Estimating Execution Time For Parallel Tasks in Heterogeneous Processing (HP) Environment

Jaehyung Yang, Ashfaq Khokhar, Sohail Sheikh†, Arif Ghafoor

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

† Dept. of Electrical Engineering
Widener University
Chester, Pennsylvania 19013

Abstract

Mapping of tasks an application program onto a suite of heterogeneous machines requires estimation of execution times of the tasks on these machines. In this paper, an efficient methodology for estimating the execution times of a given program on various machines available in an HP environment is presented. The methodology uses parametric code profiling and parametric analytical benchmarking techniques and incorporates the concept of an architecture independent computation model to estimate the execution times.

1 Introduction

The concept of Heterogeneous Processing (HP) systems and their use in solving Grand Challenge problems [1] has been introduced recently [2, 3, 5, 8]. An HP system includes heterogeneous machines (including parallel machines), high-speed networks, interfaces, operating systems, communication protocols, and programming environment, all integrated together to produce a positive impact on ease of use and performance [5]. An efficient use of HP paradigm requires thorough understanding of the characteristics of applications, the architectures of machines, and their programming features.

One major challenge for future HP, therefore, would be to manage processing of applications by finding suitable matches between codes[1] of these applications and machines. For this purpose, code profiling is used to characterize applications and identify tasks which have the same computational behavior and to estimate their execution times. Very few code profiling methodologies within the context of HP have been proposed in literature. These methodologies have limitations in their applicability and mostly are based on a rather simplistic and highly abstract view of parallelism. For example in [2], code profiling is viewed as characterizing a given code in terms of embedded parallelism such as SIMD, MIMD, etc., without providing a mechanism for estimation of execution time. The detailed architectural knowledge of machines and computational requirements of applications are ignored in such methodologies.

New code profiling techniques are needed which can incorporate the detailed architectural characteristics of machines and realistic computing requirements of the applications. These techniques then can be used more accurately and intelligently in making scheduling and mapping decisions. Indeed, such decisions significantly impact the execution of applications. One possible methodology to estimate execution time of a task on various machines is to use Parallel Assessment Window System (PAWS) [6]. The approach introduced in PAWS is quite exhaustive and may not be viable for large applications and/or over large number of machines. Similar approach for serial machines, as discussed in in [7] has the same limitation.

We propose a general framework for estimating the execution time of application tasks. This is achieved by characterizing application programs and architectures of machines using parametric code profiling and parametric analytical benchmarking techniques, introduced in this paper. The parametric code profiling gathers information about various categories of operations performed in a given code. Similarly, parametric analytical benchmarking provides information about execution times of these operations on different machines. We show how a combination of these two

[1] We will use the terms code and task interchangeably in this paper. Code is the unit of estimation and the task is the unit of allocation to a machine.

This work was partially supported by grants from the Purdue Research Foundation and NSF Grant No. CDA-912771

0-8186-5592-5/94 $3.00 © 1994 IEEE

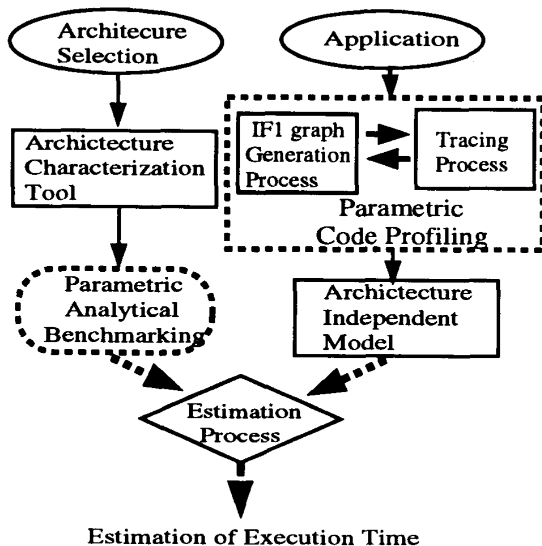

Figure 1: The proposed methodology for estimating execution time of an HP application.

approaches provides a systematic methodology for estimating the execution time of applications. The proposed technique is outlined in the next section.

1.1 The Proposed Approach

The approach proposed in this paper is a hybrid one and provides a more practical solution. It applies parametric analysis as in PAWS but avoids exhaustive computation by introducing high level analysis through an architecture independent model. The computations and communications performed in a code are characterized in terms of operations defined by a set of parameters. A given application code is analyzed in terms of computation and communication units using an architecture independent model (C^3) introduced in [4]. The input to the model is the total number of operations performed during each parallel step and the amount of communication performed in between parallel steps. This input is obtained from Intermediate Form 1 (IF1) [6] graph representation of the application code. Necessary modifications in the traditional IF1 graph representation are introduced to incorporate various features of parallel programming. The units charged are then translated into actual execution time using detailed characterization of the target machines. Such architecture characterization is obtained by measuring the processing time on target machines for each parameter in the parameter set (parametric analytical benchmarking).

A block diagram of the proposed methodology is given in Figure 1. There are three basic steps involved in this approach. First, the frequency of various operations employed in the input code is obtained. Then, computation and communication units are estimated using the architecture independent computational model introduced in [4]. Finally, the estimation of execution times on various machines is obtained by using the benchmark results of the units.

The remainder of this paper is organized as follows. Section 2 describes the parametric code profiling technique and briefly outlines the architecture independent model. Section 3 presents the parametric analytical benchmarking technique and discusses estimation process using the results obtained in the code profiling step. The proposed methodology is elaborated by an example in Section 3.2. The conclusions are given in Section 4.

2 Parametric Code Profiling

An application program is considered to be composed of tasks and is represented by a Task Flow Graph (TFG) or a Task Interaction Graph (TIG). A task is an autonomous unit of a given code[2] and can be allocated to a machine. In order to analyze an application task, we assume that a task can be partitioned into a sequence of superstep, with each superstep corresponding to local computation followed by sending and receiving (reading or writing) messages.

For the parametric code profiling, we need to define a set of parameters where each parameter represents a distinct category of low level operations performed in a task. This set should be abstracted in such a way that it can encapsulate architecture dependent factors. Also, it should be complete in the sense that all the operations in a task are represented in this set. These parameters are functionally partitioned into four categories, namely; computation, data movement and communication, input/output, and control. Each category is recursively partitioned into sub-categories upto a desired level of details. This information can be organized in a hierarchical form [6].

A task is represented with its IF1 graph. Such a representation provides architecture independent description of the task. An IF1 graph is an acyclic graphical language for dataflow graph representation. This

[2]The input code is assumed to be written in a parallel language

intermediate form supports data dependency and parallelism at all levels of granularity. The amount of local computation performed in each processor is represented by a compound node in the IF1 graph. An edge between two such nodes represents communication among the corresponding processing nodes. The construction of an IF1 graph requires decomposition of expressions in a parallel code into IF1 operation nodes. Operands, data values, and intermediate results are transformed into IF1 edges.

Subsequently, the IF1 graphs can be processed to produce parametric code profiling vector of parameters present in the input code. As this process is independent of the run time data, the worst case behavior is chosen for conditional branch statements. There can be other approaches such as test run, trace generation, etc. however, we do not consider these issues in this paper.

The boundary of IF1 graph is represented by a *boundary node*. Input and output ports of this node specify amount of communication. The input ports represent the amount of data received, while the output ports represent the amount of data sent. The total amount of communications depends on the number of processors employed, since the size of input/output data is determined by the input size assigned to a processor. Therefore, the amount must be expressed in association with the input size, n, and the number of processors, p. For the purpose of analysis, the values of parameters such as the maximum amount of data communicated between any pair of processors and the average amount of data communicated between processors can also be obtained. These are important factors that affect the congestion in the network.

Formally, the computation code profiling can be described as a Parametric Code Profiling Vector(PCPV), $\vec{V}_t$, of size size N for a task t. This vector is a summation of sequential parameter vector $\vec{V}_t^s$ and parallel parameter vector $\vec{V}_t^p$. This vector describes the counts of units for each parameter identified in a task t. Each element v_i of $\vec{V}_t$ represents the operation count for parameter i and N is the cardinality of the parameter set. The value of V_t^p depends on the size of the input and the size of the target machine.

The analysis of the parallel part is carried out based on the architecture independent model described in [4]. For the sake of completeness, the model is briefly outlined in the following section.

2.1 An Architecture Independent Model

Suppose that a given task is being performed on n data items. We assume that the n data items are distributed among p processors so that each processor receives at most $\lceil n/p \rceil$ inputs. We refer to this as a balanced input distribution. For many applications a balanced input distribution is the most natural one. However, if the given code provides a different distribution, the execution time is dictated by the heaviest processing node.

The model captures the complexity of *computation*, the pattern of *communication*, and the potential *congestion* during communication. A metric is defined for estimating the effect of link and processor congestion on the performance of a communication operation. This metric allows the evaluation of communication operations without having to specify fine scheduling details, a feature desired by application programmers. In this model, the performance of a superstep is expressed in terms of *computation units* and *communication units*. The number of computation units charged depends on the amount of local computation done. The number of communication units charged depends on the amount of data sent by a processor, the amount of data received by a processor, the latency encountered by the messages, and the congestion arising due to the volume of inter-processor communication. The routing schemas and protocols available on a machine also influence the performance and thus the number of communication units charged.

The parameters of the parallel machine used for determining computation and communication units are the following. Let p be the number of processors available in the machine. A message is the logical unit for communication between two processors and is made up of fixed-length packets. We use l to denote the length of a packet (measured in bytes), s to denote the set-up time to execute a send, and h to denote the latency. We use the average distance between two processors to represent the latency. The average distance is defined as $(\sum_{1 \le i,j \le p} d_{i,j})/p^2$, where $d_{i,j}$ is the minimum distance between processor i and j. For SIMD machines, $s, l = 1$.

The charging of computation units in a superstep is done as follows. Assume that in one superstep, processor i performs t_j bytes for executing j-type operations. The superstep is charged $\max_{1 \le i \le p} \sum_{1 \le j \le N} t_j$ computation units, where N is the number of various types operations defined by the parametric code profiling.

The communication units charged to one superstep reflect the time spent on sending messages, the time spent on receiving messages, the time messages are enrouted under ideal conditions, the amount of congestion that could occur, and an estimate on the re-

sulting delay. It is assumed that a processor cannot send and receive simultaneously. Consider a one-to-all communication being performed in a superstep. That is processor i sends L distinct bytes to each processor in the processor array. The cost of sending the message from i (without congestion), S_i, would be the total send time for processor i. Let R_j be the total receive time at processor j, $1 \leq i \leq p$ $(j \neq i)$. The total cost of the communication operation is then given as:

$$S_i + \max_{\substack{1 \leq j \leq p \\ j \neq i}} R_j + \text{delay due to potential congestion}$$

$$= s(p-1) + h + \lceil \frac{L}{l} \rceil \star (p + \frac{p-1}{b} + h)$$

Additional details on the charging method related to this model can be found in [4].

3 Parametric Analytical Benchmarking for Execution Time Estimation

For an accurate estimation of the execution time of a task, individual timing information of a single unit of each parameter on every machines in HP is required. The proposed parametric analytical benchmarking provides such information using the architecture characterization tool of PAWS. This tool uses an integrated hierarchical approach based upon low level benchmarking that completely determines the raw timing information of operations and behavior of the machine for each category of parameters. In addition to the timing information of computational parameters, parametric analytical benchmarking should provide the architectural information which affects the performance of communication.

The parametric characterization of a machine m constitutes a Parametric Computation Benchmarking Vector, B^m, whose size is the same as the cardinality of parameter set, N. Formally, this vector is described as follows:

$$\vec{B}^m = [b_{m1}, b_{m2}, ..., b_{mN}]$$

where b_{pj} represents the performance of machine m with respect to parameter j. Given M machines in an HP system, a collection of M such vectors, constitutes a Parametric Computation Benchmarking Matrix (PCBM) which is given as follows.

$$\mathbf{B} = \begin{bmatrix} \vec{B}^1 \\ \vdots \\ \vec{B}^M \end{bmatrix} = \begin{bmatrix} b_{11} & \cdots & b_{1N} \\ \vdots & \cdots & \vdots \\ b_{M1} & \cdots & b_{MN} \end{bmatrix}$$

The benchmarking of machines for communication operations obtain timing information about the parameters that determines the performance of communication. Such parameters include the number of processors (p), data transfer time for one packet (r), the setup time (s), the latency (h), and the bisection width (b). Parameters such as the number of processors, the latency, and the bisection width are readily available from the specification of the architecture. However, the setup time and data transfer time vary based on the machine configuration.

3.1 Estimation Process

Estimated Execution Time Vector, $\vec{E}_t$, for a task t is used to describe both computation and communication operations on various machines. Computation estimation is obtained by using parametric code profiling vector as a workload distribution and PCBM as weighting function. Communication estimation requires separate function depending on the type of communication protocols employed in a machine.

For a given task t, $\vec{E}_t$ is the summation of computation estimation vector $\vec{E}_t^{comp}$ and communication estimation vector $\vec{E}_t^{comm}$.

$$\vec{E}_t = \begin{bmatrix} e_1 \\ \vdots \\ e_M \end{bmatrix} = \begin{bmatrix} e_1^{comp} \\ \vdots \\ e_M^{comp} \end{bmatrix} + \begin{bmatrix} e_1^{comm} \\ \vdots \\ e_M^{comm} \end{bmatrix},$$

where M represents the number of different machines.

The element e_m represents the estimated execution time of the task t on machine m. The estimation of computation time on machine m, e_m^{comp} is obtained by adding all partial estimation for each computation parameter constituting the task.

$$e_m^{comp} = \sum_{i=1}^{N} (\text{execution time for } v_i \text{ operations of parameter } i \text{ on machine } m)$$

$$= \begin{bmatrix} b_{m1} & \cdots & b_{mN} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ \vdots \\ v_N \end{bmatrix}$$

$$\vec{E}_t^{comp} = \begin{bmatrix} b_{11} & \cdots & b_{1N} \\ \vdots & & \\ b_{M1} & \cdots & b_{MN} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ \vdots \\ v_N \end{bmatrix} = \mathbf{B} \cdot \vec{V}_t$$

Similarly, communication estimation vector, $\vec{E}_t^{comm} = [e_1^{comm}, \ldots, e_M^{comm}]^T$, can be generated by obtaining each e_m^{comm} on machine m as follows.

$$e_m^{comm} = \max_{1 \leq i \leq p} (\textit{sending time}_i + \textit{receiving time}_i) + \textit{congestion}$$

The example of communication estimation for one to one routing on a p processor mesh is described below. In one-to-one routing, also known as permutation routing, assume that every processor sends L packets to a unique destination (i.e., unique among all p processors). The total number of processor pairs in communication is p.

In a p processor square mesh, bisection width = average latency = $\mathcal{O}(\sqrt{p})$. Then, node and link congestion are balanced and we the estimation of communication is

$$e_m^{comm} = f_{one-one}(L) = s + \sqrt{p} + L \star r \star (2 + 2\sqrt{p})$$

3.2 Example

In this section, we demonstrate the proposed method by the example of matrix-matrix multiplication $C = A \star B$, where A, B are $n \times m$ and $m \times k$ matrix, respectively. First, an algorithm is presented in the form of supersteps. The execution time is estimated on a p processor mesh.

Algorithm

The algorithm is based on row-block partitioning on p processors. The initial data distribution and final results collection are performed by the host.

Superstep 0 Partition matrices into p blocks and assign ith block of A and B to processor P_i. One-to-all communication: the amount of data is $\frac{(n+l)m}{p}$.

Superstep 1 to p-1 Each processor performs computation on the assigned blocks. P_i generates a partial product for $A_i \star B_i$. Each processor P_i sends B_i to P_{i-1}. One-to-one communication: the amount of data is $\frac{nl}{p}$.

Superstep p Generate the last partial product. Each processor holds $\frac{n}{p}$ rows of the resultant matrix. The results are sent to the host. All to one communication: the amount of data is $\frac{nl}{p}$.

A and B are assumed to be double floating point operands. The amount of communication (in bytes) is obtained by multiplying the amount of data with the size of double floating point operands such as 32 or 64 bytes.

The IF1 graph of the example is shown in Figure 2. The arcs represent the communication among processors and the amount of communication can be obtained as described in Section 2. Array operations such as fetching, filling and replacing elements of an array, etc. are represented using AElement node, AFill

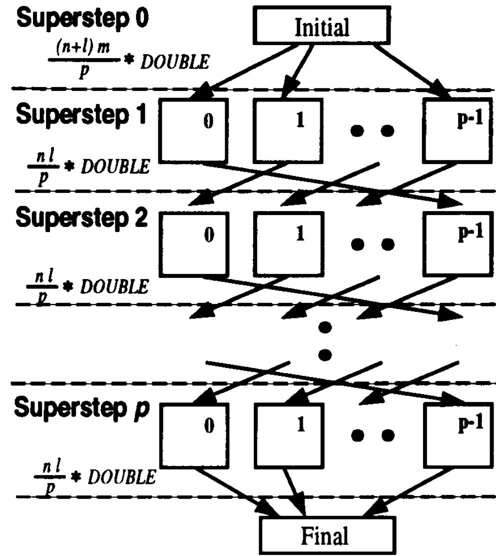

Figure 2: The supersteps in the form of IF1 graphs. The arcs are labeled with the amount of communication.

node, AReplace node, etc. From the graph, PCPV is generated. To simplify the process, we assume the simple nodes of IF1 graph as operations for code profiling.

Parametric computation code profiling: Parametric computation code profiling outputs $\vec{V}_t$ by tracing the IF1 graph. The number of operations inside a loop depend on the size of the loop.

Parametric communication code profiling: Communication code profiling provides the amount of communication based on the IF1 generation process as discussed in Section 2. As shown in the Figure 2, matching the output port variables with the input port variables of the next superstep provides the amount of communication for each superstep.

- superstep 0: one to all communication with $\frac{(32+64)128}{16} \star 32$bytes $= 24576$ bytes.

$$e_{one-all}^{comm} = s + \sqrt{p} + L \star r(p + \frac{p-1}{\sqrt{p}} + \sqrt{p})$$

- superstep 1 to 15: one to one communication with $\frac{32 \star 64}{16} \star 32$bytes $= 4096$ bytes.

$$e_{one-one}^{comm} = s + \sqrt{p} + L \star r \star (2 + 2\sqrt{p})$$

- superstep 16: all to one communication with $\frac{32 \star 64}{16} \star 32$bytes = 4096 bytes.

$$e^{comm}_{all-one} = s + \sqrt{p} + L \star r(p + \frac{p-1}{\sqrt{p}} + \sqrt{p})$$

Execution time estimation In order to compute $\vec{E}_t$, parametric computation benchmarking vector for the processors in the mesh should be obtained from parametric analytical benchmarking and is assumed as below.

$$\vec{B}^{mesh} = [b_+, b_*, \cdots, b_{Select}]$$
$$= [.1, 1, .5, .2, .5, .1, .2, .1, 5]$$

The vector $\vec{V}_t$ is now determined for the target machine, p = 16 processor mesh for n =32, m=128, k= 64. After some calculation, we can find the result $\vec{V}_t$:

$$\vec{V}_t = \begin{bmatrix} v_+ \\ v_\star \\ v_{<=} \\ v_{MOD} \\ v_{AFill} \\ v_{AElement} \\ v_{AReplace} \\ v_{FinalValue} \\ v_{Select} \end{bmatrix} = \begin{bmatrix} 32833 \\ 16400 \\ 18480 \\ 16 \\ 32 \\ 114688 \\ 32768 \\ 39070 \\ 16 \end{bmatrix}$$

Therefore,

$$e^{comp}_{mesh} = \vec{B}^{mesh} \cdot V_t = 50.9519 \ msec$$

V_t $\vec{B}^{mesh}$ Benchmarking results for communication is needed for the estimation of communication time. Setup time s and data transfer time r are assumed to be 80 and .5 microseconds, respectively. Data in double floating point are assumed to be 32 bytes long. Since in a mesh network, b = h = $\sqrt{p}$ = 4, the communication time can be estimated as follows,

- superstep 0:

$$e^{comm}_{one-all} = 291.924 \, (msec)$$

- superstep 1 to 15:

$$e^{comm}_{one-one} = 20.564 \, (msec)$$

- superstep 16:

$$e^{comm}_{all-one} = 48.724 \, (msec)$$

The estimation of communication time e^{comm}_{mesh} = 649.12 *msec*. Thus, the total execution time of the example on a 16 processor mesh is the sum of computation and communication time.

$$e_{mesh} = e^{comp}_{mesh} + e^{comm}_{mesh} = 700.0719 msec.$$

4 Conclusion

We have presented a methodology for estimating the execution time of applications in an HP environment. Specifically, we have proposed parametric code profiling and parametric analytical benchmarking techniques and have incorporated the concept of an architecture independent model. The methodology uses fine-grained computational details of the given code along with a high-level abstraction of the communication operations employed in the code.

References

[1] *Grand Challenges: High Performance Computing and Communication*, a report by the Committee on Physical, Mathematical, and Engineering Sciences, to supplement the U.S. President's Fiscal Year 1992 Budget.

[2] R. F. Freund, and D. S. Conwel, "Superconcurrency: a Form of Distributed Heterogeneous Supercomputing," *Supercomputing Review*, Vol. 3, No. 10, October. 1990, pp. 47-50.

[3] A. Ghafoor, and J. Yang, "Distributed Heterogeneous Supercomputing Management System",*IEEE Computer*, Vol 26. No. 6, June 1993, pp. 78-86.

[4] S. Hambrush, A. Khokkhar, "An Architecture-Independent Model for Coarse-grained Parallel Machines" Tech. Report, Computer Science Dept. Purdue University, October 1993.

[5] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. Wang "Heterogeneous Computing: Challenges and Opportunities",*IEEE Computer*, Vol 26. No. 6, June 1993, pp. 18-27.

[6] D. Pease, A. Ghafoor, at el., "PAWS: A Performance Evaluation Tool for Parallel Computing Systems," *IEEE Computer*, Vol. 24, No. 1, January 1991, pp. 18-29.

[7] R. Saavedra-Barrera, A. J. Smith, E. Miya, "Machine Characterization Based on an Abstract High-Level Language Machine", *IEEE Trans. Computer*, Vol. 38, No. 12, December 1989, pp. 1659-1679.

[8] L. Stapleton, "Meet the Metacomputer: Where the Network is the Computer,"*Supercomputing Review*, Vol. 4, No. 3, March 1991, pp. 42-44.