.

4 Conclusion

In this research, a methodology for designing and developing a performance modeling and prediction tool has been suggested, and presented with performance results on three parallel machines. The tool allows users derive performance of applications on different machines with different number of processors without actually executing codes on parallel machines. The strategy is good for quick performance estimation to enable more efficient usage of parallel systems.

The application which we choose as an example for demonstrating the workload characterization method is a simple finite-difference program. Even though the results which are shown from applying the tool to the application have been accurate, it has a limitation to expand the approach to other applications. Other complex programs could not be partitioned into a set of computation and communication regions to characterize workload as we do with the selected program. As people have noticed, a general approach for all programs on all parallel platforms is difficult, and still an open problem.

Acknowledgments

This research was partially supported U.S. Army contract #DACA39-97-K-0037. We also thank Dr. Radha and Dr. Turcotte for providing us access to the three parallel systems at CEWES/MSRC including the IBM-SP2, the CRAY-T3E, and the SGI-Origin2000.

References

1. Bob Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," 1994 Sigmetrics, (1994), pp. 128-137.
2. Helen Davis, Stephen Goldschmidt, and John Hennessy, "Multiprocessor Simulation and Tracing Using Tango," Proc. 1991 Int'l Conf. on Parallel Processing, Vol. II, (1991), pp. 99-107.
3. Kivanc Dincer, Zeki Bozkus, Sanjay Ranka, and Geoffrey Fox, "Benchmarking the Computation and Communication Performance of the CM-5," Concurrency: Practice and Experience Vol. 8, No. 1, (1996), pp. 47-69.
4. Roberto Giorgi, Cosimo Antonio Prete, Gianpaolo Prina, and Luigi Ricciardi, "Trace Factory Generating Workloads for Trace-Driven Simulation of Shared-Bus Multiprocessors," IEEE Concurrency, Vol. 5, No. 4, (1997), pp. 54-68.
5. http://www.wes.hpc.mil/documentation/ibmspdoc/ibmsp_faqs.html
6. Manoj Kumar and Kimming So, "Trace Driven Simulation for Studying MIMD Parallel Computers," Proc. 1989 Int'l Conf. on Parallel Processing, Vol. I, (1989), pp. 68-72.
7. James Lauden, and Daniel Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," 24th Int'l Symp. on Computer Architecture, (1997), pp. 241-251.
8. Lishing Liu and Jih-Kwon Peir: A Performance Evaluation Methodology for Coupled Multiple Supercomputers. Proc. 1990 Int'l Conf. on Parallel Processing, Vol. I, (1990), pp. 198-202.
9. http://www.wes.hpc.mil/documentation/t3edocs/t3e_faqs.html.