

10-1-1992

Distributed Heterogeneous Supercomputing Management System

Arif Ghafoor

Purdue University School of Electrical Engineering

Jaehyung Yang

Purdue University School of Electrical Engineering

Follow this and additional works at: <http://docs.lib.purdue.edu/ecetr>

Ghafoor, Arif and Yang, Jaehyung, "Distributed Heterogeneous Supercomputing Management System" (1992). *ECE Technical Reports*. Paper 270.

<http://docs.lib.purdue.edu/ecetr/270>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Distributed Heterogeneous Supercomputing Management System

Arif Ghafoor, Jaehyung Yang
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907
Tel. (317) 494-0638
FAX (317) 494-6440

E-mail: ghafoorQecn.purdue.edu, jaehyung@ecn.purdue.edu

Abstract

An efficient use of a Distributed Heterogeneous Supercomputing System (DHSS) **requires** a thorough understanding of applications and their intelligent scheduling **within** the system. In this paper we present a general management framework for the DHSS, by introducing an application characterization technique, called Code Flow Graph (CFG) and Code Interaction Graph (CIG). These models are based on code profiling and **analytical** benchmarking and provide a detailed architectural-dependent characterization of DHSS applications. A **general** cost function is presented that is based on the execution and **1/O**overheads associated with applications. An **optimal** scheduler tries to minimize this cost; the design of which is an NP-complete problem. We describe how network caching can help to reduce the **complexity** of scheduling in a DHSS.

Keywords: Heterogeneous Supercomputing, Code Profiling, Benchmarking, Task Flow Graph, ~~Task~~ Interactive Graph, Scheduling, Mapping.



1 Introduction

The concept of Distributed Heterogeneous Supercomputing System(**DHSS**) has been introduced quite recently [5, 6], with the objective to achieve a super-linear speedup using current supercomputing technology. For such a system, multiple heterogeneous supercomputers are interconnected over high speed networks to provide a **computationally** powerful environment to solve many engineering and scientific problems which are intractable on a single supercomputing system. A DHSS is also expected. to outperform a **homogeneous** supercomputing system (HSS) because no matter how powerful a single machine or a set of homogeneous machines might be, HSS cannot satisfy **the** diverse characteristics of program codes efficiently [5]. Specially, ill-matched codes can degrade the overall performance of these systems. Building a suite of heterogeneous supercomputers with existing machines having diverse computational characteristics can provide a significantly **more** effective environment for solving complex problems. However, an efficient use of such a system requires a thorough understanding of characteristics of applications, machine architectures and their operational features.

A **number** of **DHSS's** have been proposed recently, with a few of **them** already **prototyped**. The most noticeable are the five gigabit network testbeds, namely; Aurora, Blanca, **Casa**, Nectar and **Vistanet** [7]. The functional concept behind Casa and Nectar **testbeds** resembles more closely to a DHSS. However, these **testbeds** are focused to solve a specific set of applications and cannot manage resources for a wide **variety** of applications. The future DHSS, on the other hand, are expected to serve a large variety of users developing diverse applications and codes which are expected to **run** concurrently on **various** machines within DHSS. One of the major requirements **for** future DHSS,

therefore, is to manage applications and find a suitable match between the codes¹ of these applications and machines. Another concept that is closely related to DHSS is “**superconcurrency**” [5], which is targeted to achieve maximum performance for a suit of heterogeneous machines. In order to achieve this objective, a code is assigned to the best matching machine using information about the code profiling and **analytical** benchmarking. This concept has been proposed for Distributed Intelligent Network System DINS [5]. However, DINS has limited utility since it does not evaluate the overall structures of applications and I/O characteristics which are crucial to achieve a true “**superconcurrency**”. With the latency across gigabit networks becoming virtually negligible, the I/O **bottlenecks** for data exchange **as** well as data conversion overhead **among** different machines **can** be significantly high, thus resulting in high communication overhead and hence limiting the overall performance of a DHSS

In **order** to handle these issues, an integrated approach for managing a DHSS is needed, which can allow management of both computational and network resources effectively by adapting to the needs of applications and providing a true "superconcurrent"ⁿ environment. **One** such system, which we call Distributed Heterogeneous **Supercomputing** Management System (DHSMS), is suggested in this paper. Basically, through DHSMS we **describe** a framework for the management of DHSS by proposing an application **characterization** technique based on code profiling and computation and I/O benchmarking. We discuss how I/O benchmarking can be used to perform data caching over the network in order to reduce application management complexity.

The objective is to propose a general framework which is not restricted to any type or class of **supercomputers** rather it is applicable to any combination of such machines. The

¹We will use the terms code and **task** interchangeably in this paper.

proposed DHSMS has some common base with DINS in that it seeks a **good** performance by efficiently managing (scheduling / mapping) application codes across a pool of available **machines** in order to achieve a super-linear speedup. However, it **differs** from DINS in various aspects. For DHSMS, we propose a systematic methodology **for** both code profiling and analytical benchmarking and suggest a "Universal Set of Codes" (USC). The proposed USC provides a comprehensive methodology to generate architecture-dependent code-profiles at varying levels of details. Second, as indicated above, **I/O** benchmarking is also taken into account while managing applications since we expect that **I/O** subsystems of machines can become bottlenecks in a DHSS. Furthermore, we describe how *network caching* of data communicated among machines can be used to increase the performance of a DHSS. Based on the proposed USC and **I/O** benchmarking, we propose two architecture-dependent characterizations of DHSS **applications**, which are called Code Flow Graph (CFG) and Code Interaction Graph (CIG). These graphs possess enough information about applications that is useful for their **scheduling/mapping**.

This paper is organized as follows. In the next section, CFG and CIG are introduced. Section 3 describes an overall architecture of a DHSMS. In Section 4 we briefly describes an **experimental** prototype of DHSMS, currently being developed. Section 5 concludes this paper.

2 A Characterization of Applications for DHSS

A **distributed** application consists of a set of tasks with certain relations among them. Tasks **are** the basic units handled by the proposed DHSMS. To run **an** application efficiently, a DHSMS needs to analyze both computational and communicational requirements of the application. Formally, an application can be modeled **either** as a Task Flow

Graph (TFG) or a Task Interaction Graph (TIG) [1]. TFG is used to **express** explicit precedence relationships among the tasks of the application, while TIG is more suitable for representing distributed interactive tasks without explicit dependencies. Scheduling and mapping algorithms are used for TFG and TIG, respectively. Although both TFG and TIG **are** useful models, their use is limited only to homogeneous **architectures**, since they are architecture-independent models and they do not carry any information about the behavior of tasks on heterogeneous systems.

For a DHSS a more precise and general method for application **characterization** is needed, **which** should not only incorporate the information about the "degree of suitability" of a task to a specific machine, but also quantify the communication interaction among **the** tasks. This interaction is an important parameter since data needs to be **exchanged** among various machines which may have diverse I/O architectures with drastically different performance profiles.

For **the** proposed DHSMS we introduce the notion of Code Flow Graph (CFG) and Code Interaction Graph (CIG) which solve these problems by **providing** a detailed architecture-dependent task and I/O characterization. Code profiling is used to characterize tasks in order to identify those tasks which have the same computational behavior [5], and to evaluate "degree of match" between the codes and machines. Very few code profiling methodologies, in the context of DHSS, have been proposed in literature [12]. However these methodologies have limitations in their applicability. Most of them are based on a rather simplistic and highly abstract view of parallelism. The detailed architectural **knowledge** has not been taken into account in such **methodologies**. New code profiling methods are needed which can incorporate detailed architectural characteristics so that **these** profiles can be more accurate and used for making scheduling and mapping

decisions intelligently **as** they can significantly impact the execution of **applications** [4]. However, **there** is a trade-off between the accuracy of the information generated by a profile and the complexity involved in generating it.

For task **scheduling/mapping**, code profiling itself is not sufficient, **rather**, we require an **estimate** of the execution time of a code on a specific machine. For **this** purpose, we also need *analytical benchmarking*; a process used to estimate performance of a machine relative to a baseline system [5]. Up to now, research on benchmarking **has** been focused on devising methodologies to measure the overall performance of each machine on a realistic application program which is composed of several tasks with different processing requirements. However since in a DHSS environment, an application is decomposed into multiple **tasks** which run separately on different machines, it is important **that** analytical **benchmarking** for a DHSS should be able to estimate the performance of a machine on each part of the application **as** well as the performance of the I/O **subsystem** the machine. Since, the ultimate objective is to combine both code profiles and benchmarks together, we must have a finite set of codes which can be used for both the purposes. One approach is to define a "Universal Set of **Codes**"(USC) which can be viewed as a "standardized universal set" of benchmarking programs, that can also provide information (profiles) about the effect of architectural characteristics of the machine. We can then use codes from the USC for both generating code profiles and obtaining **benchmarks** which can then be used to estimate the execution time of a code on a specific machine.

Most of the benchmark programs are architecture-independent and cannot provide realistic and meaningful profiles about machines. This is due to the fact that such programs might not map properly on the machine itself, and, instead of yielding benchmark profiles, they can even result in a negative speedup, that is, a performance worse than a



uniprocessor. For example, analyses have shown that if the standard **molecular** motion computation algorithm is executed on a supercomputer with multistage interconnection network, such as Butterfly System or on a shared bus interconnection system, such as Multimax, the speed up approaches zero as we increase the number of processors beyond a **certain** value [4]. It is, therefore, highly desirable that benchmark programs should be **written** based on the architectural features of machines. The proposed architecture-driven USC provides one such solution.

There can be many ways to synthesis a USC. Our approach is hierarchical and it not only provides a systematic way of generating this set, but also provides a flexibility to the user to choose a subset of USC, in order to achieve the desired **accuracy** in profiling and **benchmarking**.

Similarly, for quantifying **I/O** overhead due to communication interaction among machines, it is desirable that machines in a DHSS should also be benchmarked for generating **I/O performance** profiles. These profiles can provide information about the timing delays in transferring data among machines. Such delays are caused by the architectural constraints of the **I/O** subsystem. In Sections 2.2 and 2.3, we discuss **these** benchmarking issues in more detail.

2.1 A Hierarchical Scheme for Generating USC

The hierarchical scheme for generating USC is basically a detailed architectural characterization of supercomputers. At the highest level, one can select the type of **processing parallelism** for classifying architectures. At the second level, a further classification of these architectures can be carried out based on the finer architectural features such as **organization** of the memory system, interconnection topology, etc. An important feature

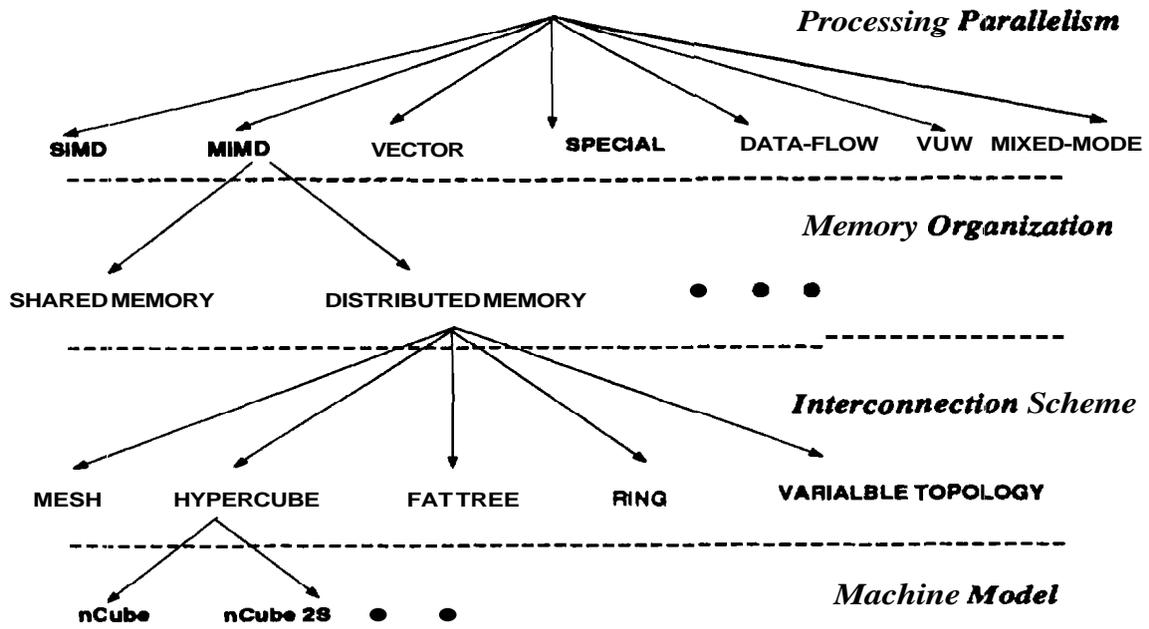


Figure 1: The Machine-driven Tree Structured Universal Set of Codes.

of this structure is that the levels in the hierarchy are selected in such a way that the main features of the architecture being characterized at any level are related **and** they do impact the execution of a code. A similar approach has been used to characterize supercomputers for evaluating their performance [9]. The leaf nodes of this hierarchy correspond to the **actual** machine models present in a DHSS. As an example, Fig. 1 shows one such possible classification hierarchy. In this example, the first level is classified according to the type of parallelism of the machines, namely; SIMD, MIMD, VECTOR, etc. The second level further classifies these machine types into different **categories** according to their **memory** organization such as Shared Memory system, Distributed Memory System, **Hierarchical** Memory System, etc. The detail and the complexity of information about architectural features increase as we go down the hierarchy.

The proposed hierarchy can be used to generate a USC. This can be **done** by assigning a code type to each node of the "hierarchical tree". The path from **the** root node to another node provides profile information (suitability of those architectural features which are **given** by the path) for the code associated with that node. A **more** detailed profile can be used to screen out machines which may have identical benchmarks. This screening can then provide a better estimate for the execution time.

Based on this hierarchy we define a Code Profile Vector (CPV), \vec{V}_i , for a given task t and **for** each level of the hierarchy. We assume that each level nodes are labeled from 1 to C , where C is the number of nodes at that level. This vector is given as:

$$4 : t \rightarrow [v_0(t), v_1(t), v_2(t).. v_C(t)]$$

The elements $v_i(t)$'s of this CPV represent the degree of match **that** exists between the **task** t and the code associated with the i -th node present at the level for which the vector **is** being generated. Such a match is determined based on **many** factors such as



the **amount** of parallelism present in the task, number of iterations of **loops** etc. Note, the size C of the CPV is the same as the size of the subset of codes of USC at that level.

For **example**, if a user selects the first level of hierarchy in Fig. 1, the length of the CPV is 7, corresponding to the type of processing parallelism, namely; SIMD, MIMD, VECTOR, SPECIAL, DATA FLOW, MIXED MODE. Similarly, if the user specifies a more detailed characterization, say up to level 2, then CPV will be of length of 14, corresponding to the two cases of memory organization (distributed and shared), with each one in turn consisting of the seven cases of the first level.

In **many** cases, code profiling may need to be done on-line and hence it introduces a run-time overhead. A "detailed" profile may take into account all the important architectural characteristics of a machine, such as the type of parallelism, **interconnection** scheme, **memory** organization scheme, etc., as shown in Fig. 1. **Generation** of this profile requires analysis of the task features with respect to the architectural characteristics defined at the selected level in the hierarchy. Although such a profile provides very useful information for efficiently **scheduling/mapping** of a task via accurately matching it to a machine, it can only be generated at the cost of an increased overhead **associated** with the analysis of the task. A "coarse" profile, on the other hand, can be generated with a relatively low overhead by choosing only a few levels in the hierarchy. However, such a profile may not be accurate enough for **scheduling/mapping** tasks effectively. An **example** of a coarse profile can be the one based on only the first level of the hierarchy that contains types of parallelism of processing, such as SIMD, MIMD, VECTOR, etc. However, such a profile can ignore many other important features of machines constituting a DHSS. This "accuracy vs. complexity" trade-off depends on the level selected in the hierarchy. This selection can be a part of the user-specified processing requirements.

Classification or ranking of code types as a code profiling tends to force discretization, ignoring the differences between actual code and the benchmark code, **such** as the one belonging to the USC. This may result in an erratic performance estimation of code. To minimize this discretized error, we can use a continuous function $v_i(t)$ as a measure of code profiling. Another important characteristics of using continuous **function** is that it provides a method to measure suboptimal selection in case a best-matching machine is not available. A continuous code-profiling method is described in [12].

2.2 Computation Benchmarking

We have already discussed that to accurately estimate the performance of a code on a certain machine, we need a standard set of codes based on architectural features, which both code profiling and benchmarking can use on a unified basis. **Here** we describe a **methodology** of benchmarking based on this concept using the architecture-driven USC **discussed** in the previous section.

There exist a number of methodologies to benchmark parallel machines, such as Kernel, *Partial* (Trace) Benchmarking, Synthetic Benchmarking, etc [2]. Also, some research results on the performance measures of benchmarking and combining several benchmarking results have been reported in [11]. A number of codes for benchmarking the performance of parallel machines have been proposed. They include Dhrystone, Whetstone, etc [2]. In a DHSS environment, an application is decomposed into multiple tasks that run separately on different machines. It is, therefore, important that analytical benchmarking for a DHSS should be able to estimate the performance of a machine on each part of the application. Also, as we have already mentioned, a benchmark program must take



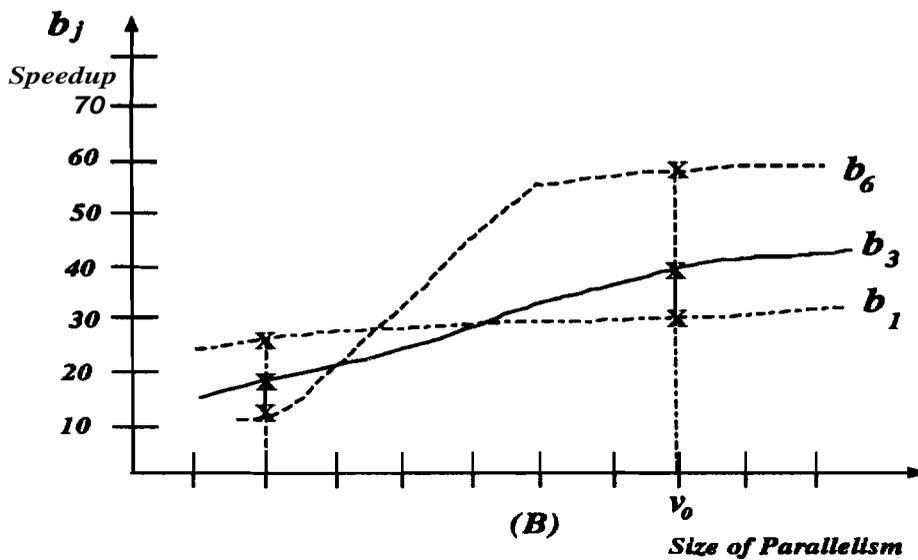
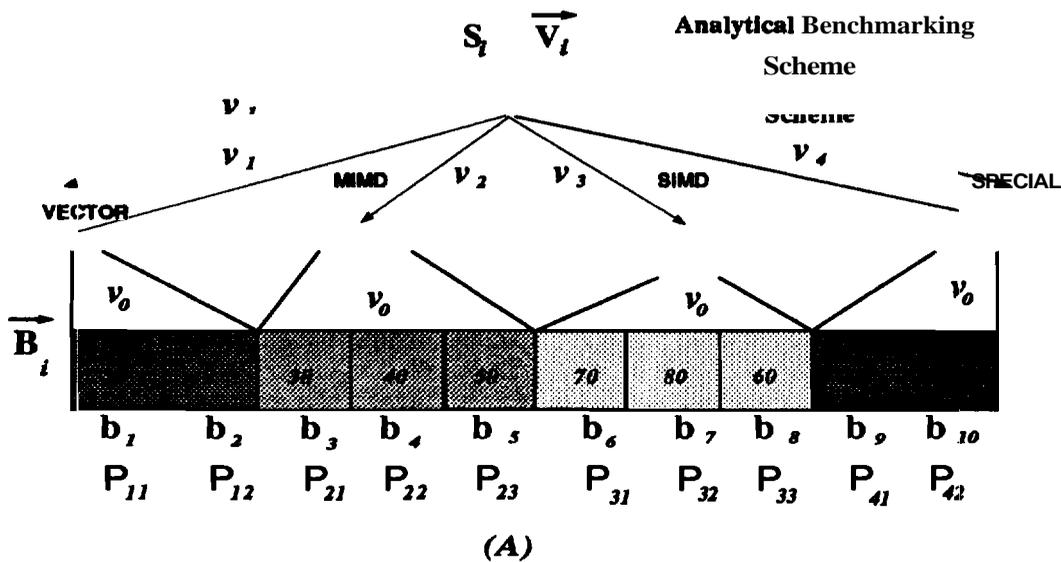


Figure 2: Generation of an Analytical Benchmarking Vector. (A) Each block represents a machine model and is grouped according to its machine type defined by the hierarchy. (B) shows benchmarks as functions of size of parallelism.

into account the architectural characteristics of machines. However, existing benchmarking programs are not specially designed to measure the architecture specific performance, rather **their** aim is to measure the overall performance of each machine **under** a simulated application environment. Some examples of such benchmarks can be found in Perfect Club [10], although some benchmarks in this case also are still being developed.

Formally, analytical benchmarking based on a code p can be defined by the following vector, which we call Analytical Benchmarking Vector (ABV) $\vec{B}_p(n)$.

$$\vec{B}_p(n) = [b^j(n)], j = 1 \dots M.$$

M is the number of machine models and $b^j(n)$'s represent the expected speedup obtained for machine model j , compared to the baseline serial machine. n is the size of parallelism in p . It is important to mention that such a benchmarking should be conducted on each machine model and the benchmarking code should be the code of corresponding machine type from the proposed USC.

An example of the benchmarking is illustrated in Fig. 2. In this figure, the ABV is based on the first level of hierarchy that consists of four machine types, namely; SIMD, MIMD, VECTOR, and SPECIAL. For the purpose of illustration, we represent machine models in the DHSS as P_{ij} 's, where i identifies the machine type and j indicates the machine model for that type. For example, nCube, CM-5, Paragon XP/S, etc., belong to the **same** class of machines, that is the MIMD. Figure 2(B) shows b^j 's as functions of the size of parallelism (n) which are obtained through the benchmarking code for each machine type. As shown, the ranking in speedup between machine **models** can change depending on the size of parallelism in benchmarking code [5]. Figure 2(A) shows an ABV, for a task S_i , having a parallelism of size v_0 , that results from code profiling. The values b_1 , and b_2 correspond to machines of type 1 ($i=1$), b_3 , b_4 , and b_5 corresponds to

machines to types 2 ($i=2$), etc. By dividing the expected execution time, s_i of the task i on the **baseline** system, by the values of these benchmarks and corresponding v_i 's, we can get **the** estimated execution time on each machine. This is discussed in Section 2.4.

2.3 I/O Benchmarking

For **analytical** benchmarking of I/O subsystems of supercomputers, not **much** work has been done. The I/O overhead depends on many factors, such as the effective bandwidth of **memory** channels, topological characteristics of the I/O **interconnection** network, the number **and** the speed of the I/O processors, etc. Accordingly, I/O benchmarking of a given architecture can be expressed as a performance function that depends on the amount **of** data being transferred through the I/O subsystem of the **machine**. For a typical I/O subsystem, this function can be given in the form of a **performance** graph, as shown in Fig 3. Typically such a function shows a linearly **increasing** latency time until it **reaches** a saturation point as shown in Fig. 3. This linear growth in the rate of latency is determined generally by a a single component; probably the **slowest** one in the I/O subsystem. However, beyond the saturation point the rate of growth in the latency can increase substantially due to the saturation and loading of various components. This saturation may be due to the higher contentions within communication interconnections, the physical limitation on the movement of disk heads, etc.

Based on these functions, one possible method to specify I/O **overhead** for an application is to use a vector of length M , \vec{D}_i , which we call Communication Overhead Vector (COV). This vector is given as follows:

$$\vec{D}_i = [d_1(a_i), d_2(a_i), \dots, d_M(a_i)]$$

The element $d_j(a_i)$ represents the expected I/O overhead of machine model j , using

its performance function d_j (Figure 3) evaluated for the communication cost a_i associated with a link i of a TFG(TIG). We can represent communication overhead associated with the whole task graph in terms of these functions. For the machines constituting the DHSS, these functions need to be tabulated. It is important to mention that the communication cost a_i between two tasks in TFG (TIG) generally represents an **aggregated** value. In reality, the exchange of data exchanged machines may be intermittent. Therefore, some sort of "stochastical performance profiles" may be more suitable.

Data conversion is another critical factor that restricts the performance of a DHSS, because it is a run-time process and execution of tasks cannot continue until data conversion is completed. The overhead associated with this process depends on the amount of data being transferred and the data types used by the communicating machines. Also it depends on the efficiency of the conversion process for a specific data type. We just assume conversion process only depends on the data size and its type. Accordingly, to handle data conversion cost, additional overhead can be added to I/O function.

Using both code profiling and analytical benchmarking, we now describe how to generate a CFG/CIG.

2.4 The CFG and CIG

Using a code profiling technique, such as the proposed USC, and analytical benchmarking, we can generate a CFG and CIG from TFG and TIG, respectively. The overall process of generating a CFG (CIG) is illustrated in Fig. 4. Starting with a TFG (TIG), which describes the computation cost of each task on a baseline system and the communication cost in terms of amount of data transmitted among tasks, an intermediate code flow graph (ICFG) or code interaction graph (ICIG) is generated, using code-profiling information.

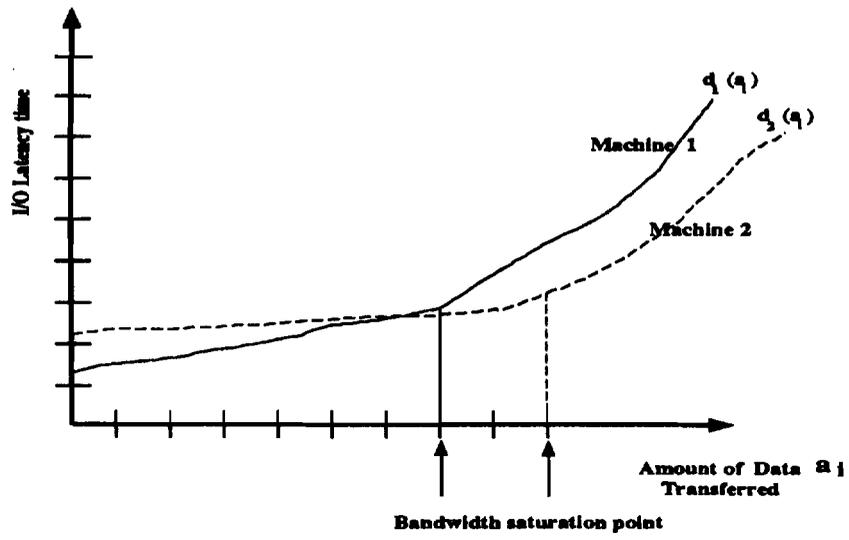


Figure 3: I/O Benchmarking. I/O Latency time vs. amount of data transferred.

As mentioned earlier, as a result of profiling each task in the TFG (TIG) is assigned a code profiling vector (CPV), which depends on the level of the hierarchy selected by the user.

The communication cost, in a ICFG (ICIG), represents the amount of data to be transferred among tasks and it stays the same as given in the original TFG (TIG). The ICFG (ICIG) is then evaluated using benchmarks and is translated into the final CFG (CIG). This transition consists of following steps.

1. Each task in an ICFG (ICIG) is labeled with an estimated execution time vector $\vec{E}_i = [e_1, \dots, e_M]$, where M is the number of different machine models, and e_i describes the estimated execution time of a code on the machine model i . Its value is given as $e_i = s_i / (v_j \cdot b^j(n))$, where s_i represents the execution time on the baseline system.
2. Each link in an ICFG (ICIG) is labeled with a COV $\vec{D}_i = [d_1(a_i), d_2(a_i), \dots, d_M(a_i)]$. As mentioned earlier, an element $d_j(a_i)$ of this vector describes the expected I/O overhead function of machine j , evaluated at the communication cost a_i associated with the link in the ICFG (ICIG), that is used by the machine j , after scheduling/mapping of the application. The I/O overhead function $d_j(a_i)$ as shown in Fig. 3 is used for this purpose. At this time, any data conversion overhead can also be incorporated in $d_j(a_i)$'s.

The resulting graph is a CFG (CIG) which carries detailed information about the machine-dependent execution and I/O performance of the tasks and data communication associated with a TFG (TIG). This elaborated machine-dependent characterization of DHSS applications is important for the DHSS to carry out its task management functions. In the next section we now describe a framework for such management.

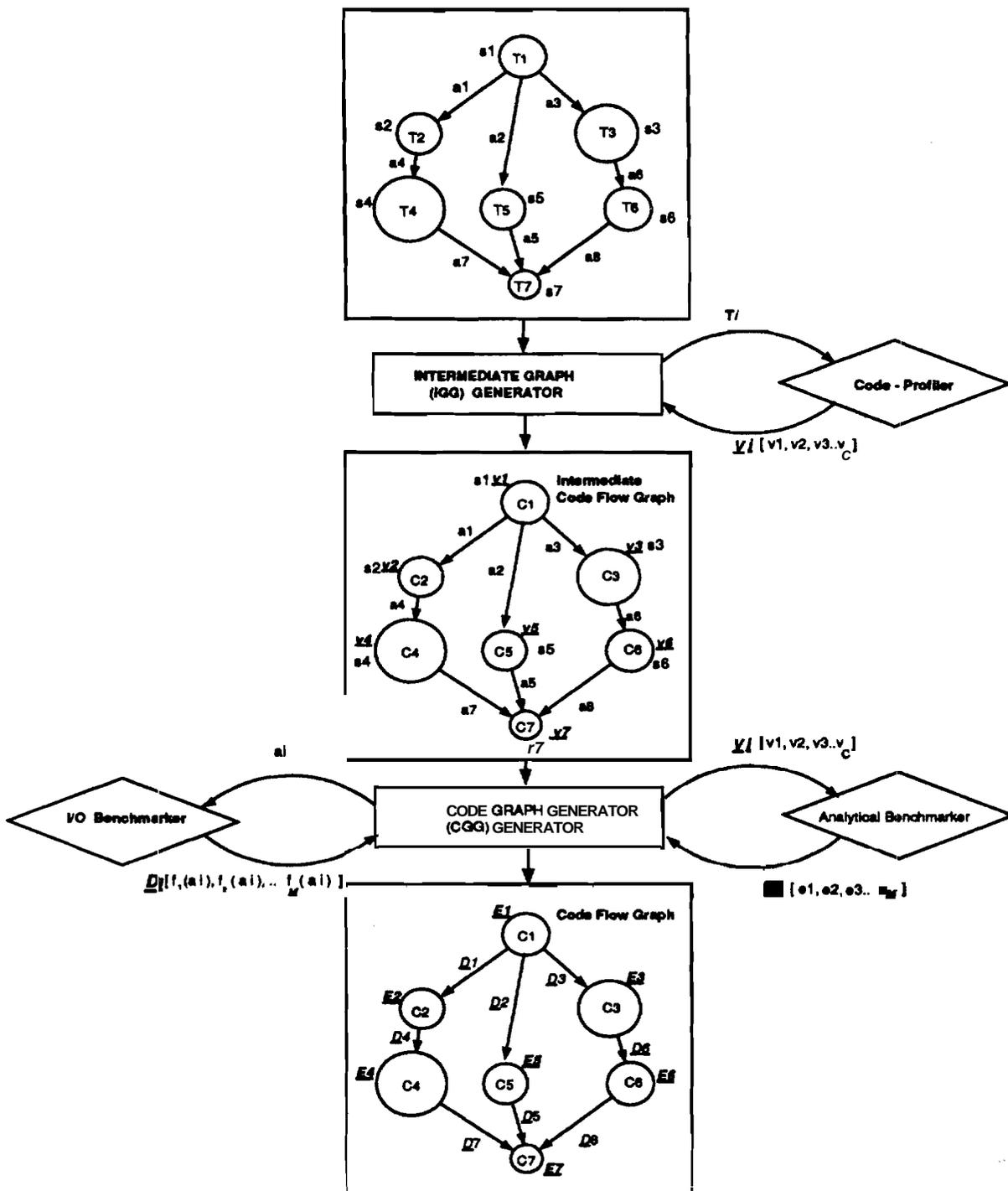


Figure 4: Code Flow Graph and Code Interaction Graph. Attached to node is a code-profiling vector. Each value of component of a vector represents the degree of match to a specified architecture in the specified level of the hierarchy

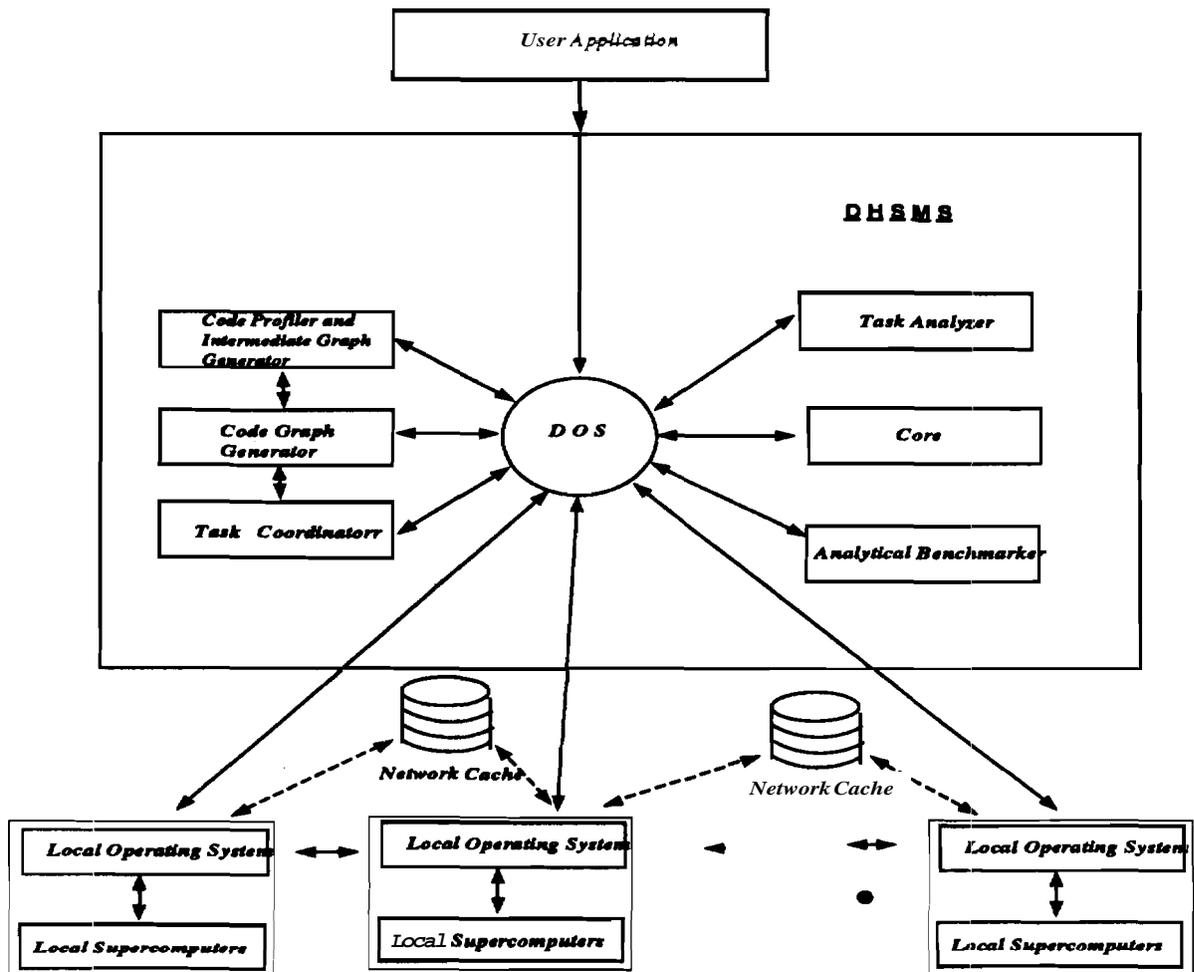


Figure 5: An Architecture of DHSMS

3 A. Framework for DHSMS

Based on the concepts discussed in the previous sections, we now present an architecture for DHSMS that provides a framework to manage applications for a DHSS. The proposed DHSMS differs from existing experimental testbeds in a sense that it provides an application management system which can accommodate different application characteristics and any set of machine architectures. A DHSMS consists of a number of modules, each one in turn contains various components. The basic function of a DHSMS is to select a proper set of modules to meet the computing needs for an application. Each module varies in its functional capability and complexity. These modules are discussed later in the following section.

A DHSMS manages the resources and application, and tries to satisfy their processing requirements, such as on-line, off-line, regularly-processed, etc., by making management decisions regarding their scheduling/mapping. A conceptual viable architecture of DHSMS is shown in Fig. 5. It consists of 7 modules, namely; Core, Distributed Operating System (DOS), Task Analyzer, Task Coordinator, Code Profiler and Intermediate Graph Generator, Analytical Benchmark, and Code Graph Generator (CGG). Each module takes a certain set of inputs and generates appropriate outputs. We now describe the detailed functionality of each module.

3.1 CORE

To satisfy the processing requirements of various applications, this module selects a proper set of participating components from various other modules and determines the degree of accuracy and complexity of arriving at a scheduling/mapping decision. By selecting such components, it satisfies the task management objective which is to minimize the

average **total** execution time of **an** application. For this purpose, it generates a list of choices for a specific processing requirement. Dynamic programming techniques or Look-up tables can be used to handle this problem.

By implementing a Core as a module, that is independent of a DOS, existing **DOS's** can be **integrated** into a DHSMS. Examples of such module are Cronus Kernal, V-kernal, etc. **This** module also allows new local operating systems to be integrated into DHSMS without changing the local system or DHSMS itself.

3.2 DOS

DOS is the actual administrator, that manages resources and enables engagement of needed **components**. It performs many important functions, such as **supporting** communication among machines, maintaining service-level protocol structures; including data type **conversion**, and handling some standard services such as managing file, directories, etc. **Most** of the existing classes of **DOS's** can be used for a DHSMS, such as Intergrated Systems, Object-oriented Systems, Sever Pool Model Based Systems etc.

3.3 Task Analyzer

This is one of the key modules in DHSMS. It accepts user applications in the form of source programs **and** converts them into graphical forms, such as TFG's or TIG's. These graphs **are** subsequently processed by other modules such as the Code **Profiler** and Intermediate Graph Generator, the Analytical Benchmarker, the Code Graph Generator, and the **Task** Scheduler. To resolve the problem of heterogeneity in **programming** languages, we assume that there exists a standard graphical model of a program **that** helps in generating TFG's or TIG's. One such possible graphical 'language" is Intermediate Form

1 (IF1). It is an acyclic graphical language, which can be used to **represent** the flow of execution of a code [9]. An IF1 type of representation can be used to **estimate** the computation time and the communication overhead for the tasks present in the application at the **compile** time. For this purpose, we need some sort of application analyzing tool as a part of the compiler. One such tool has been implemented in **Parallel** Assessment Window System (PAWS) [9], which can be used.

Accordingly, the Task-Analyzer is composed of two components, **namely**: Task-Preprocessor and TFG (TIG) Generator. It is the Task-preprocessor **that** converts an application into a graphical language. The TFG(TIG) Generator is the **mentioned** application analyzing tool.

3.4 Code Profiler and Intermediate Graph Generator

The main objective of this module is to implement a code-profiling **function** and generate CPV's for TFG's (TIG's). It accepts a TFG (TIG) from the Task Analyzer and generates an ICFG (ICIG) by assigning the CPV's, that is \vec{V}_i 's, to each node of TFG (TIG). A CPV is generated based on the subset of USC as explained in **Section 2**. After **associating** such a vector with each code of TFG (TIG), the resulting ICFG (ICIG) can then be used for estimating the execution of time of each code of the **original** application.

3.5 Analytical Benchmark

This **module** is composed of a computation benchmarker and 1/0 **benchmarker**. A computation benchmark estimates the performance (speedup) of every machine present in the DHSS, **as** described in Section 2. For providing information on 1/0 benchmarking, it

uses I/O performance profiles which can be stored in Look-Up tables. Upon requests from the: CGG, it provides benchmark values of a code and I/O data transfer profiles for the selected machines in the DHSS.

3.6 Code Graph Generator

The output of the Code Profiler and Intermediate Graph Generator is an ICFG (ICIG) which is accepted by this module in order to produce a CFG (CIG) by assigning an estimated execution time vector \vec{E}_i to each code of the ICFG (ICIG). As mentioned earlier in Section 2, such an estimation is obtained by combining a CPV (\vec{V}_i) and analytical benchmark vector (ABV) \vec{B}^j . Also the estimated I/O overhead vector, \vec{D}_j , is assigned to each link. Both \vec{B}^j and \vec{D}_j are obtained from the Analytical Benchmark. The final CFG (CIG) contains sufficient information about the estimated execution and I/O performance of the application on the machines of the DHSS. These estimates are then used by the Task Coordinator to make the scheduling/mapping decision.

3.7 Task Coordinator

As indicated above, the purpose of this module is to make scheduling/mapping decisions for applications represented as a CFG or a CIG which is the produced by Code Graph Generator. By using the values of \vec{E}_i 's and \vec{D}_j 's associated in the graph, tasks are assigned to various machines in a manner so as to optimize some cost functions, which is generally the total execution time of applications and involves elements of \vec{E}_i 's and \vec{D}_j 's. Since, we are dealing with two models, CFG and CIG, scheduling is more appropriate for a CFG, while mapping is used for a CIG. We now briefly describe the two components of this module, namely; the Scheduler and the Mapper.

Scheduler consists of a set of scheduling algorithms with varying complexity and

accuracy in scheduling decision. Most of scheduling algorithms used in homogeneous systems can be modified to handle DHSS scheduling environment. As mentioned above, the criteria for scheduling is to minimize the total execution time of an application. We now briefly describe a general formulation of a possible cost function. This function is based on the assumption that computation and the I/O are done in a sequence without any overlapping and data conversion only can start once data transfer is completed (this assumption will not be valid when network caching is employed, as discussed later in the section). The estimated execution time p_i for a code i on a machine j is given as:

$$p_i = e_i + d_j$$

where e_i represents the estimated execution time of code i on the machine model j and d_j is the I/O overhead calculated by adding I/O latency with some data conversion overhead associated with that machine. Then the total cost of running an application, C_{Total} , is the total execution time of a TFG and can be given by the length of the critical path in the TFG.

Let \mathcal{R} be the set of all the paths from START to STOP of the application graph. The total execution time including computation time and I/O overhead, therefore, becomes the total sum of p_i along the critical path. In other words,

$$C_{Total} = \max_{\mathcal{R}} \sum_{i \in \mathcal{R}} (p_i)$$

The objective of the Scheduler is to minimize C_{Total} by matching each code with a suitable machine from the pool of available machines. Since, there is a limited number of machines available, intelligent assignment for the best performance is required.

For scheduling, the performance of the I/O subsystems of these machines must also be weighed, as expressed by the cost, C_{Total} . The existence of precedence relationships among codes in CFG imposes restriction on the order the way codes **should** be executed. Such ordering is provided by various paths of the CFG. Various critical paths need to be evaluated to finalize the scheduling decision.

For a CIG, a similar cost function can be formulated. The problem then becomes that of mapping, rather than scheduling, which is handled by the Mapper.

As can be noted, minimizing C_{Total} is basically a dual optimization problem that requires **not** only the best match of codes with machines but also **requires** minimization of the I/O overhead for assuring a fast data exchange among machines. Such computation and I/O bottlenecks in a critical path of a CFG (CIG) needs to be **identified** and eliminated by assigning them to the "**most** suitable" machines; a problem **which** is NP-Hard [1]. Various heuristics approaches to handle scheduling and mapping **can** be used. In this paper **we** do not make any attempt to propose any new algorithm, rather we describe the **concept** of "network caching" and discuss how the overall **scheduling/mapping** problem can be handled more efficiently by utilizing network resources in conjunction with the Task Coordinator module of the DHSMS.

3.8 Network Caching

Our objective is to propose a mechanism for utilizing underlying network resources, **especially** its buffering capability at its various nodes in order to carry out execution of **applications** efficiently. These buffers can be used to cache data **which** is exchanged among machines during the life of the CFG (CIG). We expect that data caching among **machines** can compensate I/O bottlenecks and can reduce the data exchange and **con-**

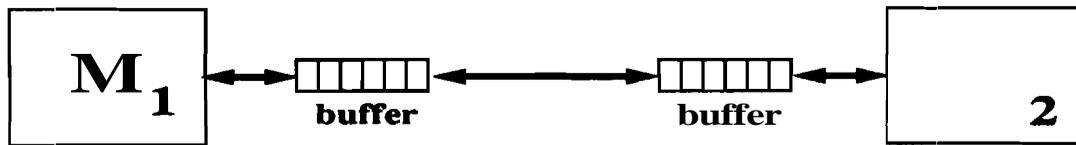


Figure 6: Data Buffering in Network

version overhead. For this purpose, fast buffers can be provided at **each** node in the network. Since, the network may be operating at extremely high rate (in multi Giga bits/sec range), we can view these buffers as a large memory with fast access. Once two machines need to communicate, at the time a CFG/CIG is scheduled, some amount of these buffers can be set-aside and used by these machines. This process for two machines is illustrated in Fig. 6. For example, when machine M_1 accesses data from the I/O subsystem of machine M_2 , some appropriate additional data can be brought out of M_2 's I/O subsystem and can be stored at intermediate nodes after converting it into a format suitable for M_1 . Various existing data caching algorithms can be used for this purpose. Similarly, when M_2 retrieves data from M_1 , the same caching process can be implemented. The size of the cache required between two machines depends on the I/O performance of these machines, which depends on the amount of data transferred between them. Such size requirement can be estimated from the elements of the COV's in the CFG (CIG). We assume that the Task Coordinator can generate such requirements. As mentioned earlier, the scheduling/mapping problem can then be formulated with reduced complexity. Using network caching, we now describe how this can be achieved.

- Starting with the CFG/CIG, the Task Coordinator carries out its scheduling/mapping decision, based only on the estimated execution time vector E_i 's. That is, only computation time estimation is used in the cost function to find the best matched machines; no communication cost needs to be used. Equivalently, we can modify CFG (CIG) by dropping COV's. Any heuristic algorithm, such as the one given in [1], can be used for scheduling/mapping.
- Once the machines are selected, the corresponding d_i 's of COV's are evaluated in order to find the I/O performance profiles for the selected machines that correspond

to the communication costs associated with the links in CFG (CIG). Such an **e-valuation** can provide the total buffer size required to implement sufficient cache memory in order to gain enough "delay compensation" to offset the I/O overhead. It is known that the performance of an I/O subsystem can be improved by increasing the size of the cache in the system. An appropriate relation between the size of the network cache and the value of d_i 's needs to be explored. The interaction between the Task Coordinator and some network resource manager also needs to be investigated for this purpose.

Fig. 7 summarizes the overall sequence of processing of an application through various modules of DHSMS.

4 An Experimental Platform for DHSMS

Currently, we are in the process of developing a DHSS platform by interconnecting a **MasPar**, an **nCube** and a **4-processor** system over **TeraNet**, which is a **high speed** optical network, that operates in **1 Gigabit/sec**. The system is being implemented in the **Parallel Processing** Laboratory of Purdue University. The objective of this platform is to provide a facility to test and evaluate various DHSMS related concepts and results, similar to the ones presented in this paper. Specifically, we are in the process of developing a **Task Analyzer**, a **Code Profiler** and **Intermediate Graph Generator**, and an **Analytical Benchmark**. For this purpose, we are planning to use the **Parallel Assessment Window System (PAWS)**, which allows us to assess the performance of various supercomputers and provides a ranking of various machines for a given application. The **major** components of PAWS are shown in Figure 8. This system has a number of capabilities which are needed

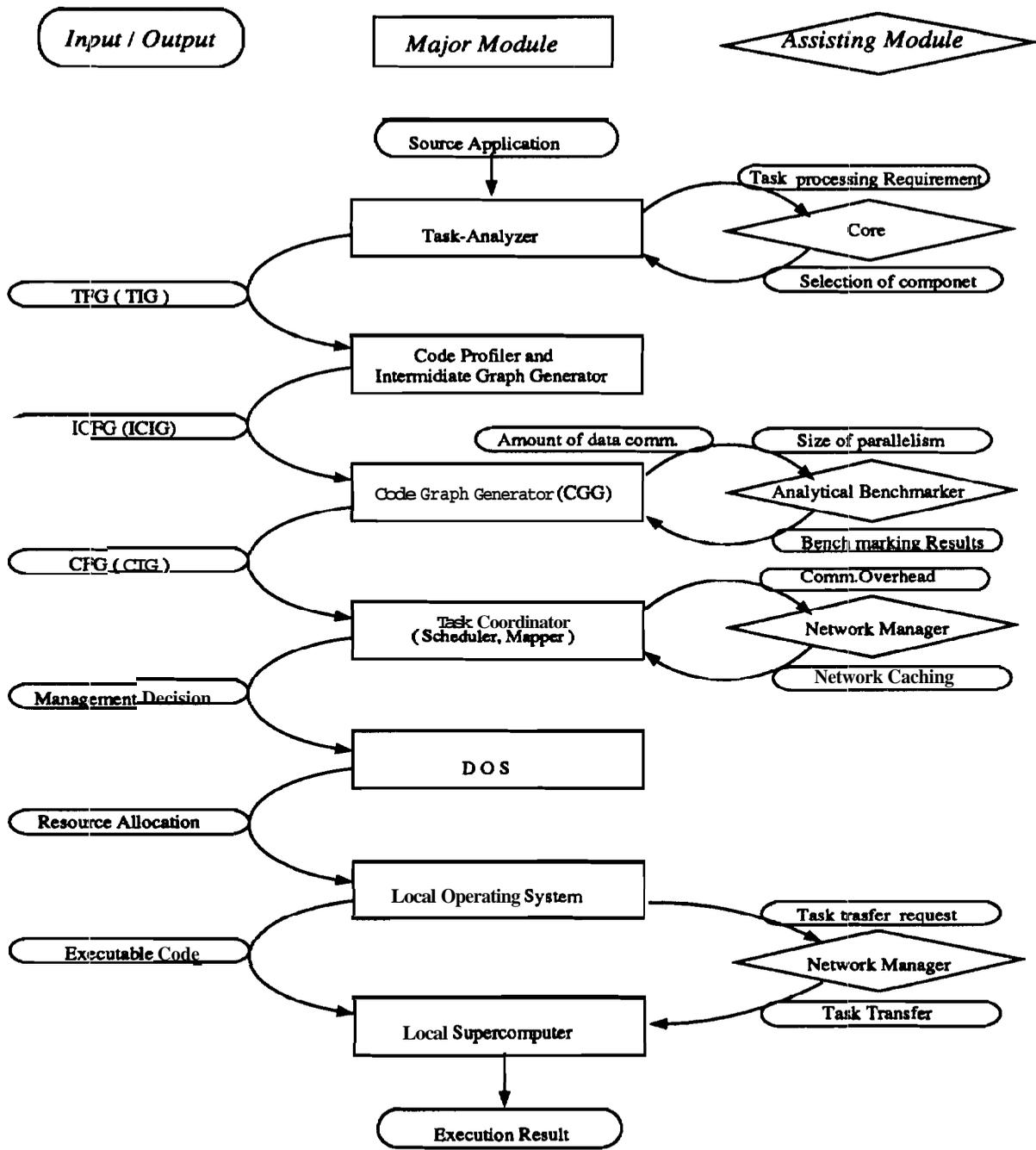


Figure 7: Processing Flow of DHSMS and Interaction among modules

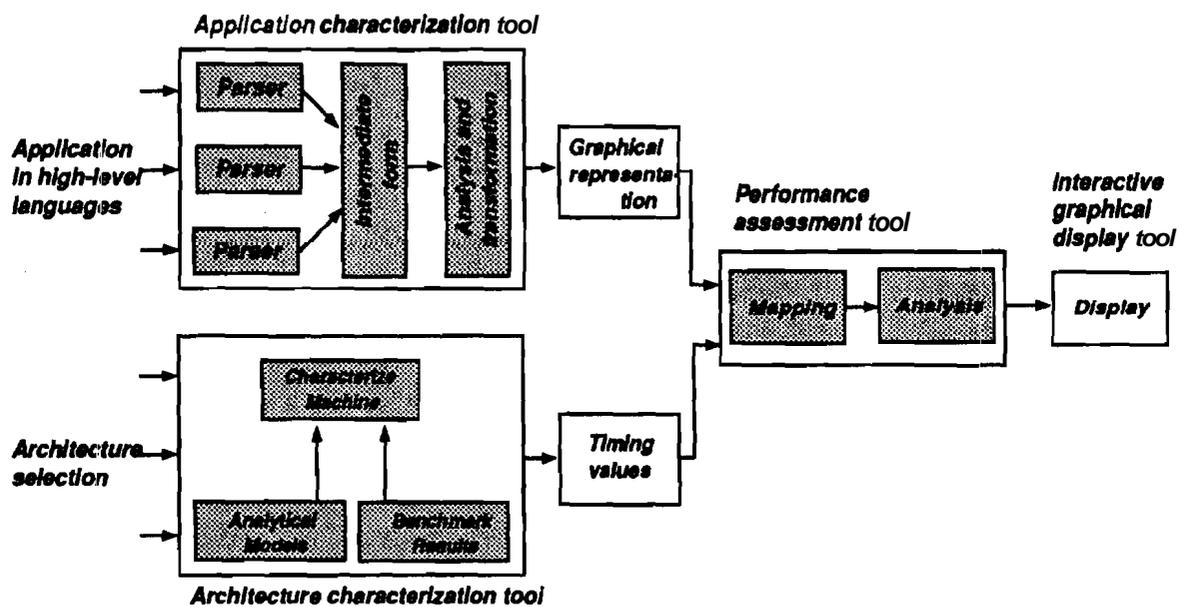


Figure 8: Architecture of PAWS

for the proposed DHSMS. Specifically, we are planning to use its following features:

- It can generate a machine independent graphical representation of an application written in a high level language, which is IF1. This representation can be easily transformed to generate an equivalent TFG (TIG).
- It can also simulate execution of a code for a parallel machine, which can provide approximate benchmark results, although exact benchmarks can be obtained by explicitly running codes on the machine.

Recently, we have proposed a mapping algorithm for heterogeneous systems [1]. We are planning to use a generalized version of this algorithm that is suitable for a DHSS environment by incorporating code profiling and benchmarking information.

5 Conclusion

Presented in this paper is a general framework for a DHSMS, for which we have proposed an architecture-dependent code profiling and benchmarking scheme. The proposed methodology incorporates both computational and I/O overheads associated with applications. We have also described how network caching can help scheduling applications in a DHSS.

References

- [1] N. Bowen, C. N. Nicolau, and A. Ghafoor, "On the Management Problem of Arbitrary Process System to Heterogeneous Distributed Computer Systems," *IEEE Transactions on Computers*, Vol. 41, No. 4, March 1992, pp: 257-273.
- [2] T. M. Conte and W. W. Hwu, "Benchmark Characterization," *IEEE Computer*, Vol. 24, No. 1, January 1991, pp: 48-56.

- [3] Z. Cvetanovic, E. G. Freedman and C. Nofsinger, "Perfect Benchmarks tm Decomposition and Performance on VAX tm Multiprocessors" Proc. of Supercomputing '90, New York, November 12-16 1990, pp: 455-464.
- [4] Z. Cvetanovic, "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems," IEEE Transactions on Computers, Vol. 36, No. 4, April 1987, pp: 421-432.
- [5] 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.
- [6] L. Stapleton, " Meet the Metacomputer: Where the Network Is the Computer," *Supercomputing* Review, Vol. 4, No. 3, March 1991, pp: 42-44.
- [7] " Gigabit Network Testbeds" IEEE Computer, Vol. 23, No. 9, September 1990, pp: 77-80.
- [8] D. Menasce, " Scheduling Tasks in Heterogeneous Systems," *Technical* Report, Dept. of Computer Sc., University of Maryland, College Park, 1990.
- [9] D. Pease, A. Ghafoor, et al., "PAWS: A Performance Evaluation Tool for Parallel Computing Systems," IEEE Computer, Vol. 24, No. 1, January 1991, pp: 18-29.
- [10] K. A. Robbins, and S. Robbins, "Dynamic Behavior of Memory Reference Streams for the Perfect Club Benchmarks," Proc. International Conf. on Parallel Processing, Chicago, August 1992, pp: 148-152.
- [11] J. E. Smith, " Characterizing Computer Performance with a Single Number" Communications of the ACM, Vol. 31, No. 3, October 1988, pp: 1202-1206
- [12] M. Wang, S. Kim, M. A. Nichols, R. F. Freund, H. J. Siegel, and. W. G. Nation, "Augmenting the Optimal Selection Theory for Superconcurrency," Proc. of the Workshop on Heterogeneous Processing, March 1992, pp: 13-22.