A Framework to Develop Symbolic Performance Models of Parallel Applications

Sadaf R. Alam Jeffrey S. Vetter

Oak Ridge National Laboratory
Oak Ridge, TN, USA 37831
{alamsr,vetter}@ornl.gov

Abstract

Performance and workload modeling has numerous uses at every stage of the high-end computing lifecycle: design, integration, procurement, installation and tuning. Despite the tremendous usefulness of performance models, their construction remains largely a manual, complex, and time-consuming exercise. We propose a new approach to the model construction, called modeling assertions (MA), which borrows advantages from both the empirical and analytical modeling techniques. This strategy has many advantages over traditional methods: incremental construction of realistic performance models, straightforward model validation against empirical data, and intuitive error bounding on individual model terms. We demonstrate this new technique on the NAS parallel CG and SP benchmarks by constructing high fidelity models for the floating-point operation cost, memory requirements, and MPI message volume. These models are driven by a small number of key input parameters thereby allowing efficient design space exploration of future problem sizes and architectures.

1 Introduction

Performance and workload modeling has numerous uses at every stage of the high-end computing lifecycle: design, integration, procurement, installation, tuning, and maintenance. Despite the tremendous usefulness of performance models, their construction remains largely a manual, complex, and time-consuming exercise. In most cases, researchers create models by manually interrogating applications with an array of performance, debugging, and static analysis tools to refine the model iteratively until the predictions fall within expectations. In other cases, researchers start with an algorithm description, and develop the performance model directly from this abstract description.

In this paper, we describe a new approach to performance model construction, called ***modeling assertions*** (MA), which borrows advantages from both the empirical and analytical modeling techniques. This strategy has many advantages over traditional methods: isomorphism with the application structure, easy incremental validation of the model with empirical data, uncomplicated sensitivity analysis, and straightforward error bounding on individual model terms. We demonstrate the use of MA by designing a prototype framework, which allows construction, validation, and analysis of models of parallel applications written in FORTRAN and C with the MPI communication library. We use the prototype to construct models of NAS CG and SP benchmarks [4].

MA generates two types of representations of the target application: control flow models and symbolic models that can be evaluated with MATLAB or Octave. Symbolic models are generated for the number of floating-point and memory operations, and for MPI point-to-point and collective communication operations. Control flow models provide a mechanism not only to understand the control flow of an application but also to generate alternate model representations in programming languages like C or Python. The models are represented in terms of an application's input parameters. Thus, an application parameter space can be explored efficiently using the MA models. Furthermore, the control flow models can be extended to produce synthetic traces like the ones generated the by MPIDtrace utility [9] to predict communication performance of an application.

In addition, the symbolic models can project performance requirements and allow us to conduct sensitivity analysis of workload requirements for future and larger problem instances of an application. For example, we studied the growth rate of the number of the floating-point operations with respect to input parameters (e.g., the array size and the number of nonzero elements) for the NAS CG benchmark. Our results show that the floating-point operation cost increases at a much faster rate by increasing the number of nonzero elements than by increasing the array size.

1-4244-0054-6/06/$20.00 ©2006 IEEE

The outline of the paper is as follows: the motivation behind the modeling assertion technique is presented in section 2. Section 3 explains the components of the modeling assertion framework, and describes model construction and validation using the NAS CG and SP benchmarks. Section 4 presents the scalability of the NAS CG and SP benchmarks together with an analysis of sensitivity of workload requirements to the input parameters of the benchmarks. Section 5 concludes with benefits and contributions of the modeling assertions approach to performance modeling studies.

2 Motivating Examples

During the past two decades, the high-end computing (HEC) community has successfully used performance modeling for many activities. In this respect, researchers have proposed numerous techniques for the creation and validation of performance models, ranging from fully analytical, manual methods to automatic, architecture-specific approaches. Many of these techniques serve the overall purpose of modeling but there have been few common techniques have gained widespread acceptance across the community. This situation has resulted in additional problems including lack of modular, interoperable performance models, and a lack of tools for model creation, analysis, and validation.

The two basic methodologies for constructing performance models are best represented by two case studies. The first case study develops a performance model of a protein folding application for the Blue Gene architecture using a top-down, analytic approach. The second case study demonstrates an empirical modeling approach based on static and dynamic observations of a target application on a specific architecture. The output of both of these approaches is a predictive performance model. With this model in hand, both methods typically use measurements, simulations, and other performance estimates to generate performance predictions for an application on a specific architecture.

2.1 Top-down Algorithmic Model Creation

Many researchers have created analytical models of important kernels and applications [3][5]. These models range from calculating the number of operations necessary to complete a common mathematical operation, such as a matrix multiply, to complete models of entire codes, such as protein folding. Almasi, et al. [3] present such a model of the protein folding application for the original Blue Gene architecture. Their analysis eloquently decomposes the application into its main computational and communication kernels. They, then, use simulations and analytical models to generate performance predictions for various system and application configurations.

2.2 *ab initio* Model Creation from Empirical Observation

Another approach to model creation is based on static and/or dynamic analysis of the characteristics of an application on a specific architecture, using these characteristics to generate an *ab initio* model. One such system developed for empirical performance modeling is the POEMS system [1]. POEMS used an elaborate compiler system along with message tracing to generate a performance model of target applications. To evaluate the model with different configurations, users could, as in the analytical modeling case, investigate a variety of important machine parameters with simulators, modeling, or measurement, and, then, feed those values into the existing performance model.

2.3 Observations

Both top-down, analytical and *ab initio* techniques have benefits and both generate valuable results. However, both of these techniques also have shortcomings that have limited their widespread adoption in the HEC community.

Empirical, *ab initio* model creation (2.2) typically relies on some combination of static and dynamic analysis. Developers must use tools [2] like debuggers, performance monitors, and memory tracers [6] to ‘reverse engineer’ the application to generate a performance model. If the application has dynamic data dependencies in the control flow, then the exercise becomes even more complex. Moreover, this automated process might discover hundreds of input parameters; whereas application users typically recognize that only a subset of input parameters may be important for a performance model. Users usually can easily identify these important parameters; however, that information is lost when relying solely on an automated process. Another disadvantage to this approach is that it has limited applicability to large, complex scientific applications. Real applications, such as those used in the DoE SciDAC program [7], have hundreds of thousands of lines of multi-language source code. Unless the automated tools take care to omit frequent operations, such as trillions of memory loads and stores, the tools can be impractical. Yet another issue with automated tools is the lack of separation between the architecture *dependent* and the architecture *independent* characteristics of the application. Take, for example, a series of memory operations fetching a floating point number through a permutation index array. Each access to the floating point number is restricted by the prior access to the permutation array; however, since many accesses through this permutation index array can occur concurrently, the latencies for these operations may be overlapped (and hidden). In contrast, when relying on memory address tracing on a specific architecture, it would be impractical and error-prone to ‘reverse engineer’ a compiler-mutated

address stream to this level of detail. This problem is even more aggravated by the recent introduction of vector and multithreaded architectures that can provide scatter/gather hardware for these types of operations.

Top-down analytic model creation is the most common method of creating a performance model. Any application user can interrogate their application, develop a structure for the model, introduce model terms, and refine the model until it represents the performance with reasonable fidelity. On the positive side, these types of models typically emphasize exactly the important variables, input parameters, and operations that are important to each application. Although this modeling technique is pervasive, it remains difficult, time-consuming, and error-prone. Furthermore, because every user develops an independent model, there are few (if any) widespread performance modeling languages and toolkits. Without these tools, the process of validating and debugging performance models remain an ad-hoc endeavor.

3 Modeling Assertions Overview

Based on these observations, we have designed *Modeling Assertions* (MA) to facilitate the creation of symbolic performance models. MA is a new technique that combines the benefits from both analytic and empirical approaches, and it adds some new advantages, such as incremental model validation. Advantages gained from modeling assertions include the following:

1. The symbolic models are parameterized, architecture-independent models that can be evaluated with a variety of tools, such as MATLAB.
2. MA performance models are modular, portable, and composable.
3. Users construct performance models incrementally with MA, validating components of the model, and adding more resolution for significant operations as necessary.
4. Models constructed with MA reflect the structure of the application so that application users can understand and easily modify the model. Furthermore, if entirely new architectures are proposed, it is straightforward to incorporate models for these new architectures into existing models.

From the perspective of constructing, validating, and evaluating performance models, we believe that MA offers many benefits over conventional techniques throughout the performance lifecycle.

3.1 MA Framework

In order to evaluate our approach of developing symbolic models with MA, we have designed a prototype framework. This framework has two main components: an API and a post-processing toolset. Figure 1 shows the components of the MA framework. The MA API is used to annotate the source code. As the application executes, the runtime system captures important information in trace files. These trace files are then post-processed to validate, analyze, and construct models. The post-processing toolset is a collection of tools: a model validation tool, a control-flow model creation tool, and a symbolic model generation tool. For example, the symbolic model shown in Figure 1 is for the MPI send volume of an application. This symbolic model can be evaluated and is compatible with MATLAB and Octave scripts.

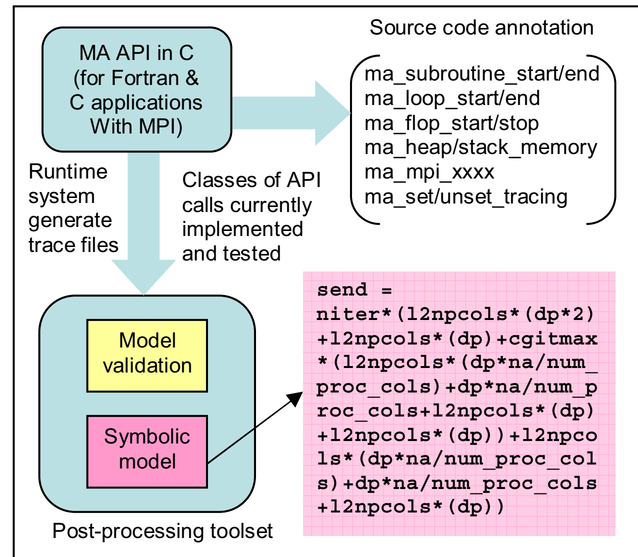

Figure 1: Design components of the Modeling Assertion (MA) framework

The modeling assertion API provides a set of functions to annotate a given FORTRAN or C code. For example, `ma_loop_start`, an MA API function, can be used to mark the start of a loop. Upon execution, the code instrumented with MA API functions generates trace files. For parallel applications, one trace file is generated for each MPI task. The trace files contain traces for `ma_xxx` calls and MPI communication events. Most MA calls require a pair of `ma_xxx_start` and a `ma_xxx_end` calls. A typical format for a `ma_xxx_start` and `end` calls are shown Figure 2:

```
ma_xxx_start(char *identifier, char
*expression, int exp_value)
ma_xxx_name(char *identifier, int
measured_value)
```

Figure 2: Pair of Modeling Assertion API calls.

The `ma_xxx_end` traces are primarily used to validate the modeling assertions against the runtime values. The assertions for hardware counter values, `ma_flop_start/stop`, invoke the PAPI hardware counter API [9]. The `ma_mpi_xxx` assertions on the other hand are validated by implementing MPI wrapper functions (PMPI) and by comparing `ma_mpi_xxx` traces to `PMPI_xxx` traces. Additional functions are provided in the MA API to control the tracing volume, i.e. size of the trace files, by

enabling and disabling the tracing at compile time and also at runtime.

Figure 3 shows the step-by-step process for creating a performance model with MA. To create a performance model with MA, a user must instrument their application with source code annotations that describe the important variables and operations in their application.

1. Declare important application variables.
2. Declare important application operations.
3. Incrementally refine performance model
   a. Validate performance model empirically at runtime using performance assertions.
   b. Refine based on these error rates by adding and modifying variable and operation declarations.
   c. Terminate when model is representative and when error level is acceptable.

Figure 3: Process of model creation with modeling assertion framework

3.2 Model Creation

As a motivating example, we demonstrate MA on NAS CG benchmark. NAS CG computes an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix, which is characteristic of unstructured grid computations. The main subroutine is `conj_grad`, which is called `niter` times. The benchmark results report time in seconds for the time step (`do it = 1, niter`) loop that is shown in Figure 4. Hence, we started constructing the model in the main program, starting from the `conj_loop`, as shown is Figure 4. The first step was to identify the key input parameters, `na`, `nonzer`, `niter` and `nprocs` (number of MPI tasks). Then, using `ma_def_variable_assign_int` function, we declared the essential derived values, which are later used in the MA annotations to simplify the model representation. Figure 4 only shows the MA annotations as applied to NAS CG main loop. It does not show the annotations in the `conj_grad` subroutine, which are similar except for additional `ma_subroutine_start` and `ma_subroutine_end` calls. The annotations shown in Figure 4 capture the overall flow of the application at different levels in the hierarchy: loops (e.g., `conj_loop`, mpi), subroutines (e.g., `conj_grad`), basic loop blocks etc. These annotations also define variables (e.g., `na`, `nonzer`, `niter`, `nprocs`), which are important to the user in terms of quantities that determine the problem resolution and its workload requirements. At the lowest level is the `norm_loop` as shown in Figure 4. The `ma_loop_start` and `ma_loop_end` annotations specify that the enclosed loop executes a number of iterations (e.g., `l2npcols`) with a typical number of MPI send and receive operations. More specifically, the `ma_loop_start` routine captures an annotation name (i.e., `l2npcols = log2(num_proc_cols)`), a symbolic expression that defines the number of iterations in terms of an earlier defined MA variable (i.e., `num_proc_cols`).

```
call ma_def_variable_int('na',na)
call ma_def_variable_assign_int
('num_proc_cols',
'2^ceil(log(nprocs)/(2*log(2)))',
num_proc_cols)
call ma_def_variable_assign_int( 'l2npcols',
'log(num_proc_cols)/log(2)', l2npcols)
........
call ma_loop_start('conj_loop', 'niter',
niter)
  do it = 1, niter
    call conj_grad ( colidx,
reduce_recv_lengths )
........
    call ma_loop_start('norm_loop',
'l2npcols', l2npcols)
    do i = 1, l2npcols
      call ma_mpi_irecv('nrecv','dp*2',
dp*2,1)
      call mpi_irecv( norm_temp2, 2, dp_type
........
    call ma_loop_end('norm_loop',i-1)
........
```

Figure 4 : Annotation of the CG benchmark with MA API calls

Within the `norm_loop` the sizes of MPI send and receive operations are given in symbolic expressions as well. In the `norm_loop`, the message sizes for send and receive operations are constant (16 bytes), while in the some cases, this could depend on input parameters and workload distribution schemes. For example, in Figure 5, the number of floating-point operations depends on a derived variable, `num_proc_cols`, which in turn depend on the number of MPI tasks.

```
call ma_flop_start ('flopzeta', '4 * na
/ num_proc_cols', 4*na/num_proc_cols)
do j=1, lastcol-firstcol+1
   norm_temp1(1)=norm_temp1(1)+x(j)*z(j)
   norm_temp1(2)=norm_temp1(2)+ z(j)*z(j)
enddo
call ma_flop_stop('flopzeta')
```

Figure 5: FP MA based on an input variable (na) and a derived variable (num_proc_cols)

3.3 Runtime Execution

At runtime, the MA runtime system (MARS) tracks and captures the actual instantiated values as they execute in the application. MARS creates an internal control flow representation of the calls to the MA library as they are executed. It also captures both the symbolic values and the actual values of the expressions. These values are stored in the trace files. The MA post-processing toolset processes the trace files and creates an intermediate representation file (shown in Figure 6) for individual MPI tasks. Multiple calls to the same routines with similar parameters maps

onto the same call graph, therefore, the data volume is manageable.

```
1:num_proc_cols:2^ceil(log(nprocs)/(2*log
(2)))
2:loop:niter
3:loop:l2npcols
4:mpi_irecv:dp*2
4:mpi_send:dp*2
4:loop:cgitmax
5:loop:l2npcols
6:mpi_irecv:dp*na/num_proc_cols
........
```

Figure 6: Intermediate model representation.

The first integer in Figure 6 is the context of the calling function that reveals the stack depth of the annotations, which may be different than the actual stack depth of the execution call graph itself. Variables derived from input parameters like `num_proc_cols` do not have an associated MA type like `mpi_xxx` operations. This intermediate representation is used to create user-defined symbolic models. For example, a user may wish to identify the computational intensity within a loop or within an important function using `ma_flop` and `ma_load/store` assertions. Similarly, other quantities like byte-to-flop and message-volume-to-flop ratios can also be studied at different resolutions.

3.4 Validating Modeling Assertions

The validation of an MA performance model is a two-stage process. When a model is initially being created, validation plays an important role in guiding the resolution of the model at various phases in the application. Later, the same model and validation technique can be used to validate against historical data and across the parameter space.

Although the model validation tool is part of the post-processing toolset, it can be invoked separately. This helps in the model building process, as the key parameters and expressions are being identified. For instance, in Figure 7, the MPI send message size is (`send_len * sizeof (double)`). The `send_len` is calculated from the input parameters `na`, `nprocs`, and subsequently from `num_proc_cols`. Hence, the incremental process enables a user to design the model with only a small number of key input parameters.

```
call ma_mpi_send('l3snd',
'dp*na/num_proc_cols', ............
call mpi_send(w(send_start),
send_len,dp_type,............
```

Figure 7: Send message size depend on na and a derived parameter (num_proc_cols).

A model validation output file is shown in Figure 8. The format is `ma_id: type: predicted_value: measured_value: error_rate: passes: failures`. In the NAS CG benchmark, the sizes of MPI messages do not change at runtime, therefore, all MPI assertions are successful. On the other hand, the numbers of floating-point operations are dependent on the compiler and underlying system design. Nonetheless, we can define an acceptable error rate. In this case, the maximum error rate is less than 20%, which is acceptable for the floating-point operation count.

```
cgmain:ma_loop:25:25:0.0:PASS=2:FAIL=0
floprhopq:ma_flop:21001:21001:0.0:.PASS=50:
   FAIL=0
cj_rho:ma_loop:1:1:0.0: PASS=50: FAIL=0
l5rcv:ma_mpi_irecv:8:8:0.0:PASS=50:FAIL=0
l5snd:ma_mpi_send:8:8:0.0: PASS=50: FAIL=0
flopbeta:ma_flop:7002:7001:1.426E-4: PASS=6:
   FAIL=44
flopnzx:ma_flop:3503:4347:-0.194: PASS=0:
   FAIL=2
```

Figure 8: Model validation output.

In order to benchmark parallel systems with Teraflops-scale processing power and increasing memory capacities, the NAS parallel benchmarks provide different problem resolutions or classes. Class S is typically the smallest problem size, which is considered too small to even benchmark a current single processor system. However, these problem instances provided us a mechanism to develop, validate, and test the MA approach. Table 1 lists the input parameters for the NAS CG and SP benchmarks that define the problem resolution. In addition to input parameters listed in Table 1, the number of MPI tasks (also an input parameter) determines some derived parameters like the `num_proc_cols` in CG and the square-root of number of processors in the SP benchmark.

Class	CG	SP
S	na=1400, nonzer=7	problem_size=12
W	na=7000, nonzer=8	problem_size=36
A	na=14000, nonzer=11	problem_size=64
B	na=75000, nonzer=13	problem_size=102
C	na=150000, nonzer=15	problem_size=162

Table 1: Values of the input parameters for the different problem instances (Classes)

We validated our MA models on an IBM p690 SMP cluster. Each processing node on the SMP cluster is composed of 32 Power4 processors with floating-point MAC units. The runtime hardware counter data is collected using the PAPI hardware counter `PAPI_FP_OPS` since the `PAPI_FP_INS` considers a multiply and add instruction as a single FP execution entity [8].

Figure 9 and Figure 10 compare the measured value with the predicted values of the number of floating-point operations for single time step iteration per processor for NAS CG and SP benchmarks, respectively. For both CG and SP benchmarks, the maximum error rate is less than 30% (typical error rate < 10%). The MPI assertions are validated using the PMPI (MPI profiling library) runtime values. There is no error for CG and SP MPI assertions,

since the message sizes and message count do not change at runtime.

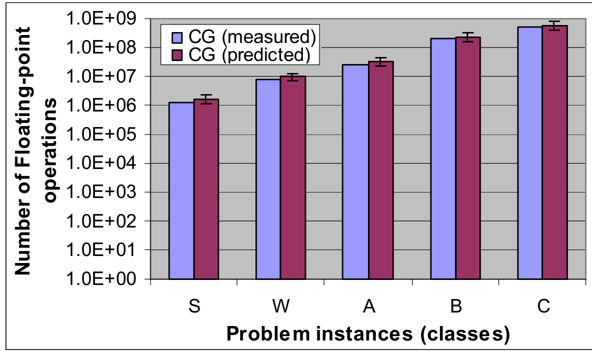

Figure 9: Measured vs. predicted number of FP operations for NAS CG

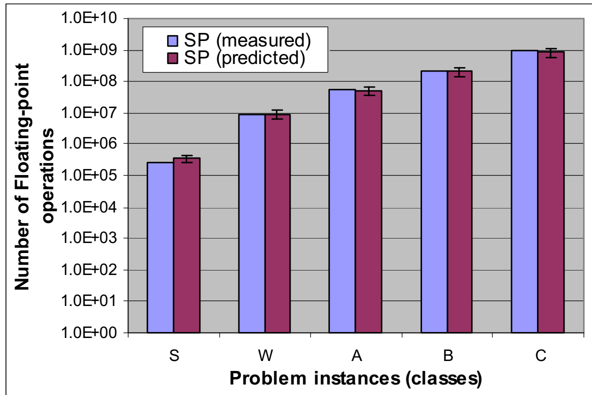

Figure 10: Measured vs. predicted number of FP operations for NAS CG

3.5 MA Model Output

Upon termination of a runtime experiment, MA outputs a control flow model representation (shown in Figure 11), an intermediate file (Figure 6), and symbolic models for three factors: number of floating-point, load-store, and communication operations. The control flow model representation is similar to the actual code annotations (Figure 4); that is, it is a high level, visual flow of the annotated parts of the application. From this representation, a user can create several different models with important application parameters.

The intermediate representation serves as an input to develop symbolic models for user-defined characteristics, such as the send message volume or relative quantities like computational intensity or memory byte-to-flop ratio. For instance, a user can create models for load/store-to-flop ratios using the intermediate representation (figure 1 shows MATLAB/Octave compatible symbolic model representation for CG communication operations). Only three input parameters are required to evaluate this model, `nprocs` (number of MPI tasks), `na`, and `nonzer`. The goal is to be able to generate symbolic models that represent the architecture independent requirements of an application and that can be evaluated efficiently by existing mathematical software frameworks.

```
loop (NAME=conj_loop)(COUNT=niter)
  {
   loop (NAME=norm_loop)(COUNT=l2npcols)
      {
        mpi_irecv(NAME=nrecv)(SIZE=dp*2)
        mpi_send(NAME=nsend)(SIZE=dp*2)
        mpi_wait (NAME = nwait)
      }
    conj_grad()
      {
        loop (NAME=cj1)(COUNT=l2npcols)
```

Figure 11: Example of a control-flow model

4 Evaluation of Modeling Assertions

4.1 Scaling Studies

We conducted scalability studies of the NAS CG and SP benchmarks using their validated MA models.

Figure 12 shows the message volume and the message count for the two benchmarks for Class C problem instance. The message size to message count ratio can determine whether an application sends a large number of small messages or small number of very large messages.

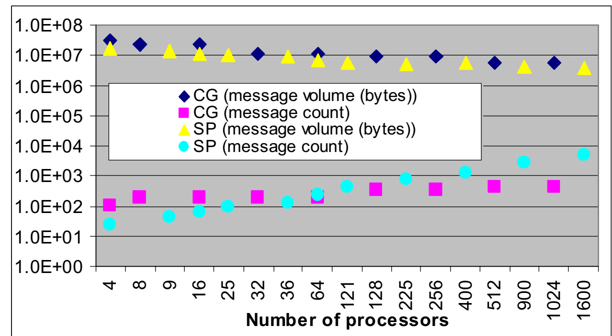

Figure 12: Message size (bytes) and message count scaling for NAS CG and SP

Generally, small messages are considered *latency bound* while large messages are *bandwidth bound* on large-scale systems. We calculated the message count and size distribution for Class C CG benchmark on 1024 processors. Approximately 65% of messages are 8 bytes while the remaining messages are over 37 Kbytes. For a 1600 processor SP Class C run, over 95% of messages are about 28 Kbytes, 2% are 50 Kbytes and 1% are 64 Kbytes. Hence, a first-order analysis of the overall communication pattern shown in

Figure 12 does not determine the actual message distribution. Thus, an analysis that is primarily based on

the data shown in Figure 12 could be misleading. A detailed model generated by MA is necessary to understand the communication behavior for larger problem instances and large-scale system runs. Using this analysis, we can determine that a machine with a very high communication bandwidth and with slightly higher MPI latency like Cray XT3 (PingPong latency=~6usec) will not scale the CG problem as shown in Figure 13. The SP benchmark scales because the message sizes are not too small (order of tens of Kbytes).

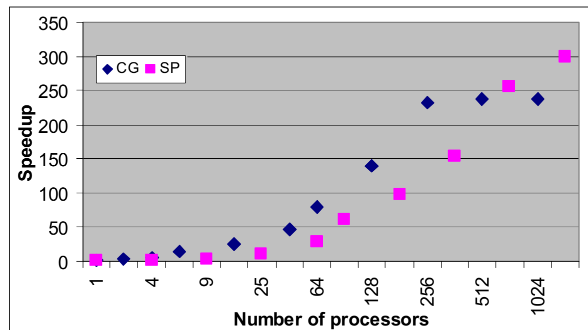

Figure 13: Speedup on the Cray XT3.

The scaling of the number of floating-point operations is shown in Figure 14. Class C problem instances are scaled up to 1024 processors for the CG benchmark and up to 1600 processors for the SP benchmark. The scaling pattern for the two benchmarks is similar because the way in which the MPI tasks are distributed is CG ($\log_2(\log_2(\text{MPI_tasks}))$), the floating-point operation count does not scale linearly with the number of processors. The SP benchmark uses a square-root of MPI processor grid and the scaling of the floating-point operation count scales to the square-root of the MPI tasks. Nonetheless, the sequential components of the code dominate the floating-point operation count when large numbers of MPI tasks are used. Thus, the floating-point operation count does not change significantly on 900 and 1600 processor runs.

4.2 Sensitivity Analysis

One of the aims of creating the models of applications is to be able to predict the application requirements for the future problem sizes. We used our MA models to understand the sensitivity of floating-point operations, memory requirements per processor, and message volume to applications' input parameters. We started with a validated problem instance, Class C, for both the NAS CG and SP benchmarks, and scaled the input parameters. For instance, na=150000 for CG Class C benchmark are x1 in the figure below. For x2, na=300000 and for x10 na=1500000.

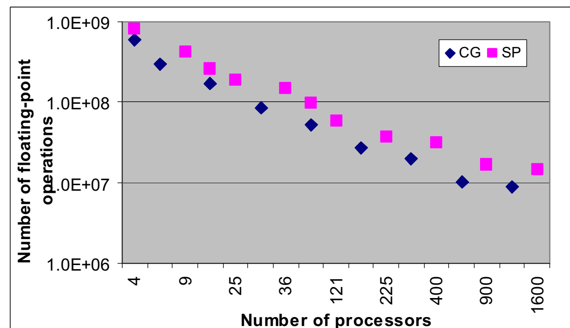

Figure 14: Scaling behavior of the number of FP operations of NAS CG and SP benchmark

As shown in Figure 15, the floating-point operation cost in CG is more sensitive to the increase in the number of `nonzer` elements in the array. On the other hand, the `nonzer` parameter has no effect on the message volume. The floating-point operation count and average memory per processor increase almost linearly with increase in `na`.

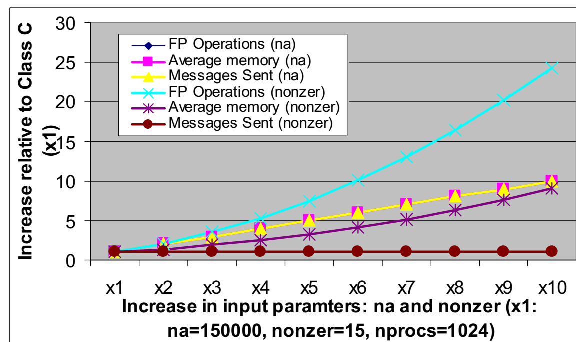

Figure 15: Sensitivity of NAS CG workload requirements as a function of input parameters, na and nonzer.

The affect of the SP's input parameter, `problem_size`, to the system requirements in terms of number of floating-point operations, average memory requirements, and message volume is shown in Figure 16. The floating-point operation cost increases exponentially by increasing the `problem_size`. The memory requirements and message volume also increase significantly with an increase in the `problem_size` parameter.

Using the MA models, we can get an insight into the workload distribution and scaling behavior of the number of floating-point operations within an application as a function of the `problem_size` parameter. Figure 17 shows contribution of different functions in SP time step iterations. The `z_solve` is the most expensive function for runs with large number of processors. The cost of `x_solve` and `y_solve` are identical and consistent. Moreover, we can safely ignore cost of `txinvr` and `add` functions in the further analysis.

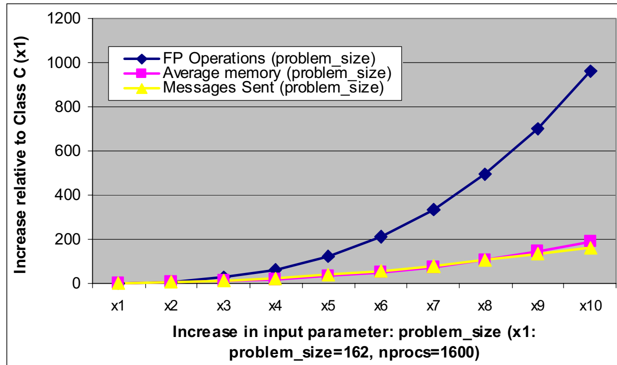

Figure 16: Sensitivity of application requirements as a function of NAS SP input parameter, problem_size.

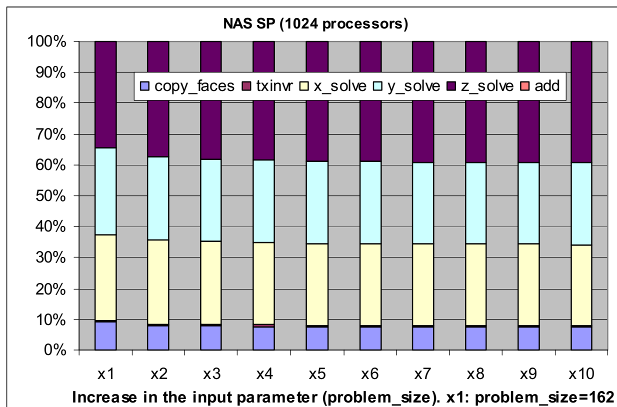

Figure 17: Impact of individual functions on the overall increase in the number of FP operations

5 Conclusions

Clearly, performance modeling has numerous uses at every stage of the high-end computing lifecycle: design, integration, procurement, installation, tuning, and maintenance. But despite the tremendous usefulness of performance models, their construction remains largely a manual, complex, and time-consuming exercise. In this paper, we have proposed a new approach to symbolic performance model construction, called ***modeling assertions*** (MA), which borrows advantages from both the empirical and analytical modeling techniques. MA has many advantages over traditional methods: incremental construction of a realistic performance model, isomorphism with the application structure, straightforward model validation against empirical data, and intuitive error bounding on individual model terms. We demonstrate this new technique on the NAS CG and SP benchmarks, and show that MA does make the construction and use of performance models more practical than either empirical or analytical modeling techniques alone.

Acknowledgements

This research was sponsored by the Office of Mathematical, Information, and Computational Sciences, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Batelle, LLC. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

References

[1] V.S. Adve, R. Bagrodia *et al.*, “POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems,” *IEEE Trans. Software Engineering*, 26(11):1027-48, 2000.

[2] V.S. Adve, R. Bagrodia *et al.*, “Compiler-supported Simulation of Highly Scalable Parallel Applications,” Proc. SC99 (electronic publication), 1999.

[3] G.S. Almasi, C. Cascaval *et al.*, “Demonstrating the scalability of a molecular dynamics application on a Petaflop computer,” Proc. Int'l Conf. Supercomputing, 2001, pp. 393-406.

[4] D. Bailey, E. Barszcz *et al.*, “The NAS Parallel Benchmarks (94),” NASA Ames Research Center, RNR Technical Report RNR-94-007, 1994, http://www.nas.nasa.gov/Pubs/TechReports/RNRreports/dbailey/RNR-94-007/RNR-94-007.html.

[5] M.J. Clement and M.J. Quinn, “Analytical performance prediction on multicomputers,” Proc. SC 1993, 1993, pp. 886-94.

[6] A. Snavely, L. Carrington *et al.*, “A Framework for Performance Modeling and Prediction,” Proc. SC 2002 (electronic publication), 2002.

[7] US Department of Energy Office of Science, “A Science-Based Case for Large-Scale Simulation,” US Department of Energy Office of Science 2003, http://www.pnl.gov/scales.

[8] S. Browne, J Dongarra, N Garner, G. Ho, P Mucci, “A Portable Programming Interface for Performance Evaluation on Modern Processors, ” *The International Journal of High Performance Computing Applications*, Volume 14, number 3, pp. 189-204, Fall 2000.

[9] G. Rodriguez, R. Badia, and J. Labarta, “Generation of Simple Analytical Models for Message Passing Applications, ” Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, 2004.