The Multigraph Modeling Tool

Amy Apon
Clemson University, aapon@clemson.edu

C A. Childers
Vanderbilt University

W H. Hooper
Belmont University

K D. Gordon
Vanderbilt University

L W. Dowdy
Vanderbilt University

The Multigraph Modeling Tool[*]

C.A. Childers, A. W. Apon, W. H. Hooper[†], K. D. Gordon, L. W. Dowdy
Department of Computer Science
Vanderbilt University
Nashville, TN 37253
[†]Department of Mathematics/Computer Science
Belmont University
Nashville, TN 37212

Abstract

The Multigraph Modeling Tool (MMT) has been developed as a performance prediction tool for parallel applications executing on multicomputer systems. MMT is a program generator that accepts as input a system description (i.e., a parallel application and the hardware on which it executes) and from this description automatically generates an analytic model which can be used to predict the performance of the system. The solution of the analytic model results in standard performance metrics such as processor utilization and application response time. A change in a parameter of the system description results in MMT automatically generating a new analytic model. The different sets of metrics produced for a system by varying description parameters can be used by engineers to determine those parameters which result in the best performance. MMT has been applied to a network of RS6000 workstations and to an Intel Paragon.

1. Introduction

With the wide availability of high–powered computing resources, it is often the case that several hardware platforms are available to execute any given parallel application. Such platforms range from loosely–connected heterogeneous workstations connected via an Ethernet to more tightly coupled homogeneous processors connected via a mesh topology. Besides a variety of available hardware platforms, the software configuration, such as the assignment of processes to processors, can affect the performance of an application. With these wide ranges of hardware platforms and software configurations, it is not feasible to execute a parallel application under all possible scenarios for the purpose of determining which one results in the best performance for that application. Given that the performance of a parallel application can vary dramatically depending on the communication and computation patterns of the application and the hardware on which it executes, it is important to determine the hardware platform and software configuration that maximizes performance. The Multigraph Modeling Tool (MMT) has been developed for this purpose.

MMT accepts as input a high level description of a parallel application (i.e., the hardware platform and software configuration) via the Multigraph High-level Description Language (HDL) [1]. From this description MMT generates an analytic model in the form of a Generalized Stochastic Petri Net (GSPN)[4] which is solved for various performance metrics. Any change in the HDL of an application automatically results in the generation and solution of a new GSPN model which produces updated performance metrics.

The remainder of the paper is organized as follows. Section 2 gives an example of the use of MMT. Section 3 describes MMT's automatic generation of the GSPN model. Section 4 validates several models with two actual parallel applications on two different hardware platforms. Section 5 is the summary and conclusions.

2. Example

The purpose of MMT is to take as inputs 1) a description of an application in the form of a data flow graph, 2) a description of the intended hardware on which the application executes, and 3) the mapping between the two, and produce as outputs performance metrics for the particular system. As an example, consider the task graph and intended hardware platform (a processor mesh) shown in Figure 1. The task graph is a simple fork–join application shown in Figure 1(a). The data flow graph shown in Figure 1(b) consists of *actor nodes* A_i that are equivalent to the application's tasks in a task graph and *data nodes* D_i that serve as buffers of data written (read) by the tasks (actor nodes). For example, data node D_1 is written by actor node A_1 and read by actor nodes A_2 and A_3. In addition to the data flow graph, the application description includes the mean actor execution time, t_i, and the amount of data written to and read from each

[*]This work was partially supported by the US Air Force Arnold Engineering Development Center and by sub–contract 19X–SL131V from the Oak Ridge National Laboratory managed by Martin Marietta Energy Systems, Inc. for the U.S. Department of Energy under contract no. DE–AC05–84OR21400.

data node. In this example, assume that actor node A_1 takes 4 time units to complete, A_4 takes 8 time units to complete, and actor nodes A_2 and A_3 require 1 time unit each ($t_1 = 4$, $t_4 = 8$, $t_2 = t_3 = 1$). Also assume that A_1 writes 20 data units to D_1, A_2 and A_3 each read (write) 10 data units from D_1 (to D_2 and D_3), and A_4 reads 10 data units each from data nodes D_2 and D_3.

The hardware description (shown in Figure 1(c)) includes the link connections between processors, the scheduling policy at each processor, and the speed of the links. In this example, each processor employs a FCFS scheduling policy and all links transfer data at a rate of 1 data unit per unit time.

The software configuration (i.e., the mapping between the application and hardware descriptions) provides the processor assignments of the actor and data nodes. Also included in the software configuration is the assignment of communications between actor and data nodes to sets of physical links. For example, let the notation $link_{x-y}$ represent the link between processor x and processor y. If actor node A_1 is assigned to processor 1 and data node D_1 to processor 6, then the set of physical links assigned to the communication between the two might be $(link_{1-2}, link_{2-6})$. In this example, assume the following processor assignments: 1) A_1 and D_1 are assigned to processor 1, 2) A_2 and D_2 are assigned to processor 2, 3) A_3 and D_3 are assigned to processor 3, and 4) A_4 is assigned to processor 4. Communications between actor and data nodes residing on the same processor do not require the use of physical links and, therefore, are assumed to take negligible time. Specific parameters of this example are summarized in Figure 2.

When the above scenario is input to MMT, MMT generates a C program. The C program represents a GSPN model of the system. This model is solved by a GSPN solver, the Stochastic Petri Net Package[2] (SPNP), via the execution of the C program, and performance metrics are generated as output. The complete GSPN model in shown in Figure 3. It is composed of a Petri net compound (i.e., a collection of places and transitions) for every element of the system description each of which is distinguished by dashed lines in the figure. Several performance metrics produced by MMT for this example are shown in Table 1. The response times of actor nodes A_1, A_2, and A_3 are equal to their respective service demands. The response time of actor node A_4 is higher than its service demand since it is required to wait on data from both actor nodes A_2 and A_3 before it may begin execution. Given the high link utilizations and queue lengths, it

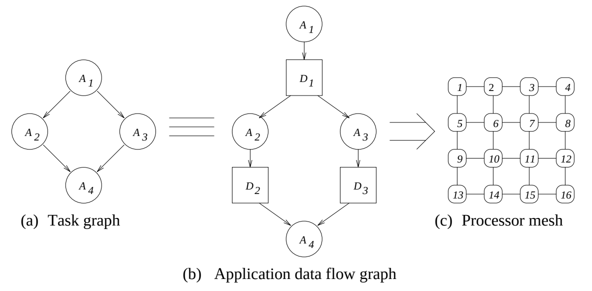

Figure 1: Example application task graph, data flow graph, and hardware platform

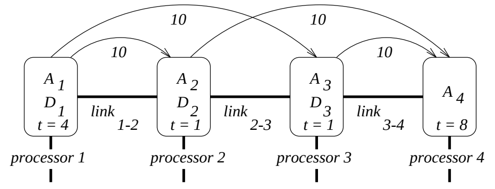

Figure 2: Pictorial representation of the example hardware and software configuration

is evident that communication is the bottleneck for this application on the specified hardware platform and software configuration. From this information, one might: 1) choose another software configuration (i.e., a different process to processor assignment to reduce link contention), 2) execute the parallel application on more processors in order to reduce the size of each communication, 3) execute the application on fewer processors to take advantage of excess processor capacity and reduce total interprocessor communication, or 4) speed up the interprocessor communication by changing the communication protocol or link speeds.

3. MMT's GSPN Model

3.1 MMT Inputs

The Multigraph HDL is used to describe the system inputs to MMT and consists of a hardware de-

	A_1	A_2, A_3	A_4
Response time	4.000	1.000	10.954
	Processor 1	Processors 2, 3	Processor 4
Utilization	0.086	0.021	0.171
	$link_{1-2}$	$link_{2-3}$	$link_{3-4}$
Utilization	0.428	0.428	0.423
Queue length	0.642	0.685	0.675

Table 1: Example MMT performance metrics

scription, an application description, and a software configuration. The hardware description consists of: 1) a list of all processors and their queueing disciplines (e.g., FCFS, PS), 2) a list of all communication links, their capacities, and their type (e.g., Ethernet, store–and–forward, virtual circuit), and 3) the topology of the hardware platform (i.e., which links connect which processors). The application description is in the form of a data flow graph and consists of: 1) a list of all actor nodes and their service time distributions (i.e., their mean execution times and their execution time variances), 2) a list of all data nodes and their capacities, and 3) the topology of the actor and data nodes (i.e., which actor nodes read from (write to) which data nodes). The software configuration specifies the mapping between the hardware and software descriptions and consists of: 1) the assignment of actor nodes and data nodes to specific processors and 2) the assignment of actor/data node communications to specific physical links.

3.2 MMT Detailed GSPN Model

MMT automatically generates a GSPN model by using generic Petri net compounds for each element of the system description. These general Petri net compounds are templates. The templates are filled in and joined together with the information specified in a system description resulting in a complete GSPN model. In this section, several MMT Petri net compound templates are presented.

The Petri net compound template used to model the processors and links in the hardware description is made up of one place with input and output arcs. The origin (destination) of the input (output) arcs is determined by the parameters input to the software configuration. For example, the Petri net compound template for a FCFS processor is shown in Figure 4. The input and output arcs for a given processor connect to actor node compounds for all actor nodes which execute on that processor. The token in the processor place represents an idle FCFS processor. The token flows from this place to an actor node compound when that actor node is currently executing on the processor. Thus, when the token is not present, the processor is being utilized by an actor node, and no other actor node may execute on the processor (i.e., none may capture the token). When an actor node completes execution, it returns the processor token via the input arcs from the actor node compound to the processor place.

In Figure 5, the Petri net compound template for an actor node is shown. The input arcs to places $ready_i$ represent input from the supplying data nodes. These arcs are determined by the connections from

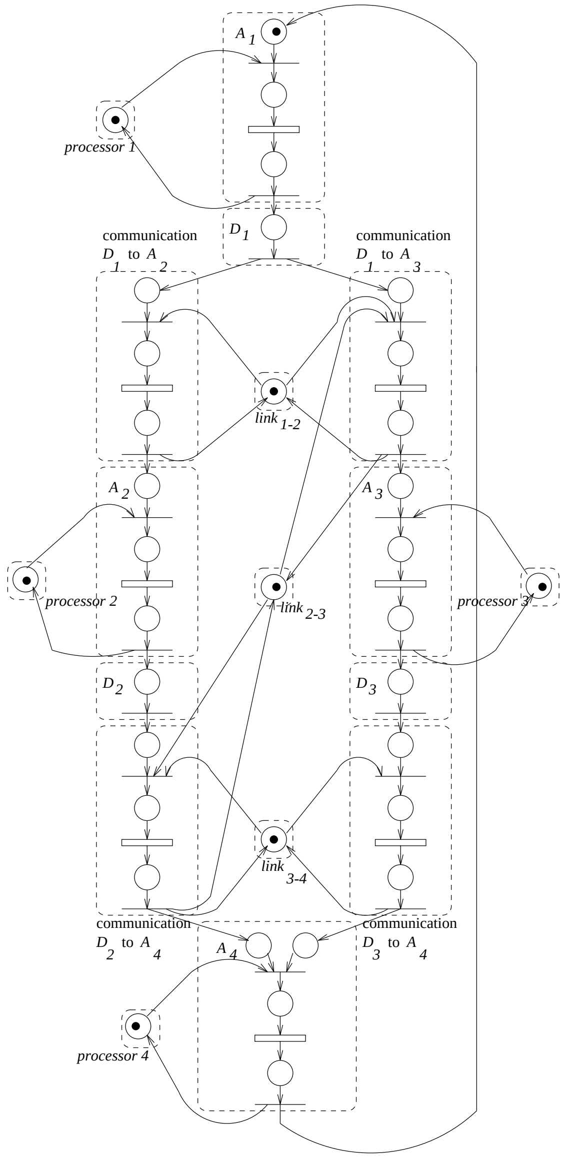

Figure 3: The complete GSPN model for the example

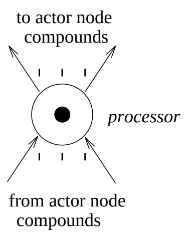

Figure 4: FCFS processor compound template

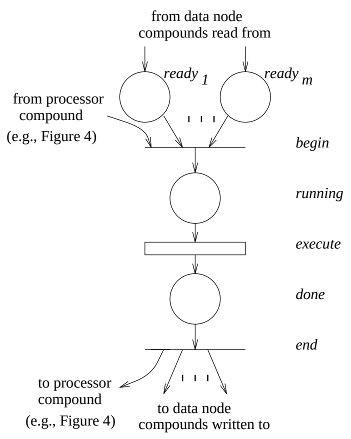

Figure 5: Actor node compound template

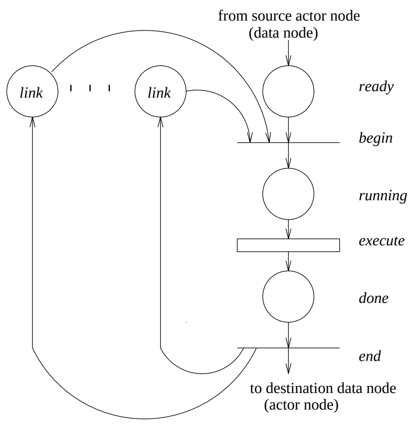

Figure 6: Virtual circuit compound template

data to actor node specified in the application description. When all of the $ready_i$ places contain a token, the actor is ready to begin execution. If a token is available in the place representing the processor on which this actor node is executing (e.g., Figure 4 template), the transition *begin* fires, placing a token in the place *running*. If the processor token is not available, another actor is currently utilizing the processor and this actor node is blocked. With a token in place *running*, the timed transition *execute* fires according to the specified service time distribution. When the *execute* transition fires, it places a token in place *done*. This token causes the transition *end* to fire which places a token in all data node compounds to which this actor node writes data. The firing of the *end* transition also places a token in the processor place on which this actor node is executing thereby freeing the processor.

A virtual circuit Petri net compound template is shown in Figure 6. A communication between an actor node and a data node (or vice versa) utilizing a virtual circuit must acquire all of the links in the path specified in the assignment description before the communication may begin. This is modeled by the input arcs from all the required links to the transition *begin*. A token in the place *ready* indicates that the source actor node is ready to send its data to the destination data node. Once all of the links are available, the *begin* immediate transition fires placing a token in the *running* place. The time spent waiting for the links models the virtual circuit set up time. A token in the *running* place enables the *execute* timed transition. This transition fires at a rate that is a function of the physical link capacity (specified in the hardware description) and the message size (specified in the application description). This transition represents the message traversing the links to its destination node. After the communication has completed, the links are released by the output arcs from transition *end* returning tokens to the link places.

3.3 MMT Outputs

The output metrics generated by MMT are 1) actor response time, 2) processor utilization, throughput, response time, and queue length, 3) communication throughput and response time, 4) link utilization and queue length, and 5) application throughput and response time.

4. Model Validation

In this section, MMT is used to predict the performance of two systems as input parameters are varied. The predicted performance is then compared to the actual measured performance of each system. These predictions assume no *a priori* knowledge of the applications other than their data flow graphs.

4.1 RS6000 Workstations

The first system consists of an image processing morphological filtering algorithm executed on several IBM RS6000 workstations connected by a 200 KB/second Ethernet. The data flow graph for this application is shown in Figure 7. Actor node S splits an image into n equal pieces, sending each piece to a separate data node, Da_i, $i = 1, ..., n$. After actor node A_i executes, it sends its output data to actor node M, via data node Db_i. Actor node M merges and displays the filtered image. Actor and data nodes within the same dashed lines execute on the same workstation. Communications between actor and data nodes on separate processors, such as actor node S and data node Da_2, take place across the Ethernet. Negligible communication time is assumed when actor and data

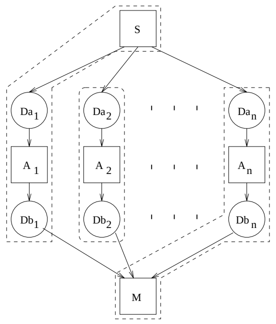

Figure 7: Image processing application data flow graph

nodes reside on the same processor (e.g., S and Da_1). Input to MMT for this algorithm along with experimental results are taken from[5]. Given P processors and an NxN pixel image, the service rate of actors S and M is $\frac{200000.0}{N^2+8N(P-1)}$ and the service rate of actor A_i is $\frac{P}{0.0002N^2}$. The message startup overhead is 415μs and the message size is $\left\lceil \frac{N^2}{P} \right\rceil + 8N$.

Predicted versus experimental results are graphed for two different image sizes in Figures 8 and 9. As N changes, the parameters varied are the service times of each actor node, the number of actor nodes, and the communication message sizes. The predictions accurately track the same behavior as the experimental results.

4.2 Intel Paragon

The second system is an LU decomposition application executing on an Intel Paragon XP/S 5. The Paragon is a 6 x 11 mesh of 66 processors connected by 200 MB/sec uni–directional links. Each processor has a link in the north, south, east, and west directions. The Paragon uses wormhole routing. Each communication is assumed to have its own single link virtual circuit. With n threads and a matrix of dimension m (i.e., an m x m matrix), each thread executes m loops in each of which $n-1$ messages are sent, one to every other thread. The mean and variance of the thread service time parameters are obtained by averaging 5 executions of the LU decomposition application on 1 processor. A linear speedup is assumed, giving the service time of each thread on n processors. For example, from measurement data, an LU decomposition of a 16 x 16 matrix executes on one processor in an average of 0.1378 seconds. This is the service time for the one actor node of this application. The assumed service time for the application executing on 2 processors is $\frac{0.1378}{2} = 0.0689$ seconds, representing the time for each of the 2 actors to execute m loops. The processor scheduling policy is assumed to be FCFS. The system parameters varied include actor node service times, the number of actors nodes (i.e., the number of processors over which the parallel application is forked), and the communication message sizes. Results are graphed in Figures 10 and 11 for matrices of dimensions 16 and 64, respectively. The figures indicate that the MMT predictions accurately track actual performance.

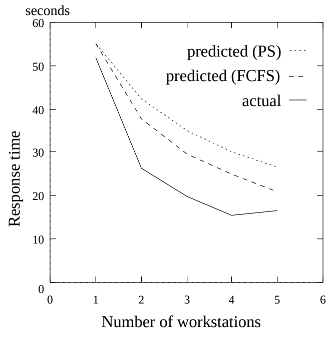

Figure 8: Response times for 512 x 512 image

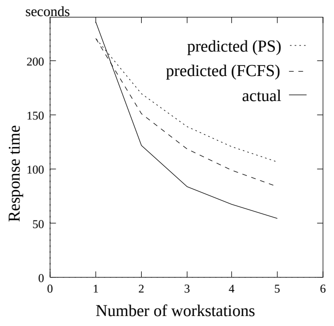

Figure 9: Response times for 1024 x 1024 image

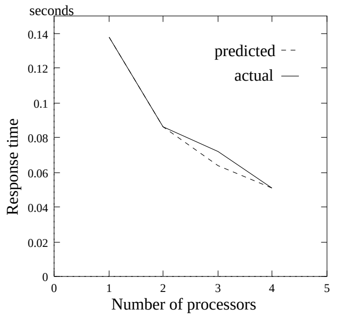

Figure 10: Response times for a 16 x 16 matrix

4.3 Model Extension

Both of the above case studies indicate that a portion of the predictive errors is due to the exponential service time distribution assumption. The high variance of an exponential distribution is not representative of the small variance measured in the actor node service times. To validate this observation, the previous GSPN models generated by MMT are solved with constant distributions via simulation for the actor node service times. The results of the simulations for the image processing application are shown in Figures 12 and 13. These are the same experimental results shown in Figures 8 and 9 in Section 4.1. The figures indicate that the constant service time assumption results in more accurate models for this application.

5. Conclusions

MMT is a program generator of GSPN performance predicting models. The automatic generation and solution of a GSPN model given a high–level description of the hardware and software makes it convenient for engineers to evaluate the performance of their applications on multiple configurations. In this paper, the predicted performance of two example systems is compared to actual measured performance. The predictions and measurements are in good agreement. MMT can be applied to address "What if ... ?" types of questions such as "What if the background workload on the processors increases by 50%?", "What if the communication links over which threads communicate are upgraded?", "What if the communication paradigms changed from store–and–forward to virtual circuit?", or "What if the thread placement for this application is changed?". By automatically generating GSPN models, MMT is useful in predicting performance across a large number of hardware platforms and software configurations.

Acknowledgements

We would like to thank Janos Sztipanovits, Gabor Karsai, Ben Abbott, Ted Bapty, and Michael Moore.

References

[1] B. Abbott, T. Bapty, C. Biegl, G. Karsai, and J. Sztipanovits, "Model–based software synthesis", *IEEE Software*, (May 1993), pp. 42 – 52.

[2] G. Ciardo, J. Muppala, and K.S. Trivedi, "SPNP: Stochastic Petri Net Package," Proc. of the 3rd Intl. Workshop on Petri Nets and Perf. Models, IEEE Press, (Dec, 1989), pp. 142–151.

[3] G. Karsai, "IPCS Report: Hierarchical Description Language (HDL) User's Manual", Vanderbilt University, Nashville, TN, January, 1993.

[4] M. Marsan, G. Conte, and G. Balbo, "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems", *ACM TOCS*, 2, (May 1984), pp. 93–122.

[5] M. Moore, "Programming environment for parallel image processing progress report 2", Vanderbilt University, Nashville, TN, March, 1993.

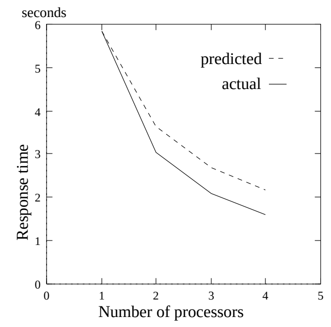

Figure 11: Response times for a 64 x 64 matrix

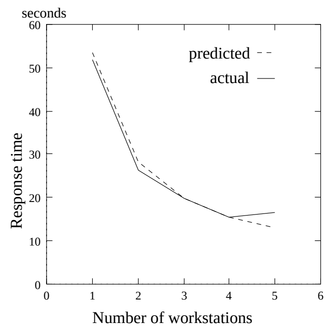

Figure 12: Response times for a 512 x 512 image

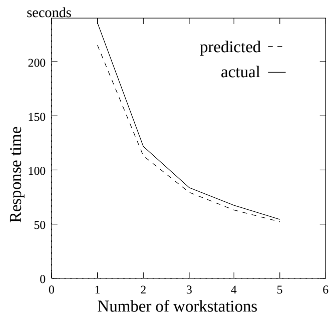

Figure 13: Response times for a 1024 x 1024 image