Specification and Performance Metrics for Parallel Programs

Brian J. d'Auriol
Department of Computer Science
The University of Texas at El Paso
El Paso, TX, USA 79968
Email: dauriol@acm.org

Juan Ulloa
Department of Computer Science
The University of Texas at El Paso
El Paso, TX, USA 79968
Email: juanu@utep.edu

***Abstract*— *This paper describes a model for specification and performance metrics of parallel programs. A specification metric qualifies and quantifies the abstraction of a code fragment written in some computer language. A performance metric describes the run-time effects of a specification. A brief statement of the model is presented. An application to a C/MPI program that describes specification and performance metrics is described.*

Keywords: Specification, Performance, Software Metrics, Parallel programming.

I. Introduction

The programming of parallel computers may be thought of as an exercise in specifying communication-linked concurrent computations. The primary issue is to specify subsets of computations that are allocated to different processors. In so doing, concurrency is enabled. This process is often referred to as algorithm partitioning (decomposition) and allocation. Concurrent computations process input data and generate output data. For the majority of parallel programs, the output data of some concurrent process is required as input data to some other concurrent process. This gives rise to dependences between computations. In the ideal case, concurrent computations are independent of each other. In many realistic applications, however, communication between the processes is required.

Parallel computing models guide the parallel algorithm design, algorithm partitioning, allocation of the partitioned computations to processors, and the resulting communication requirements. Parallel computing models also affect the specification of concurrent computations as well as that of communication operations. There is an abundance of parallel computing models; over 50 are surveyed in [1]. Parallel programming models differ in various ways. Moreover, the models abstract parallel computing and programming at different levels, some are more architecturally oriented, others, theoretical. The survey in [1] uses six criteria to classify these models; four of which "... relate to the need to use the model as a target for software development." The classification describes the nature of the specifications for partitioning, allocation, communications, etc.

Programming through parallel models describes the viewpoint adopted in this paper. The writing of a parallel program is guided by at least one model, and may be guided by several. Where more than one parallel model is involved, often, the models combine vertically, that is, provide for a lower-level to higher-level abstraction continuum.

The focus of this paper is to present a model under development that describes both specification metrics and performance metrics for parallel programs in the context of multiple parallel models. A specification in this paper refers to program code as described by a parallel model. Performance refers to the run-time operation of that code. The intent is to model informative information about parallel programs in terms of the constituent parallel models.

This paper is organized as follows. A model for specification and performance metrics for parallel programming is developed in Section II. Applications of the model together with preliminary results are presented in Section III. Some closing comments about the proposed model is discussed in

Section IV. Conclusions are given in Section V.

II. A Specification and Performance Model

In other work [2], [3], a relational abstraction model is described wherein each statement in a program (parallel or otherwise) has an associated operational and descriptive semantics together with a property set. Formally, a Level 1 relation is defined as $R^1 = (o, d, p)$ where o denotes a program statement, d represents semantic information about o and p is a property set associated with o. A hierarchy of relations is defined. In the hierarchy, a higher level relation $R^l = (R_j^{l-i_j} | j \geq 1, 1 \geq i \geq (l-1), d, p)$ where $l > 1$ relates one or more lower-level relations. Such higher level relations therefore abstract groups of relations, that is, abstract code fragments. These higher level relations also consist of semantic information and property sets where both are abstractions over all of the lower level participating relations. The processes to accomplish this are described elsewhere and are left out of this paper.

Definition 1: A *specification* is the functional or declarative abstraction of a code fragment written in some computer language.

Our notion of specification differs from that used in some areas of software engineering, for example, in [4] a formal specification is defined as a part of formal methods wherein the specification is compared with the program code to determine that the actual implementation meets the specification. Our definition supports the context of abstractions provided by parallel models. Definition 1 interpreted in the context of the afore-mentioned relational hierarchy leads to the subsequent definition.

Definition 2: A specification is represented by some R_i^l in a defined relational hierarchy.

The executions of specifications lead to run-time effects. An analysis of these run-time effects includes the analysis of the properties of a running program [5]. To a significant extent, the specification and its run-time effects must correspond. Thus, $R^1(o, (d_s, d_r), (p_s, p_r))$ where d_s and p_s refer to the specification of o and d_r and p_r refer to the run-time effects of o.

The role of a specification metric is to provide a degree-of classification of the abstraction described by the specification. Qualitative metrics classify a given program statement, for example, a *computation specification* is the abstraction of a statement or code fragment that belongs to the well-known notion of computations whereas a *communication* specification is the abstraction that belongs to the well-known notion of information transmission. Quantitative metrics identify estimated or measured 'belongness' to a category, for example, 50% computation.

Definition 3: A *specification metric* is a qualitative or quantitative measurement of a specification: $m^s = M^s(s)$ where M^s is a measurement method or technique (abbrev. $m^s(s)$).

Definition 4: A *performance metric* is a measurement of the run-time effect(s) of a specification: $m^p = M^p(e(s|r))$ where M^p is a measurement method or technique, e is a particular run-time effect of s given run-time parameter(s) r (abbrev. $m^p(s)$).

The correspondence between specifications and its run-time effects is modeled as a constraint on the specification and performance metrics applied to the same statement.

Definition 5: Congruence is the reasonable and meaningful association of a specification metric with a performance metric of the same specification: $m_i^s(k) \cong m_j^p(k)$.

The relations of interest, $R(o, (d_s, d_r), (p_s, p_r))$, can now be qualified and quantified: p_s is given by $m_i^s(o)$ within the role established by d_s, p_r is given by $m_j^p(o)$: $R(o, (d_s, d_r), (m_i^s(o), m_j^p(o)))$. Congruence implies that d_s and d_r are reasonably and meaningfully consistent. To accomplish this, we construct d_r such that $d_r = d_s||e||r$ where $||$ in this expression refers to the catenation of d_s, e and r.

Congruence establishes meaningful association so that a relation represents consistent information. However, congruence does not imply sameness of the metrics. A specification metric is directly tied to the abstraction(s) that a particular parallel model provides; it is useful when considering the *programming* component. A performance metric is directly tied to the execution of a program statement or fragment; it is useful when considering the systems performance component.

III. APPLICATIONS

This section considers the application of the above model to C/MPI parallel programs. The computational algorithm is coded in the C language. Concurrent tasks are defined during the program's development, guided in part, by the Single Program Multiple Data (SPMD) and the processor farm (PF) parallel models. Typically, each task is allocated to a distinct processor. The Message-Passing Interface (MPI) provides library support for communication functions between concurrent tasks. In our work, we consider that C is a language level model, MPI is a library level model and both SPMD and PF are programming level models. The example application concentrates primarily on the MPI level for purposes of illustrating the specification and performance metrics model previously proposed.

Although the MPI library supports many functions, only six are required at a minimum to complete a data transfer. The `MPI_Send` and `MPI_Recv` library functions support, respectively, transmission and reception. The other four provide house keeping functions, e.g., initialization. These six are referred to in this paper as the six elementary functions of MPI. Also important to our discussion is one other function not usually considered in the minimum six, but never-the-less, frequently used. The `MPI_Reduce` function combines reception from multiple sources with a reduction operation performed over all the values received. Usually, the reduction is the additional operation.

The send and receive functions are specifications that wholly classify its operation as a communication, whereas, the reduce function is a specification for both a communication and a computation. These three functions do not specify any house keeping. The four house keeping functions, on the other hand, specify house keeping but neither computation nor communication. The roles of specification in this application therefore are computation, communication and house keeping. We concentrate on the first two roles only and assume computation and communication are mutually exclusive.

Let m_t^p denote communication time, m_d^p denote data rate, and m_u^p denote the out-going port utilization. Let s denote the specification `MPI_Send` (in the following, we assume that `MPI_Recv` has the same metrics as `MPI_Send`) and t denote the specification `MPI_Reduce`. We are interested in determining:

$$
\begin{array}{ll}
m^s(s) = M^s(s) & m_t^p(s) = M_t^p(t(s|r)) \\
 & m_d^p(s) = M_d^p(d(s|r)) \\
 & m_u^p(s) = M_u^p(u(s|r)) \\
m^s(t) = M^s(t) & m_t^p(t) = M_t^p(t(t|r)) \\
 & m_d^p(t) = M_d^p(d(t|r)) \\
 & m_u^p(t) = M_u^p(u(t|r))
\end{array}
$$

We first describe M^s followed by $m^s(s)$ and $m^s(t)$. Next we describe the set of techniques for M^p and subsequently quantify the performance metrics.

Recall that M^s describes the inclusion of the statement's semantics in terms of the three categories: computation, communication and house-keeping. In general, $m^s = x$Cp + yCm, where x and y describe the percentages of inclusion into the categories of Computation (Cp) and Communication (Cm) (recall, in this paper, we do not consider the house-keeping category). We define a *baseline* program that consists of the minimal MPI program statements needed in which to implement the specification of interest. The baseline program will consist, maximally, of combinations of the six elementary functions. Let f_I be a code fragment of the input MPI program to parse and let f_b be a code fragment of the baseline program. The specification metric is quantified as follows: compute the code difference, D^s between the two code fragments: $D^s(f_b, f_I)$ (appropriately structured, this could be implemented via the `diff` unix utility). The intent is to determine the numbers and types of the six elementary functions needed to replace f_I, in so doing, establish the parameters x and y. Therefore, f_b is constructed based on the need for replacing the operations in f_I.

The specification metrics can now be quantified. First, $m^s(s)$ is trivial, since, f_I consists of the single statement corresponding to `MPI_Send`, and, f_b is constructed so that it also contains a single statement of the same. Hence, the difference is nil, and since, the f_b code fragment is already determined to be 100% communication, so too is f_b. The purpose of including this discussion is to illustrate the procedure of M^s. The quantification of $m^s(t)$ is more involved since `MPI_Reduce` has

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Fig. 1. Input code fragment, f_I.

```
{
int i;
double r;
MPI_Status stat;
pi = mypi;
if (myid==0)
  for (i=0; i<numprocs-1; i++) {
    MPI_Recv(&r, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &stat);
    pi += r;
  }
else
  MPI_Send(&pi, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
}
```

Fig. 2. Baseline code fragment, f_b.

semantics corresponding to communication as well as computation. Figure 1 shows f_I. The fragment f_b, Figure 2, is constructed to included multiple `MPI_Recv` calls followed by a computational component to reduce the communicated results. If there are N processes(ors) (represented by the variable `numprocs` in the figure), there are $N-1$ `MPI_Send` – `MPI_Recv` pairs of communications and $N-1$ loop iterations in f_b. Hence, $x = y = 50$, that is, 50% computation and 50% communication. We point out that $m^s(t)$ for *this program fragment* does not necessarily generalize for *all* uses of `MPI_Reduce`.

We now consider the performance metrics. The experiments were run on a four node dual-processor 900 MHz HP Itanium-2 workstation cluster interconnected via a 1Gbps HP ProCurve Switch 6108[1]. The MPI version is MPICH vs. 1.2.5 and the operating system is Red Hat Linux kernel 2.4.18-e.25smp. The experiments were both formally and informally conducted at various times over a two week period; the latter was used to verify the consistency of the results. There are a number of issues involved in performance modeling [6]–[8]. For purposes of this paper, only a limited study is performed.

Recall that we wish to determine the communication time or wall-clock time, data rates and out-going port utilization for the execution of MPI sends, receives and reduces. Two aspects are of interest. First, the *system's performance*, that is, what is the cost to the system?. Second, the *specification's performance*, that is, what is the cost of the specification? For the latter, we again use a baseline measurement approach, specifically, we define the fastest communication component (hence, the one with the least time per unit) as TCP socket layer, f_b, and measure the cost difference $D^p(f_b, f_I)$.

The measurement techniques vary slightly between the three metrics. In all cases, the `MPI_Wtime` function is used to measure the elapsed wall-clock time. The data rates are calculated based on the time and message size. The timings for socket communications are obtained from in-house developed code. The code used to time for `MPI_Send` communications is obtained from [6]. Time measurements of `MPI_Reduce` used modified code available from [9]. Port utilization is measured by a network monitor available on the switch.

Multiple iterations of communications with long messages were timed for socket, `MPI_Send` and `MPI_Reduce` communications. The total amount communicated varied between 125 MB and 375 MB. In terms of MPI, this will utilize the rendezvous feature of the communications. A linear regression analysis was performed on the measured times for all three cases, the lowest r^2 statistic is 0.96 corresponding with the socket time measurements. The data rates were calculated from the respective regression curves and plotted in Figure 3. As expected, socket layer communication had the fastest data rates, the `MPI_Send` was slower and the `MPI_Reduce` was the slowest. A further measurement was conducted to estimate the elapsed wall-clock time of the reduction operation — the addition in this case. Our technique here measures multiple iterations consisting of a single addition over an array (i.e., with similar array accesses that would likely be occurring during the `MPI_Reduce` operation) and measures the base time of multiple empty iterations; the difference establishes the amount of time required to perform the reduction operation during the `MPI_Reduce` operation. Conversion to data rates for the communication-only component of the `MPI_Reduce` measurements follows the previous procedure. The fourth curve in the figure, located just below the `MPI_Send` curve, reports the data rates for the communication-only adjustment of the `MPI_Reduce`. The data rates are very close to the send operation, suggesting that

[1]The system was obtained under the Hewlett-Packard Company Advanced Technology Platforms - Itanium 2 2003 Academic Grant Initiative. P.I.: Brian J. d'Auriol.

TABLE I

NOMINAL DATA RATES (MBPS)

Socket	MPI_Send	Adjusted MPI_Reduce	MPI_Reduce
641.6919	417.6212	403.7618	184.4455

TABLE II

NOMINAL TIMES (S): $y = mx + b$

	Socket	MPI_Send	Adjusted MPI_Reduce	MPI_Reduce
m	0.012699	0.019124	0.019738	0.043298
y	-0.070156	0.006101	0.015383	0.015383

similar communication mechanisms are used. These curves are near-horizontal.

Nominal values for the system's performance metrics can now be determined. For data rates, the average across all points in the respective curve is reported in Table I. For time, the linear regression curves are reported in Table II. For out-going port utilization, the measured rates are reported in Table III.

Specification performance metrics can now be determined. Table IV gives the specification performance metrics for data rates of MPI_Send and the communication component of MPI_Reduce. For MPI_Reduce, measured times for the communication and computation components establish: 45.66% communication and 54.34% computation.

IV. DISCUSSION

The specification metrics may guide the performance analysis. In the application example, $m^s(\texttt{MPI\_Reduce}) = 0.5\text{Cp} + 0.5\text{Cm}$ indicated a computation and communication component in the program. In turn, this guided a performance study to establish the computation and communication performance metrics. Although the two types of metrics are congruent, there is, in general, no requirement that they should be the same, or even, approximately the same. The two metrics measure different aspects of the parallel program.

TABLE III

MEASURED OUT-GOING PORT UTILIZATION (%)

Socket	MPI_Send	Adjusted MPI_Reduce	MPI_Reduce
70.6	23.8	20.0	20

TABLE IV

DATA RATE SPECIFICATION PERFORMANCE METRICS (MBPS)

MPI_Send	Adjusted MPI_Reduce	MPI_Reduce
224.0707	237.9301	457.2464

A parallel program may be written either along the lines of Figure 1 or Figure 2. Many would consider that the single MPI_Reduce function is conceptually easier to code as compared with the alternative. In the context of programming through parallel models, Figure 1 represents a single statement which has computation and communication semantics derived from the MPI parallel model. By contrast, Figure 2 represents a code fragment of many statements with various computation and communication semantics derived from the C language, the MPI and the SPMD parallel models.

The specification metrics provide a method to measure the induced semantics. In the application example, only MPI_Send, MPI_Recv and MPI_Reduce were measured; all of these derive semantics singly from the MPI parallel model. This procedure may be generalized. A single statement may have single or multiple semantics derived from a single parallel model or from multiple models. A set of semantic attributes derived from the parallel model(s) can be assigned to the statement (in the example, the two-vector of computation and communication semantics). Specification metrics may then be quantified to measure more precisely the nature of the statement.

The relational hierarchy, described in Section II provides suitable abstractions that allow both specification and performance metrics, measured against single statements, to abstract code fragments [2], [3]. In this case, the specification metrics applied to a code fragment indicate the nature and purpose of the fragment with respect to the parallel models that guided the development of that fragment.

V. CONCLUSIONS

This paper describes a model for specification and performance metrics of parallel programs. Semantic and property information is provided by one or more parallel program models that are used to guide the

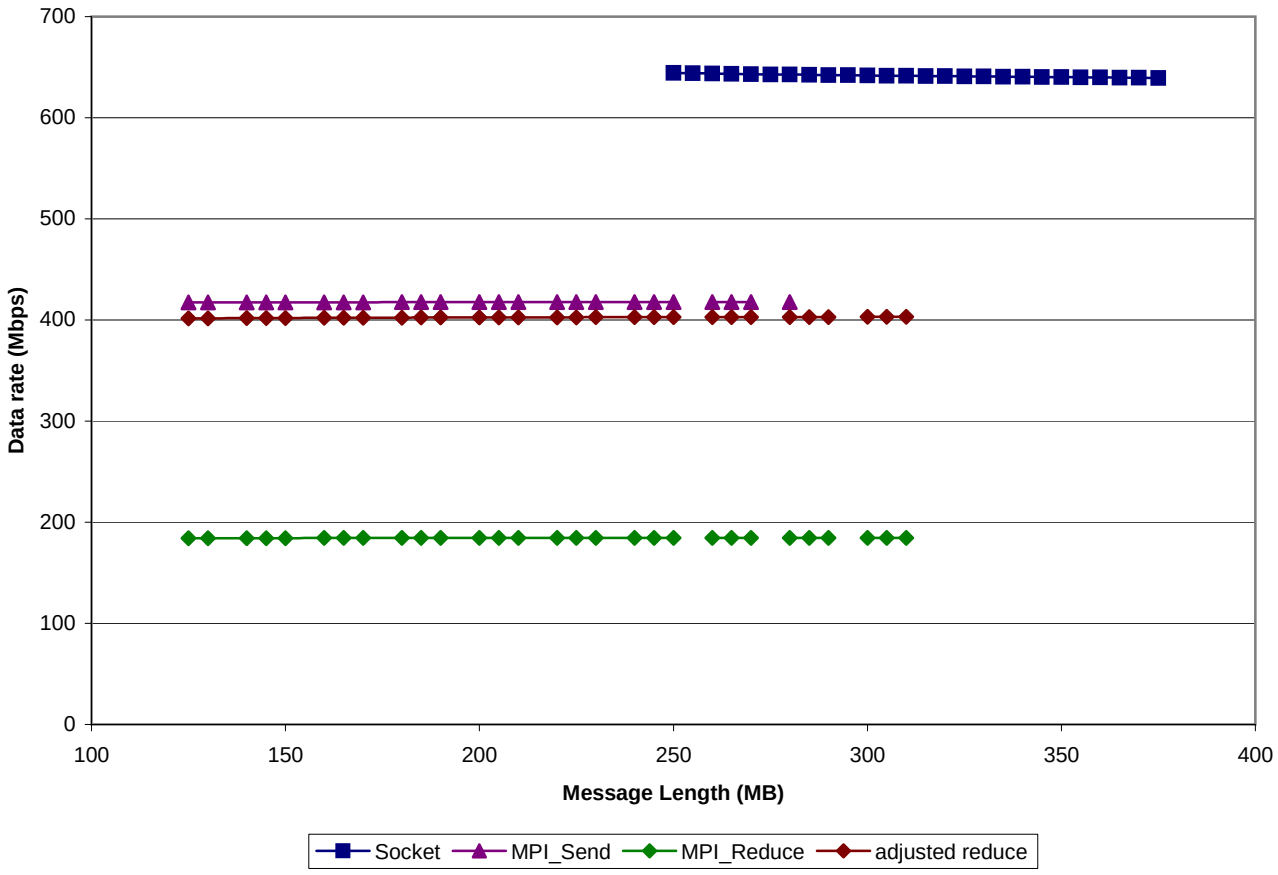

Fig. 3. MPI performance metrics (time)

development of the program. A specification refers to a program statement or a code fragment in a program. A specification metric qualifies and quantifies the abstraction of the statement or fragment as given by a parallel program model. A corresponding performance metric describes the run-time effects of that specification.

An application of the specification and performance metric model to a C/MPI program is also described. The application considers specifics of the `MPI_Send`, `MPI_Recv` and `MPI_Reduce` function calls. Specification and performance metrics for these functions are measured and discussed. Congruence between the two types of metrics is shown by the analysis. A generalization of the model is also briefly described.

Currently, the specification and performance metric model is incorporated into the Advanced Relation Model for Program Visualization [2]. The visualization model uses the relational hierarchy to provide visual presentations of these metrics together with semantic information about the program code.

REFERENCES

[1] D. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys*, vol. 30, no. 2, pp. 123–169, June 1998.

[2] B. J. d'Auriol, "Advanced relation model for program visualization (ARM 4 PV)," in *Proc. of the 2004 International Conference on Modeling, Simulation & Visualization (MSV'04) and Proc. of the 2004 International Conference on Algorithmic Mathematics & Computer Science (AMCS'04),*, H. R. Arabnia and et. al, Eds. Monte Carlo Resort, Las Vegas, NV, USA: CSREA Press, June 2004, pp. 489–493.

[3] ——, "A concept visualization study of a parallel computing program," in *Proceedings of the 2004 Internatioal Conference on Parallel Processing Workshops (ICPP Workshops),*, Y. Yang, Ed. Montreal, Canada: IEEE Computer Society, August 2004, pp. 239–246.

[4] A. Diller, *Z An Introduction to Formal Methods*. John Wiley & Sons Ltd., 1990.

[5] T. Ball, "The concept of dynamic analysis," in *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1999, pp. 216–234, lNCS 1687.

[6] *MPI Performance Topics*, Lawrence Livermore National Laboratory (LLNL), May 2004, UCRL-MI-133316. [Online]. Available: http://www.llnl.gov/computing/tutorials/mpi_performance/

[7] W. Gropp and E. Lusk, *Tuning MPI Applications for Peak Performance*. [Online]. Available: http://www-unix.mcs.anl.gov/mpi/tutorials/perf/

[8] G. R. Luecke, J. Yuan, S. Spanoyannis, and M. Kraeva, "Performance and scalability of MPI on PC clusters," *Concurrency and Computation: Practice and Experience*, vol. 16, pp. 79–107, Jan. 2004, (also,Performance Evaluation & Modeling of Computer Systems, November, 2002.

[9] W. Gropp and E. Lusk, *Tuning MPI Applications for Peak Performance*, (Tutorial examples). [Online]. Available: http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl/src3/barrier/C/main.h%tml