AMEEDA: A General-Purpose Mapping Tool for Parallel Applications on Dedicated Clusters

X. Yuan\(^1\), C. Roig\(^2\), A. Ripoll\(^1\), M. A. Senar\(^1\), F. Guirado\(^2\) and E. Luque\(^1\)

\(^1\) Universitat Autònoma de Barcelona, Dept. of CS
xiaoyuan@ow410.uab.es, a.ripoll@cc.uab.es, miguel.angel.senar@uab.es, e.luque@cc.uab.es

\(^2\) Universitat de Lleida, Dept. of CS
roig@eup.udl.es, fernando@eup.udl.es

Abstract. The mapping of parallel applications constitutes a difficult problem for which very few practical tools are available, most of which are tied to applications that follow a specific programming paradigm. AMEEDA has been developed in order to overcome the lack of a general-purpose mapping tool. The automatic services provided in AMEEDA include instrumentation facilities, parameter extraction modules and mapping strategies. With all these services, and a novel graph formalism called TITIG, users can apply different mapping strategies to the corresponding application through an easy-to-use GUI, and run the application on a PVM cluster using the desired mapping.

1 Introduction

Several applications from scientific computing, e.g., from numerical analysis, signal processing, image processing and multidisciplinary codes such as global climate modeling, contain different kinds of potential parallelism: task parallelism and data parallelism [1] [2]. Task parallelism results from the structure of the program. Independent program parts (tasks) can be executed in parallel on several disjoint processors. Data parallelism is the result of data independences. It allows that the same operation can be applied concurrently on different data. The efficient exploitation of potential parallelism available from the application leads to efficient parallel programs. When a user generates a parallel program, it is difficult to determine how many processors the application should use and how to distribute the application tasks in such a way that the completion time is minimized on a specific parallel machine. Using pure data parallelism, we have small computation costs and possibly high collective communications costs. Using task parallelism, we reduce the number of participating processors for an application and this for its part, reduces the internal communication costs. On the other hand, a reduced number of processors increases pure computation time.

Both data and task parallelism can be expressed using parallel libraries such as PVM and MPI. However, these libraries are not particularly efficient in exploiting the potential parallelism of applications. In both cases, the user is required to choose the number of processors before the computation begins and
once underway, this number cannot change. Moreover, the processor mapping mechanism is based on very simple heuristics that take decisions independently of the relationship exhibited by tasks. However, smart allocations should take these relationships into account in order to guarantee that independent tasks are executed concurrently by independent processors while tasks with a strong precedence dependency should be better allocated onto the same processor. This allocation will guarantee that the shortest running time is achieved. A good mapping strategy should also be responsible for allocating as many processors as can be efficiently used, according to the maximum parallelism achievable by the application.

Despite the existence of a large body of literature in the area of scheduling and mapping, most of the existing works have been focused on theoretical studies that have been applied in practice only to particular examples. Only a few works have proposed practical tools that support automatic mapping. In particular, PYRROS [3], Hypertool [4], and CASCH [5] are restructuring tools that automatically transform sequential programs to parallel programs, and the parallel code that is executed on a target machine (i.e., iPSC/2 hypercube, Intel Paragon and IBM SP2) is optimized by a proper scheduling and mapping algorithm. The scheduling algorithms used in these tools are based on a synthetic graph representation known as Task Precedence Graph (TPG). For cluster systems, CoPA [6] provides a mapping environment for coarse-grain applications consisting of different functional modules which are able to run all of them in parallel and as a consequence, they can be synthesized as a Task Interaction Graph (TIG).

In this work, we present a new tool called AMEEGA (Automatic Mapping for Efficient Execution of Distributed Applications) that takes into account information related to the temporal behavior of the parallel applications and automatically provides an efficient mapping on cluster-based systems. In contrast to the above mentioned tools, AMEEGA is not tied to a particular synthetic graph model. It accepts as input any parallel application written for the PVM environment with any task interaction pattern. AMEEGA extracts the most suitable graph model that synthesizes the application behavior and subsequently applies the suitable strategy. From this point of view, AMEEGA is an automatic general-purpose mapping tool that provides a unified environment for the efficient execution of parallel applications on dedicated cluster environments. It is also a useful tool for the practical evaluation of mapping strategies because it provides an easy-to-use interface that simplifies the execution of a given application under different allocation schemes.

In the following sections, we review some mapping background (Section 2), present a general overview of AMEEGA (Section 3) and some practical results obtained with AMEEGA when it was applied to a set of representative synthetic programs and to a real parallel image processing application (Section 4). The conclusions summarize the main contributions to this work.
2 Background and Motivation

Our work focuses on parallel applications based on the message-passing paradigm (i.e., applications consist of several tasks, each task is responsible for part of the application’s computational workload and interacts with the other application tasks by message exchange). Depending on the way in which work is divided into tasks, applications may be said to use task parallelism, data parallelism, or a mixture of both. Furthermore, we address the static mapping problem, which is supposed to deal with parallel applications that exhibit a fixed task structure and a significant regular behavior throughout their whole execution. Parallel applications with variable number of tasks (due to dynamic task creation and deletion) are beyond the scope of this work. Static strategies are not suitable for these because they require allocation strategies capable of reacting at run-time to the changes in the application. AMEEDA currently supports PVM applications. However, the general concepts underlying AMEEDA would be extendible and equally applicable to MPI applications, and work is currently in progress to also adapt AMEEDA to this library.

Despite its complete programming interface, PVM provides a very simple mechanism to carry out the allocation of tasks to processors. The default allocation method is based on a round-robin strategy which only obeys the order at which tasks are created. PVM offers the user the possibility to implement its own allocation methods by using the services of a PVM resource manager. With this service, the user may register a special task that will be responsible for deciding the allocation of all the tasks of a given program. Sophisticated methods may be included in this special task, which will provide a better performance of the application by generating a smart allocation. Unfortunately, writing such special task is not by any means a trivial thing. The complexity incurred in writing a generic resource manager would imply that users only write, if at all, a special resource manager for every new application.

On the other hand, considerable literature has been devoted to proposing and evaluating static mapping strategies for parallel applications [7] [8]. In general, static mapping strategies make use of synthetic models to represent the application. These models are based on graph formalisms. Two distinct kinds of graph models have been extensively used. The first is the TPG (Task Precedence Graph), which models parallel programs as a directed acyclic graph with nodes representing tasks and directed edges representing dependencies and communication requirements. The second is the TIG (Task Interaction Graph) model, in which the parallel application is modeled as an undirected graph, where vertices represent the tasks and edges denote intertask interactions.

The TPG model (also known as control or task parallelism) lets the programmer define different types of tasks that communicate with each other at the beginning and at the end through message-passing primitives. In contrast to the TPG, the TIG model is well-suited to data-parallel applications which apply the same operation concurrently on different elements of the data set [9].

In addition to the two above-mentioned models, the authors have proposed a new model, TTIG (Temporal Task Interaction Graph) [10], which represents ap-
lications that exhibit a mixture of task and data parallel in a more realistic way and which are defined by the programmer as a set of processes with an arbitrary interaction pattern among them. The TTIG model captures temporal information about parallel programs with a parameter called degree of parallelism, whose value ranges from 0 to 1. This gives information about the potential concurrency that each pair of adjacent tasks can achieve. Thanks to the degree of parallelism, the TTIG constitutes a generalized model that includes both the TPG and the TIG (if all tasks have a degree of parallelism equal to 0, the graph is equivalent to a TPG; a degree of parallelism of 1 between all communicating tasks implies that the application can be modeled as a pure TIG).

This paper focuses on the practical exploitation of the TTIG model by means of the AMEEDA tool. This tool is intended to significantly simplify the complexity related to the control of task allocation in parallel applications. On the one hand, AMEEDA provides a generic resource manager for PVM applications that is responsible for allocating the tasks of all parallel applications that the user submits to the system. Additionally AMEEDA integrates a set of automatic services that are used to compute the most suitable mapping for a given application. The resource manager will subsequently carry out task allocation according to the results of this computed mapping. A detailed description of AMEEDA follows.

3 Overview of AMEEDA

The AMEEDA tool provides a user-friendly environment that performs the automatic mapping of tasks to processors in a PVM platform. As several mapping strategies are integrated in the tool, it can be used to evaluate their performance when they are applied to real applications running on cluster platforms. First, the user supplies to AMEEDA a C+PVM program whose behaviour is synthesized by means of a tracing mechanism. This synthesized behaviour is used to derive the task graph model corresponding to the program, which will be used later to automatically allocate tasks to processors, in order to subsequently run the application. Figure 1 shows AMEEDA's overall organization and its main modules, together with the utility services that it is connected with, whose functionalities are described below.

3.1 Program Instrumentation

Starting with a C+PVM application, the source code is instrumented using the TapePVM tool (ftp://ftp.imag.fr/pub/APACHE/TAPE). We have adopted this technique, in which instructions or functions that correspond to instrumentation probes are inserted in users' code before compilation, because of its simplicity. Future work will investigate the use of other more effective instrumentation techniques that do not require, for example, recompiling the application with the instrumentation library [11]. Using a representative data set, the instrumented application is executed in the PVM platform, where a program execution trace is obtained with TapePVM and is recorded onto a trace file.
3.2 Synthesized Behaviour

For each task, the trace file is processed to obtain the computation phases where the task performs sequential computation of sets of instructions, and the communication and synchronization events with their adjacent tasks. This information is captured in a synthetic graph called the Temporal Flow Graph (TFG).

Figure 2(a) shows an example of the TFG graph for a program with 5 tasks \{T1,..,T5\}. It can be seen for example that task T3 has three computation phases with a computation time of 5, 30 and 15 units, respectively. It has a receive from task T1 with a volume of data equal to 15, and it communicates a volume of data of 5 and 10, respectively to and from task T5.

3.3 AMEEDA tool

With the synthesized behaviour captured in the TFG graph, the AMEEDA tool executes the application using a specific task allocation. The necessary steps to physically execute the application tasks using the derived allocation are carried out by the following AMEEDA modules.

1. Task Graph Model

Starting from the TFG graph, the TTIG model corresponding to the application is calculated. This model consists of a directed graph where nodes are tasks with their estimated computation time, and arcs join communicating tasks
and have two associated parameters: communication volume to be transferred between tasks and degree of parallelism of adjacent tasks. Figure 2(b) shows the TTIG graph corresponding to the temporal behaviour represented in Figure 2(a). From the analysis of this graph, it can be observed that task T2 has to be executed sequentially with T1, as it has a degree of parallelism of 0, while task T3 exhibits near maximum parallelism with T5, as the degree of parallelism is 0.9. In the case of all the degrees of parallelism being 1, the TTIG models the same situation assumed in the TIG model. If all the degrees of parallelism are 0, this corresponds to the same situation as having a TPG.

Note that, although different traces may be collected if an application is executed with different sets of data, only one TTIG is finally obtained, which captures the most representative behavior of the application. Execution times and communication volumes obtained in a given execution are made relative to the overall execution time and communication volume of the application. Later, the obtained values are averaged with other existing values obtained in previous executions.

An additional sub-module is included in this module of AMEEDA. It is referred to as Number of Processors-bound and estimates the minimum number of processors to be used in the execution that allows the potential parallelism of application tasks to be exploited. This is calculated using the methodology proposed in [12] for TPGs, adapted to the case of having the temporal information or tasks summarized in the TFG graph.

2. Mapping Method

Currently, there are three kinds of mapping policies integrated in AMEEDA that can be applied to the information captured in the TTIG graph of an application. Each of these corresponds to a group of mapping methods especially

Fig. 2. (a) Temporal Flow Graph (TFG). (b) Corresponding TTIG model.
targeted to the new TTIG model, and the classical TIG and TPG models. As we have previously mentioned, the TTIG is a sort of general model that contains the other two. Using one of these models, the suitable mapping method can be selected to do the allocation of tasks, whose operation is summarized below.

- **(a) TTIG mapping.** This option contains the MATE (Mapping Algorithm based on Task Dependencies) algorithm, based on the TTIG model. The assignment of tasks to processors is carried out with the main goal of joining the most dependent tasks (i.e., the tasks with greater communication and a lesser degree of parallelism) to the same processor, while the least dependent tasks are assigned to different processors in order to exploit their ability for concurrency. The interested reader may refer to [13] for a theoretical and detailed description of the algorithm.

- **(b) TIG mapping.** In this case, the allocation is carried out using the CREMA heuristic [14] which was proved to outperform other similar heuristics based on the TIG model. The heuristic is based on a two-stage approach that first merges the tasks into as many clusters as the number of processors, and then assigns clusters to processors. The merging stage is carried out with the goal of achieving load balancing and minimization of communication cost.

- **(c) TPG mapping.** The allocation is based on the TPG model. In particular, we have integrated the ETF heuristic (Earliest Task First) [15], which assigns tasks to processors with the goal of minimizing the starting time for each task, and has obtained good results at the expense of a relatively high computational complexity.

### 3. Connecting AMEEEDA with the Execution Platform

The physical mapping of tasks to processors is carried out according to the allocation yield by the selected mapping method. To integrate the mapping method in PVM we used the Resource Manager (RM) mechanism. The RM is a PVM task responsible for making task and host placement decisions. By intercepting libpvm calls related to the creation of tasks, the RM can directly command the PVM daemons to start a task to a specific host. The steps performed by a RM are depicted in Figure 3(a), where Task1 sends a SMLSPAWN message to the RM through *pvmd1* in steps 1 and 2. The RM calculates the corresponding allocation (which by default will be carried out in a round-robin ordering), and sends an SMLEXEC message to *pvmd2* and *pvmd3* to create Task2 in both (steps 3 and 4).

In AMEEEDA, the RM is implemented in such a way that when a SMLSPAWN is received, a request to the tool AMEEEDA core is carried out to obtain the allocation calculated using the selected mapping. Subsequently, the TMLEXEC is sent to the suitable daemons. This idea of this modification is represented in Figure 3(b).

### 4. User Interface

This module provides several options through a window interface that facilitates the use of the tool. The *Task Graph* sub-module allows the information from
the TFG and the TTIG graph to be visualized. Additionally, the TTIG graph can be displayed. The Architecture sub-module shows the current configuration of the PVM virtual machine, and allows us to modify this configuration by adding or removing hosts. The execution of the application, with a specific allocation chosen in the Mapping option, can be visualized using the Execution tracking submodule that monitors it and graphically shows the execution states for tasks (running, communicating, blocked, exited). Finally, the Performance option gives the final execution time, processor occupancy and speedup of a specific run. It can also show historic data recorded in previous executions in a graphical way, so that performance analysis studies are simplified.

4 Experimental Results

The goal of this section is to show the viability and usefulness of AMEEEDA, as well as to highlight the benefits achievable in task assignment, using the mapping methods integrated in the tool. The experiments were conducted on a cluster of 10 Linux machines running PVM 3.4. Each machine was a Pentium II interconnected with a 100 Mbps Fast Ethernet network. Although the cluster was a dedicated one and the experiments were carried out under the same conditions, different runs of the same program produced slightly different final execution times. Hence, average case results are reported for several runs of the same application and mapping. On the one hand, we used a set of synthetic applications that are representative of real program structures, and on the other hand we experimented with a real application.

The set of synthetic applications that we experimented with are C++PVM programs with regular task graph topology, which correspond to the pipe, mesh, binomial tree and in-out tree structures. In order to show the effectiveness in mapping by taking into account the temporal behaviour captured in the TTIG model, the applications under study had the following characteristics: (a) all the tasks had uniform computation time and, (b) the ratio between computation and communication ranged from 4 to 10. In this way the time incurred in
communications is small compared to the time in which tasks are carrying out computation. Each graph was generated with different levels of degree of parallelism between all adjacent tasks. The values used can be classified in the four following cases: low (ranging from 0 to 0.3), medium (0.3 to 0.7) high (0.7 to 1) and variable: (0 to 1).

Each application was executed with the three mapping strategies integrated in AMEEDA, based on the TTIG, TIG and TPG models. It is worth observing that the application tasks do not communicate at the beginning and at the end, as is supposed in the TPG model. Despite this, we carried out this mapping strategy in order to show the benefits of using the knowledge of potential parallelism of tasks. On the other hand, like other TPG mapping strategies, ETF not only assigns tasks to processors, but also indicates the execution ordering of tasks in each processor. However, in our experiments this order could not be strictly forced, and the ready-to-run tasks assigned to a processor competed altogether for the CPU using the time-slice scheme of the Linux scheduler.

Figure 4(a) shows the average percentage of gain obtained with MATE over ETF, the number of processors varying from 2 to 10. As can be observed, MATE always achieves better results over ETF. In the cases where tasks exhibited a medium, high or a variable degree of parallelism, these results simply confirm the usefulness of the degree of parallelism parameter included in the TTIG model (improvements were up to 35.2%, 13.5% and 50.2%, respectively). Without this parameter, ETF treats all the adjacent tasks as being sequential, while in practice some of them will have distinct possibilities of concurrent execution. In the case of tasks with a low degree of parallelism, which is close to the basic assumption of the TPG model, one would expect both strategies to be very close or a better performance of ETF over MATE. This is not observed because, as mentioned before, the full benefits of a TPG mapping strategy are achieved by following a task allocation and a scheduling order between tasks allocated to the same processor. And this order can only be ensured, for instance, by changing the CPU scheduler (in the Linux kernel), or by adding a wrapper function that encapsulates all the tasks assigned to the same processor and calls each one of them in the right order. We have discarded any of these solutions because the first severely restricts the portability of the environment and the second introduces a significant overhead.

Figure 4(b) shows the average percentage gain obtained by MATE over CREMA. It can be seen that MATE obtains remarkable gains in all cases, due once again to the exploitation of the temporal behaviour contained in the TTIG graph. This fact is more significant in programs with a low, medium or a variable degree of parallelism. For programs with a high degree of parallelism, MATE also benefits from the way it traverses the graph structure. All the graphs used in these tests had a regular structure in which chains of dependent tasks were captured by MATE, due to the way in which it traverses the graph, as it tries to first allocate tasks that are direct neighbors of other already assigned tasks.

In order to show the influence of the different mapping strategies with a real application, we carried out the mapping of an image processing application
called BASIZ (Bright And Saturated Image Zones). BASIZ is a parallel application, with task and data parallelism, which performs the detection of zones having more brightness and colour intensity for a given image. The BASIZ application was developed on the existing support of URT modular image processing (http://www.autah.com), from which the application has been parallelized using PVM. The BASIZ application is composed of a set of 26 tasks, which are organized in the following seven pipelined stages: colour separation, blur, adding, merging, image conversion, threshold and marking zones.

Using the mechanisms provided in AMEEDA, the corresponding TTIG model was obtained. The computation times for tasks and communication volumes deduced in the graph show significant differences, as well as several situations related to tasks parallelism. BASIZ has a stable execution behaviour for different runs and different images, so this is an adequate application for carrying out the mapping statically. Figure 5 corresponds to the AMEEDA window, showing the TTIG graph for BASIZ, together with the speedup graphic generated with the Performance sub-module, obtained when the BASIZ application was executed using the PVM default allocation and the three different mapping strategies under evaluation.

It can be observed that when four or fewer processors were used, all the strategies provided similar speedup values, because as more tasks were allocated to the same processor, they easily had ready tasks to execute and dependencies had less influence. For more than four processors, the differences become greater, and MATE and CREMA mappings provide significant improvements with respect to the ETF and PVM round-robin allocations. This is because, despite the task graph topology being a TPG-like structure, the behaviour of tasks is better represented with the TIG or the TTIG models. Most of the connected tasks can execute concurrently most of their time, and only a few set of tasks exhibit strong dependencies. This situation explains the best performance achieved by CREMA
Fig. 5. AMEEDA windows showing the TTIG graph and the speedup for BASIZ.

over ETF. Furthermore, the knowledge about dependencies that has MATE, allows it to further refine the allocation generated by CREMA. It is worth pointing out that the Number of processors-bound module of AMEEDA estimated that a minimum of eight processors were necessary to exploit the application parallelism. In this case, it can be observed that this bound also coincides with the point at which the maximum speed-up is achieved. By adding more CPUs, the speedup remains equal or falls a little (because the benefit of more parallelism is compensated by the extra amount of communication between tasks allocated in different processors.

5 Conclusions

We have described the AMEEDA tool, a general-purpose mapping tool that has been implemented with the goal of generating efficient allocations of parallel programs on dedicated clusters. AMEEDA is based on a new graph model (TTIG), which is a generalization of the other two most used models in the literature to solve the mapping problem (namely, the TPG and the TIG). As a consequence, AMEEDA provides a unified environment for computing the mapping of long-running applications with a relatively stable computational behavior. The tool is based on a set of automatic services that instrument the application and generate the suitable synthetic information. Subsequently, the application will be executed following the allocation computed by AMEEDA, without any user code re-writing. Its graphical user interface constitutes a flexible environment for analyzing various mapping algorithms and performance parameters. In its current state of implementation, the graphical tool includes a small set of representative mapping policies. Further strategies are easy to include by simply providing an executable function to implement them, which is also a highly de-
sirable characteristic in its use as a teaching and learning aid in understanding mapping algorithms. As future work, AMEEDA is going to be enhanced in such a way that the most convenient mapping strategy is being automatically chosen, according to the characteristics of the application graph, without the user intervention.

References