Syracuse University

SURFACE

1996

Compile-Time Performance Prediction of HPF/Fortran 90D

Manish Parashar
University of Texas at Austin

Salim Hariri
Syracuse University

Recommended Citation

Compile-Time Performance Prediction of HPF/Fortran 90D

Manish Parashar

Department of Computer Sciences

University of Texas at Austin

Austin, TX 78712-1081

parashar@cs.utexas.edu

Salim Hariri

Department of Computer Engineering

Syracuse University

Syracuse, NY 13244-4100

hariri@fruit.ece.syr.edu

Contents

Compile-Time Performance Prediction of HPF/Fortran 90D

Manish Parashar

Department of Computer Sciences

University of Texas at Austin

Austin, TX 78712-1081

parashar@cs.utexas.edu

Salim Hariri

Department of Computer Engineering

Syracuse University

Syracuse, NY 13244-4100

hariri@fruit.ece.syr.edu

Abstract

In this paper we present an interpretive approach for accurate and cost-effective performance prediction in a high performance computing environment, and describe the design of a compile-time HPF/Fortran 90D performance prediction framework based on this approach. The performance prediction framework has been implemented as a part of the HPF/Fortran 90D application development environment that integrates it with a HPF/Fortran 90D compiler and a functional interpreter. The current implementation of the environment framework is targeted to the iPSC/860 hypercube multicomputer system. A set of benchmarking kernels and application codes have been used to validate the accuracy, utility, and usability of the performance prediction framework. The use of the framework for selecting appropriate HPF/Fortran 90D compiler directives, for application performance debugging and for experimentation with run-time and system parameters is demonstrated.

Keywords: *Performance experimentation & prediction, HPF/Fortran 90D application development, System & Application characterization.*

1 Introduction

Although currently available High Performance Computing (HPC) systems possess large computing capabilities, few existing applications are able to fully exploit this potential. The fact remains that development of efficient application software capable of exploiting available computing potential is non-trivial and is largely governed by the availability of sufficiently high-level languages, tools, and application development environments.

A key factor contributing to the complexity of parallel/distributed software development is the increased degrees of freedom that have to be resolved and tuned in such an environment. Typically, during the course of parallel/distributed software development, the developer is required to select between available algorithms for the particular application; between possible hardware configurations and amongst possible decompositions of the problem onto the selected hardware configuration; and between different communication and synchronization strategies. The set of reasonable alternatives that have to be evaluated is very large and selecting the best alternative among these is a formidable task. Consequently, evaluation
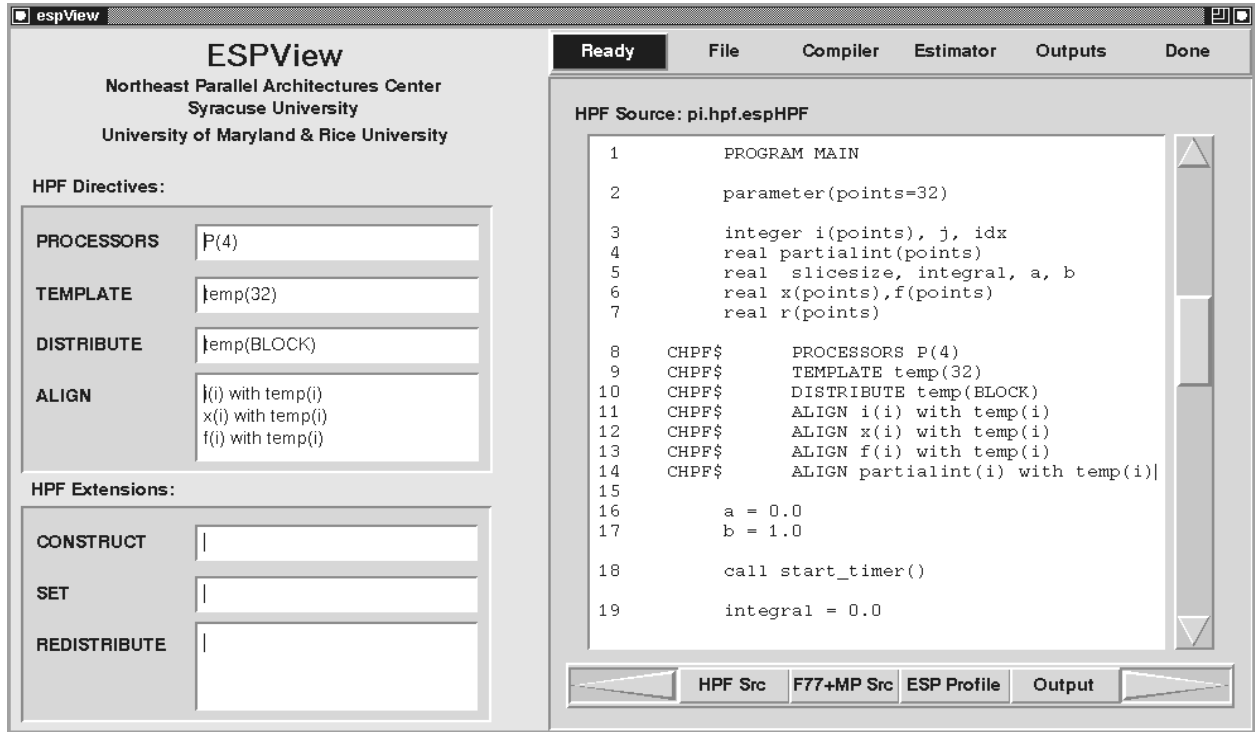
Figure 1: The HPF/Fortran 90D Application Development Environment

tools form a critical part of any software development environment. These tools, in symbiosis with other development tools, complete the feedback loop of the "develop-evaluate-tune" cycle.

In this paper we present a novel interpretive approach for accurate and cost-effective performance prediction in a high performance computing environment and describe the design of a source-driven HPF[1]/Fortran 90D performance prediction framework based on this approach. The interpretive approach defines a comprehensive characterization methodology which abstracts system and application components of the HPC environment. Interpretation techniques are then used to interpret performance of the abstracted application in terms of parameters exported by the abstracted system. System abstraction is performed off-line through a hierarchical decomposition of the computing system. Parameters required to abstract each component of this hierarchy can be generated independently, using existing techniques or system specifications. Application abstraction is achieved automatically at compile time.

The performance prediction framework has been implemented as a part of the HPF/Fortran 90D application development environment developed at the Northeast Parallel Architectures Center, Syracuse University (see Figure 1). The environment integrates a HPF/Fortran 90D compiler, a functional interpreter and the source based performance prediction tool and is supported by a graphical user interface. The current implementation of the environment is targeted to the iPSC/860 hypercube multicomputer system.

---

[1] High Performance Fortran

its behavior. Performance prediction is then achieved by interpreting the execution costs of the abstracted application in terms of the parameters exported by the abstracted system. The interpretive approach is illustrated in Figure 2 and is composed of the following four modules:

1. A system abstraction module that defines a comprehensive system characterization methodology capable of hierarchically abstracting a high performance computing system into a set of well defined parameters which represent its performance.

2. An application abstraction module that defines a comprehensive application characterization methodology capable of abstracting a high-level application description (source code) into a set of well defined parameters which represent its behavior.

3. An interpretation module that interprets performance of the abstracted application in terms of the parameters exported by the abstracted system.

4. An output module that communicates the estimated performance metrics.

A key feature of this approach is that each module is independent with respect to the other modules. Further, independence between individual modules is maintained throughout the characterization process and at every level of the resulting abstractions. As a consequence, abstraction and parameter generation for each module, and for individual units within the characterization of the module, can be performed separately using techniques or models best suited to that particular module or unit. This independence not only reduces the complexity of individual characterization models allowing them to be more accurate and tractable, but also supports reusability and easy experimentation. For example, when characterizing a multiprocessor system, each processing node can be characterized independently. Further, the parameters generated for the processing node can be reused in the characterization any system that has the same type of processors. Finally, experimentation with another type of processing node will only require the particular module to be changed. The four modules are briefly described below. A detailed discussion of the performance interpretation approach can be found in [1].

2.1 System Abstraction Module

Abstraction of a HPC system is performed by hierarchically decomposing the system to form a rooted tree structure called the *System Abstraction Graph (SAG)*. Each level of the SAG is composed of a set of *System Abstraction Unit's (SAU's)*. Each SAU abstracts a part of the entire system into a set of parameters representing its performance, and exports these parameters via a well defined interface. The interface can be generated independent of the rest of the system using evaluation techniques best suited to the particular unit (e.g. analytic, simulation, or specifications). The interface of an SAU consists of 4 components: (1) Processing Component (P), (2) Memory Component (M), (3) Communication/Synchronization Component (C/S), and (4) Input/Output Component (I/O). Figure 3 illustrates the system abstraction process using
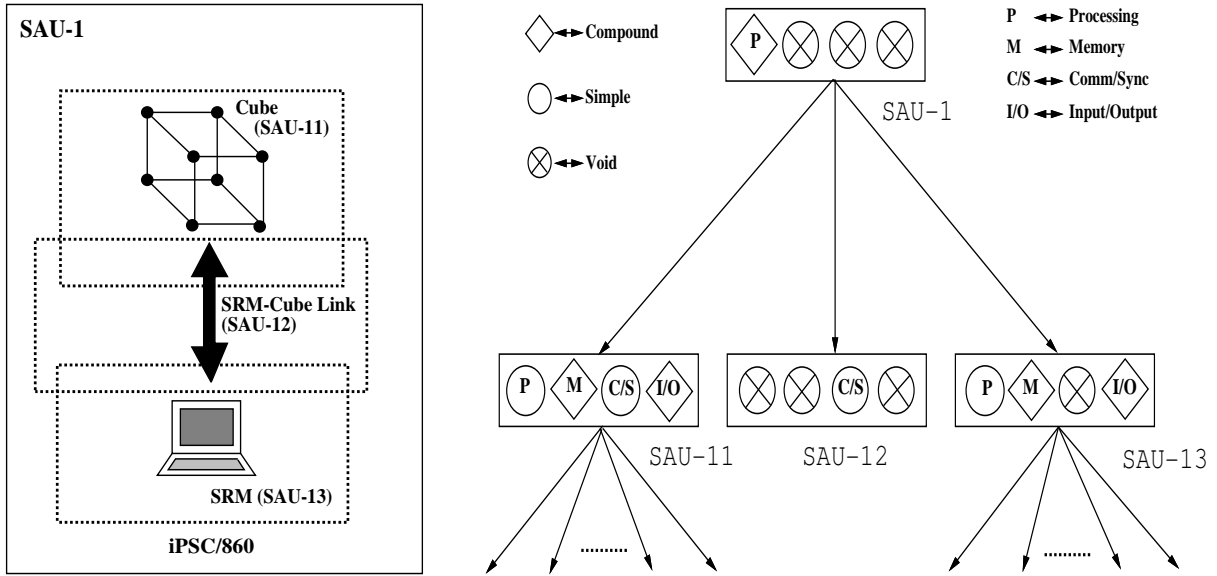
Figure 3: System Abstraction Process

the iPSC/860 system. At the highest level (SAU-1), the entire iPSC/860 system is represented as a single compound processing component. SAU-1 is then decomposed into SAU-11, SAU-12, and SAU-13 corresponding to the i860 cube, the interconnect between the System Resource Manager (SRM) and the cube, and the SRM or host respectively. Each SAU is composed of P, M C/S, and I/O components, each of which can be simple, compound or void. Compound components can be further decomposed. A component at any level is void if it is not applicable at that level (for example, SAU-12 has void P, M, and I/O components). Parameters exported by the i860 cube (SAU-11) are presented in Section 4.4. System characterization thus proceeds recursively down the system hierarchy, generating SAU's of finer granularity at each level.

2.2 Application Abstraction Module

Machine independent application abstraction is performed by recursively characterizing the application description into *Application Abstraction Units (AAU's)*. Each AAU represents a standard programming construct and parameterizes its behavior. An AAU can be either *Compound* or *Simple* depending on whether it can or cannot be further decomposed. Various classes of simple and compound AAU's are listed in Table 1. AAU's are combined so as to abstract the control structure of the application forming the *Application Abstraction Graph (AAG)*. The communication/synchronization structure of the application is superimposed onto the AAG by augmenting the graph with a set of edges corresponding to the communications or synchronization between AAU's. The structure generated after augmentation is called the *Synchronized Application Abstraction Graph (SAAG)* and is an abstracted application task graph. The machine specific filter then incorporates machine specific information (such as introduced compiler

```
Host Program
 N = 2
 DO I = 0,N-1
   Spawn Node I
 ENDDO
 Recv RESULTS
END


Node Program
 ME = MYNODE()
 CALC.....
 SyncSend (ME+1) MOD 2
 SyncRecv (ME-1+2) MOD 2
 IF ME EQ 0
   Send RESULTS
 ENDIF
END
```
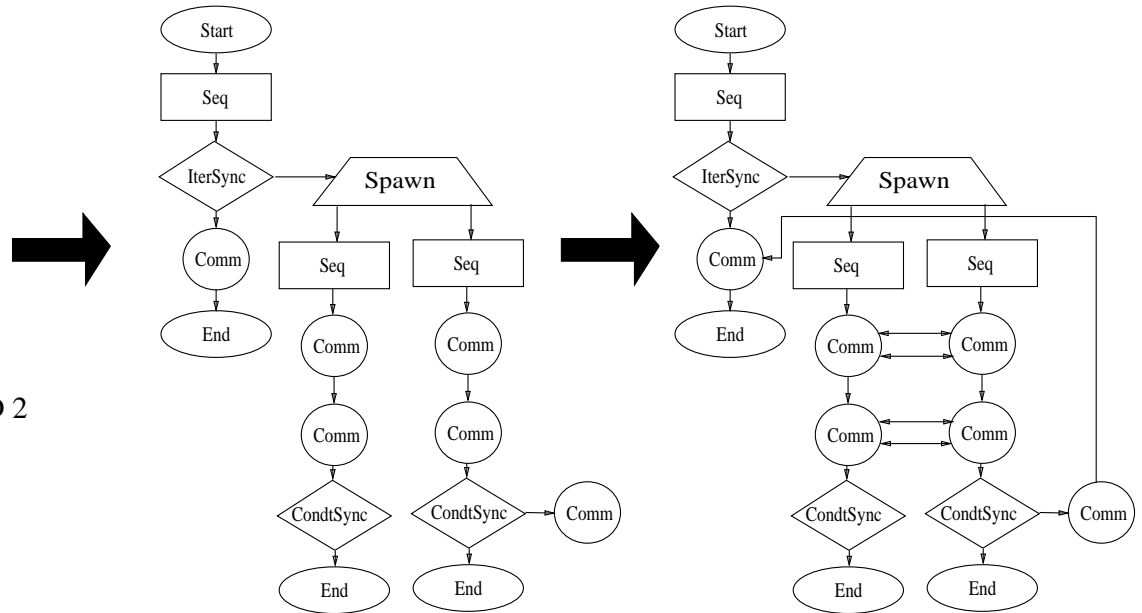
Application Description   **Application Abstraction Graph (AAG)**   **Synchronized AAG (SAAG)**

Figure 4: Application Abstraction Process

transformations/optimizations which are specific to the particular machine) into the SAAG based on the mapping that is being evaluated. Figure 4 illustrates the application abstraction process using a sample application description.

2.3 Interpretation Engine

The interpretation engine (or interpretation module) estimates performance by interpreting the abstracted application in terms of the performance parameters obtained via system abstraction. The interpretation module consists of two components; an interpretation function that interprets the performance of an individual AAU, and an interpretation algorithm that recursively applies the interpretation function to the SAAG to predict the performance of the corresponding application. Interpretation functions defined for each AAU class abstract its performance in terms of parameters exported by the SAU to which it is mapped. Functional interpretation techniques are used to resolve the values of variables that determine the flow of the application such as conditions and loop indices. Models and heuristics used to interpret communications/synchronizations, iterative and conditional flow control structures, accesses to the memory hierarchy, and user experimentation are described below. This description omits a lot of details for brevity; a more detailed discussion of these models and the complete set of interpretation functions can be found in [1].

| AAU Class	AAU Type	Description
Start AAU (Start)	Simple	marks the beginning of the application
End AAU (END)	Simple	represents the termination of an independent flow of control
Sequential AAU (Seq)	Simple	abstracts a set of contiguous statements containing only library functions, system routines, assignments and/or arithmetic/logical operations
Spawn AAU (Spawn)	Compound	abstracts a "fork" type statement generating independent flows of control
Iterative-Deterministic AAU (IterD)	Compound	abstracts an iterative flow control structure with deterministic execution characteristics and no comm/sync in its body
Iterative-Synchronized AAU (IterSync)	Compound	abstracts an iterative flow control structure with deterministic execution characteristics and at least one comm/sync in its body
Iterative-NonDeterministic (IterND)	Compound	abstracts a non-deterministic iterative flow control structure e.g. number of iterations depends on loop execution
Conditional-Deterministic (CondtD)	Compound	abstracts a conditional flow control structure with deterministic execution characteristics and no comm/sync in any of its bodies
Conditional-Synchronized (CondtSync)	Compound	abstracts a conditional flow control structure which contains a communication/synchronization in at least one of its bodies
Communication AAU (Comm)	Simple	abstracts statements involving explicit communication
Synchronization AAU (Sync)	Simple	abstracts statements involving explicit synchronization
Synchronized Sequential AAU (SyncSeq)	Simple	abstracts any Seq AAU which requires synchronization or communication e.g. a global reduction operation
Call AAU (Call)	Compound	abstracts invocations of user-defined functions or subroutines

Table 1: Application Characterization
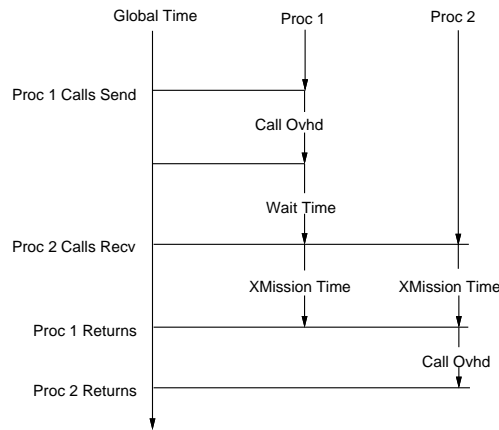
Figure 5: Interpretation Model for Communication/Synchronization AAU's

Modeling Communication/Synchronization: Communication or synchronization operations in the application are decomposed during interpretation into three components (as shown in Figure 5):

- **Call Overhead:** This represents fixed overheads associated with the operation.

- **Transmission Time:** This is the time required to actually transmit the message from the source to the destination.

- **Waiting Time:** Waiting time models overheads due to synchronizations, unavailable communications links, or unavailable communication buffers.

The contribution of each of the above components depends on the type of communication/synchronization and may differ for the sender and receiver. For example, in case of an asynchronous communication, the waiting time and transmission time components do not contribute to the execution time at the sender.

The waiting time component is determined using a global communication structure which maintains specifications and status of each communication/synchronization, and a global clock which is maintained by the interpretation algorithm. The global clock is used to timestamp each communication/synchronization call and message transmission, while the global communication structure stores information such as the time at which a particular message left the sender, or the current count at a synchronization barrier.

Modeling of Iterative Flow-Control Structures: The interpretation of an iterative flow control structure depends on its type. Typically, its execution time comprises three components: (1) loop setup overhead, (2) per iteration overhead, and (3) execution cost of the loop body.

In case of deterministic loops (IterD AAU) where the number of iterations is known and there are no communications or synchronizations in the loop body, the execution time is defined as

$$TExec_{IterD} = TOvhd_{Setup} + NumIters \times [TOvhd_{PerIter} + TExec_{Body}]$$

where $TExec$ and $TOvhd$ are estimated execution time and overhead time respectively.

In the case of the IterSync AAU, although the number of iterations are known, the loop body contains one or more communication or synchronization calls. This AAU cannot be interpreted as described above since it is necessary to identify the calling time of each instance of the communication/synchronization calls. In this case, the loop body is partitioned into blocks without communication/synchronization and the communication/synchronization calls themselves. The interpretation function for the entire AAU is then defined as a recursive equation such that the execution time of the current iteration is a function of the execution time of the previous iteration. Similarly, the calling and execution times of the communication/synchronization calls are also defined recursively. For example, consider a loop body that contains two communication calls calls (Comm$_1$ & Comm$_2$). Let Blk$_1$ represent the block before Comm$_1$ and Blk$_2$ represent the block between Comm$_1$ and Comm$_2$. If the loop starts execution at time T, the calling times ($TCall$) for the first iteration are:

$$
\begin{aligned}
TCall_{IterSync}(1) &= T \\
TCall_{Comm_1}(1) &= TCall_{IterSync}(1) + TOvhd_{IterSync} + TExec_{Blk_1} \\
TCall_{Comm_2}(1) &= TCall_{IterSync}(1) + TOvhd_{IterSync} + TExec_{Blk_1} + TExec_{Comm_1}(1) + TExec_{Blk_2}
\end{aligned}
$$

And for the ith iteration

$$
\begin{aligned}
TCall_{IterSync}(i) &= TCall_{IterSync}(i-1) + TOvhd_{IterSync} + TExec_{Blk_1} + TExec_{Comm_1}(i-1) + TExec_{Blk_2} \\
&\quad + TExec_{Comm_2}(i-1) \\
TCall_{Comm_1}(i) &= TCall_{IterSync}(i) + TOvhd_{IterSync} + TExec_{Blk_1} \\
TCall_{Comm_2}(i) &= TCall_{IterSync}(i) + TOvhd_{IterSync} + TExec_{Blk_1} + TExec_{Comm_1}(i) + TExec_{Blk_2}
\end{aligned}
$$

The final case is a non-deterministic iterative structure (IterND) where the number of iterations or the execution of the loop body are not known. For example the number of iterations may depend on the execution of the loop body as in the *while* loop, or the execution of the loop body varies from iteration to iteration. In this case performance is predicted by unrolling the iterations using functional interpretation and interpreting the performance of each iteration sequentially.

Modeling of Conditional Flow-Control Structures: The execution time for a conditional flow control structure is broken down into three components: (1) the overhead associated with each condition tested (i.e. every "if", "elseif", etc.), (2) an additional overhead for the branch associated with a true condition, and (3) the time required to execute the body associated with the true condition. The interpretation function for the conditional AAU is a weighted sum of the interpreted performances of each of its branches; the weights evaluate to 1 or 0 during interpretation depending on whether the branch is taken or not. Functional interpretation is used to resolve the execution flow. Modeling of CondtD and CondtSync AAU's is similar to the corresponding iterative AAU's described above.

Modeling Access to the Memory Hierarchy: Access to the memory hierarchy of a computing element is modeled using heuristics based on the access patterns in the application description and the physical structure of the hierarchy. In the current implementation, application access patterns are approximated during interpretation by maintaining an access count and a detected miss count at the program level and by associating with each program variable, a local access count, the last access offset (in case of arrays), and values of both program level counters at the last access. A simple heuristic model uses these counts and the size of the cache block, its associativity and the replacement algorithm, to estimate cache misses for each AAU. This model is computationally efficient and provides the required accuracy as can be seen from the results that presented in Section 5.

Modeling Communication-Computation Overlaps: Overlap between communication and computation is accounted for during interpretation, as a fraction of the communication cost; i.e. if a communication takes time t_{comm} and $f_{overlap}$ is the fraction of this time overlapped with computation, then the execution time of the Comm AAU is weighted by the factor $(1 - f_{overlap})$; i.e.

$$
t_{AAU_{Comm}} = (1 - f_{overlap}) \times t_{comm}
$$

The $f_{overlap}$ factor could be a typical (or explicitly defined) value defined for the system. Alternately the user can define this factor for the particular application or could experiment with different values.

Supporting User Experimentation: The interpretation engine provides support for two types of user experimentation:

- Experimentation with run-time situations, e.g. computation and communications loads.

- Experimentation with system parameters, e.g. processing capability, memory size, communication channel bandwidth.

The effects of each experiment on application performance is modeled by abstracting its effect on the parameters exported by the system and application modules and setting their values accordingly. Heuristics are used to perform this abstraction. For example, the effect of increased network load on a particular communication channel is modeled by decreasing the effective available bandwidth on that channel. An appropriate scaling factor is then defined which is used to scale the parameters exported by the C/S component associated with the communication channel. Similarly, doubling the bandwidth effectively decreases the transmission time over the channel; while increasing the cache size will reflect on the miss rate.

2.4   Output Module

The output module provides an interactive interface through which the user can access estimated performance statistics. The user has the option of selecting the type of information and the level at which the information is to be displayed. Available information includes cumulative execution times, the communication time/computation time breakup, existing overheads and wait times. This information can be obtained for an individual AAU, cumulatively for a branch of the AAG (i.e. sub-AAG), or for the entire AAG.

3   An Overview of HPF/Fortran 90D

High Performance Fortran (HPF) [2] is based on the research language Fortran 90D developed jointly by Syracuse University and Rice University and has the overriding goal to produce a dialect of Fortran that can be used on a variety of parallel machines, providing portable, high-level expression to data parallel algorithms. The idea behind HPF (and Fortran 90D) is to develop a minimal set of extensions to Fortran 90 to support the data parallel programming model. The incorporated extensions provide a means for explicit expression of parallelism and data mapping. These extensions include compiler directives which are used to advise the compiler on how data objects should be assigned to processor memories, and new language features like the *forall* statement and construct.

REAL, ARRAY(5,4) :: A
REAL, ARRAY(5,6) :: B
CHPF$ PROCESSORS PROC(4)
CHPF$ TEMPLATE TMPL(8,8)
CHPF$ DISTRIBUTE TMPL(*,BLOCK)
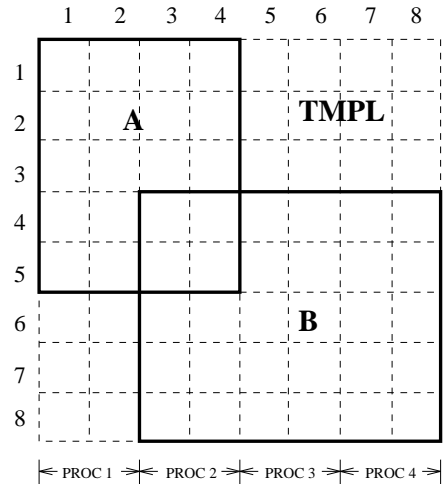CHPF$ ALIGN A(I,J) WITH TMPL(I,J)
CHPF$ ALIGN B(I,J) WITH TMPL(I+3,J+2)

Figure 6: HPF/Fortran 90D Directives

HPF/Fortran 90D adopts a two level mapping using the PROCESSORS, ALIGN, DISTRIBUTE, and TEMPLATE directives to map data objects to abstract processors. The data objects (typically array elements) are first *aligned* with an abstract index space called a *template*. The template is then *distributed* onto a rectilinear arrangement of abstract *processors*. The mapping of abstract processors to physical processors is implementation dependent. Data objects not explicitly distributed are mapped according to an implementation dependent default distribution (e.g. replication). Supported distributions types include BLOCK and CYCLIC. Use of the directives is shown in Figure 6.

Our current implementation of the HPF/Fortran 90D compiler and performance prediction framework supports a formally defined subset of HPF. The term HPF/Fortran 90D in the rest of this document refers to this subset.

4   Design of the HPF/Fortran 90D Performance Prediction Framework

The HPF/Fortran 90D performance prediction framework is based on the HPF source-to-source compiler technology [3] which translates HPF into loosely synchronous, SPMD (single program, multiple data) Fortran 77 + Message-Passing codes. It uses this technology in conjunction with the performance interpretation model to provide performance estimates for HPF/Fortran 90D applications on a distributed memory MIMD multicomputer. HPF/Fortran 90D performance prediction is performed in two phases: Phase 1 uses HPF compilation technology to produce a SPMD program structure consisting of Fortran 77 plus calls to run-time routines. Phase 2 then uses the interpretation approach to abstract and interpret the performance of the application. These two phases are described below:

4.1 Phase 1 - Compilation

The compilation phase uses the same front-end as the HPF/Fortran 90D compiler. Given a syntactically correct HPF/Fortran 90D program, phase 1 parses the program to generate a parse tree and transforms array assignment and *where* statements to equivalent *forall* statements. Compiler directives are used to partition the data and computation among the processors and parallel constructs in the program are converted into loops or nested loops. Required communication are identified and appropriate communication calls are inserted. The output of this phase is a loosely synchronous SPMD program structure consisting of alternating phases of local computation and global communication.

4.2 Phase 2 - Interpretation

Phase 2 is implemented as a sequence of parses: (1) The abstraction parse generates the application abstraction graph (AAG) and synchronized application abstraction graph (SAAG); (2) The interpretation parse performs the actual interpretation using the interpretation algorithm; and (3) The output parse generates the required performance metrics.

Abstraction Parse: The abstraction parse intercepts the SPMD program structure produced in phase 1 and abstracts its execution and communication structures to generate the corresponding AAG and SAAG (as defined in Section 2). A communication table (global communication structure) is generated to store the specifications and status of each communication/synchronization.

The compiler symbol table is extended in this parse by tagging all variables that are critical (a critical variable being defined as a variable whose value effects the flow of execution, e.g. a loop limit). Critical variables are then resolved using functional interpretation by tracing their definition paths. If this is not possible, or if they are external inputs, the user is prompted for their values. If a critical variable is defined within an iterative structure, the user has the option of either explicitly defining the value of that variable or instructing the system to unroll the loop so as to compute its value. Access information required to model accesses to the memory hierarchy is abstracted from the input program structure in this parse and stored in the extended symbol table.

The final task of the abstraction parse is the clustering of consecutive Seq AAU's into a single AAU. The granularity of clustering can be specified by the user; the tradeoff here being estimation time versus estimation accuracy. At the finest level, each Seq AAU abstracts a single statement of the application description.

Interpretation Parse: The interpretation parse performs the actual performance interpretation using the interpretation model described above. For each AAU in the SAAG, the corresponding interpretation function is used to generate performance measures associated with it. Metrics maintained at each AAU are its computation, communication and overheads times, and the value of the global clock. In addition,

metrics specific to each AAU type (e.g. wait and transmission times for a Comm AAU) are also maintained. Cumulative metrics are maintained for the entire SAAG, and for each compound AAU. The interpretation parse has provisions to take into consideration a set of system compiler optimizations (for the generated Fortran 77 + Message Passing code) such as loop re-ordering and inline expansion. These can be turned on or off by the user.

Output Parse The final parse communicates estimated performance metrics to the user. The output interface provides three types of outputs. The first type is a generic performance profile of the entire application broken up into its communication, computation and overhead components. Similar measures for each individual AAU and for sub-graphs of the AAG are also available. The second form of output allows the user to query the system for the metrics associated with a particular line (or a set of lines) of the application description. Finally, the system can generate an interpretation trace which can be used as input to a performance visualization package. The user can then use the capabilities provided by the package to analyze the performance of the application.

4.3   Abstraction & Interpretation HPF/Fortran 90D Parallel Constructs

The abstraction/interpretation of the HPF/Fortran 90D parallel constructs i.e. *forall*, array assignment and *where* is described below:

forall Statement: The *forall* statement generalizes array assignments to handle new shapes of arrays by specifying them in terms of array elements or sections. The element array may be masked with a scalar logical expression. Its semantics are an assignment to each element or section (for which the mask expression evaluates true) with all the right-hand sides being evaluated before any left-hand sides are assigned. The order of iteration over the elements is not fixed. Examples of its use are:

$$forall \ (I = 1 : N, J = 1 : N) \ P(I, J) \ = \ Q(I - 1, J - 1)$$

$$forall \ (I = 1 : N, J = 1 : N, Q(I, J).NE.0.0) \ P(I, J) \ = \ 1.0/Q(I, J)$$

Phase 1 translates the *forall* statement into a three level structure consisting of a collective communication level, a local computation level and another collective communication level, to be executed by each processor. This three level structure is based on the "owner computes rule". The processor that is assigned an iteration of the *forall* loop is responsible for computing the right-hand-side expression of the assignment statement while the processors that owns an array element used in the left-hand side or right-hand side of the assignment statement must communicate that element to the processor performing the computation. Consequently, the first communication level fetches off-processor data required by the computation level. Once this data has been gathered, computations are local. The final communication level writes calculated values to off-processors.

ADJUST_BOUNDS()

PACK_PARAMETERS()

GATHER_DATA (G)

forall (K=2:N-1,V (K) .GT. 0)

X (K+1) = X (K) + X(K-1)

Phase 1

DO K = LocalLB,LocalUB

IF (V (K) .GT. 0) THEN

X (K+1) = X (K) + G (K)

END IF

END DO
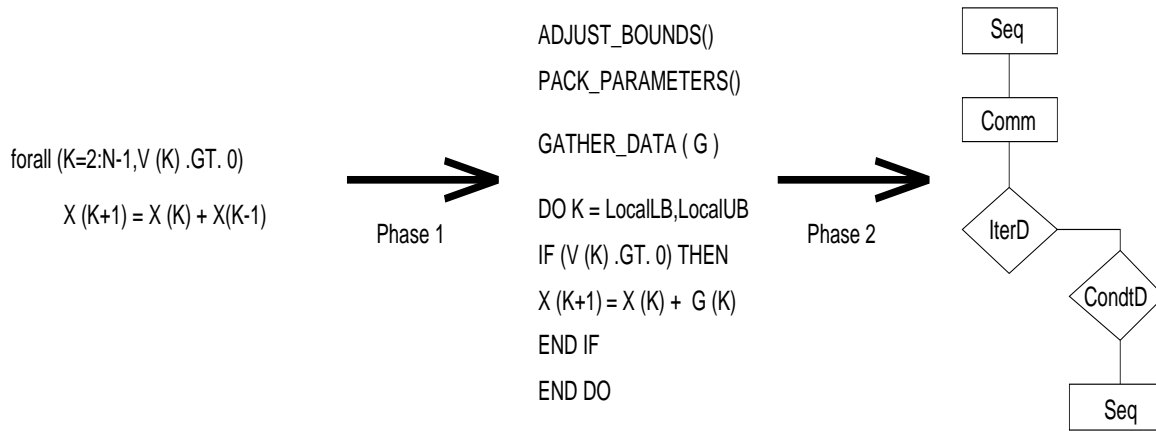
Phase 2

Seq

Comm

IterD

CondtD

Seq

Figure 7: Abstraction of the *forall* Statement

Phase 2 then generates a corresponding sub-AAG using the application abstraction model. The communication level translates into a Seq AAU corresponding to index translations and message packing performed, and a Comm/Sync AAU. The computation level generates an iterative AAU (IterD/IterND/IterSync) which may contain a conditional AAU (CondtD/CondtSync) (depending on whether a mask is specified). The abstraction of the *forall* statement is shown in Figure 7. In this example, the final communication phase is not required as no off-processor data needs to be written.

Array Assignment Statements: HPF/Fortran 90D array assignment statements allow entire arrays (or array sections) to be manipulated atomically, thereby enhancing the clarity and conciseness of the program and making parallelism explicit. Array assignments are special cases of the *forall* statement and are abstracted by first translating them into equivalent *forall* statements. The resultant *forall* statement is then interpreted as described above. The translation is illustrated by the following example:

$$A(l_1 : u_1 : s_1) \;=\; B(l_2 : u_2 : s_2)$$

translates to:

$$forall(i = l_1 : u_1 : s_1) \; A(i) \;=\; B(l_2 + ((i - l_1)/s_1) * s_2)$$

where Statement: Like the array assignment statement, the HPF/Fortran 90D *where* statement is also a special case of the *forall* statement and is handled in a similar way. The translation of the *where* statement into an equivalent *forall* is illustrated below:

$$where(C(l_3 : u_3 : s_3) \; .NE. \; 0.0) \; A(l_1 : u_1 : s_1) \;=\; B(l_2 : u_2 : s_2)$$

translates to:

$$forall(i = l_1 : u_1 : s_1, C(l_3 + ((i - l_1)/s_1) * s_3)) \; A(i) \;=\; B(l_2 + ((i - l_1)/s_1) * s_2)$$

Processing Component	Memory Component	Comm/Sync Component
Arithmetic Opers	**Memory Org**	**Specifications**
Integer/Float Add/Sub	Cache Size	Topology
Integer/Float Multiply	Cache Block Size	Routing Scheme
Integer Divide	Cache Replication Policy	Static Buffer Size
Float Divide	Cache Associativity	**Comm - Static Buffers**
Conv: Integer->Float	Cache Write Policy	Startup Ovhd
Conv: Float->Int	Main Memory Size	Transmission Time/byte
......	Main Memory Page Size	Per Hop Ovhd
Iterative Opers	Instruction Cache Size	Receive Ovhd
Per-Iteration Loop Ovhd	Instruction Cache Block Size	**Comm - Dynamic buffers**
Step Limit Ovhd	**Memory Hierarchy**	Startup Ovhd
......	Fetch/Fetch Miss Clks	Transmission Time/byte
Conditional Opers	Store/Store Miss Clks	Per Hop Ovhd
Condition Ovhd	**Main Memory**	Receive Ovhd
Branch Taken Ovhd	Main Memory Fetch (pipelined)	**Synchronization**
......	Main Memory Store (pipelined)	Sync Ovhd
Call Opers	**Access Ovhds**	**Group Communication**
Call Ovhd	TLB Miss	Broadcast Algorithm
Lib Chars	Read/Write Switch	Multicast Algorithm
abs()	......	Reduction
exp()		Shifts
......		......

Table 2: Abstraction of the iPSC/860 System

4.4   Abstraction of the iPSC/860 System

Abstraction of the iPSC/860 hypercube system to generate the corresponding SAG was performed off-line using a combination of assembly instruction counts, measured timings and system specifications. The processing and memory components were generated using system specification provided by the vendor, while iterative and conditional overheads were computed using instruction counts. The communication component was parameterized using benchmarking runs. These parameters abstracted both low-level primitives as well as the high-level collective communication library used by the compiler. Benchmarking runs were also used to parameterize the HPF parallel intrinsic library. The intrinsics included circular shift (*cshift*), shift to temporary (*tshift*), global sum operation (*sum*), global product operation (*product*), and the *maxloc* operation which returns the location of the maximum in a distributed array. Some of the parameters exported by each component of the i860 cube are summarized in Table 2. Sample values for these parameters can be found in [1]. Characterization of the SRM (System Resource Manager) and the communication channel connecting the SRM to i860 cube was performed in a similar manner.

5 Validation/Evaluation of the Interpretation Framework

In this section we present numerical results obtained using the current implementation of the HPF/Fortran 90D performance prediction framework. In addition to validating the viability of the interpretive approach, this section has the following objectives:

1. To validate the *accuracy* of the performance prediction framework for applications on a high performance computing system. The aim is to show that the predicted performance metrics are accurate enough to provide realistic information about the application performance and to be used as a basis for design tuning.

2. To demonstrate the application of the framework and the metrics generated to HPC application development. The results presented illustrate the utility of the framework for the following:

   - Application design and directive selection.
   - Experimentation with system and run-time parameters.
   - Application performance debugging.

3. To demonstrate the usability (ease of use) of the performance interpretation framework and its cost-effectiveness.

The high performance computing system used for the validation is an iPSC/860 hypercube connected to a 80386 based host processor. The particular configuration of the iPSC/860 consists of eight i860 nodes. Each node has a 4 KByte instruction cache, 8 KByte data cache and 8 MBytes of main memory. The node operates at a clock speed of 40 MHz and has a theoretical peak performance of 80 MFlop/s for single precision and 40 MFlop/s for double precision. The validation application set was selected from the NPAC[2] HPF/Fortran 90D Benchmark Suite. The suite consists of a set of benchmarking kernels and "real-life" applications and is designed to evaluate the efficiency of the HPF/Fortran 90D compiler and specifically, automatic data-mapping schemes. The selected application set includes kernels from standard benchmark sets like the Livermore Fortran Kernels and the Purdue Benchmark Set, as well as real computational problems. The applications are listed in Table 3.

5.1 Validating Accuracy of the Framework

Accuracy of the interpretive performance prediction framework is validated by comparing estimated execution times with actual measured times. For each application, the experiment consisted of varying the problem size and number of processing elements used. Measured timings represent an average taken over 1000 runs. The results obtained are summarized in Table 4. Error values listed (and plotted) are percentages of the measured time and represent maximum/minimum absolute errors over all problem sizes and

---

[2]Northeast Parallel Architectures Center

Name	Description
LFK 1	Hydro Fragment
LFK 2	ICCG Excerpt (Incomplete Cholesky - Conjugate Gradient)
LFK 3	Inner Product
LFK 9	Integrate Predictors
LFK 14	1-D PIC (Particle In Cell)
LFK 22	Planckian Distribution
PBS 1	Trapezoidal rule estimate of an integral of f(x)
PBS 2	Compute the value of $e^* = \sum_{i=1}^{n} \prod_{j=1}^{m} \left(1 + \frac{0.5}{-|i-j|+0.001}\right)$
PBS 3	Compute the value of $S = \sum_{i=1}^{n} \prod_{j=1}^{m} a_{ij}$
PBS 4	Compute the value of $R = \sum_{i=1}^{n} \frac{1}{x_i}$
PI	Approximation of PI by calculating the area under the curve using the n-point quadrature rule
N-Body	Newtonian gravitational n-body simulation
Finance	Parallel stock option pricing model
Laplace	Laplace solver based on Jacobi iterations

LFK = Livermore Fortran Kernel

PBS = Purdue Benchmarking Set

Table 3: Validation Application Set

system sizes. For example, the N-Body computation was performed for 16 to 4094 bodies on 1, 2, 4, and 8 nodes of the iPSC/860. The minimum absolute error between estimated and measured times was 0.09% of the measured time while the maximum absolute error was 5.9%. Plots for estimated and measured execution times are included in Appendix A.

The obtained results show that in the worst case, the interpreted performance is within 20% of the measured value, the best case error being less than 0.001%. The larger errors are produced by the benchmark kernels which have been specifically coded to task the compiler. Further, it was found that the interpreted performance typically lies within the variance of the measured times over the 1000 iterations. This indicates that the main contributors to the error are the tolerance of the timing routines and fluctuations in the system load. The objective of the predicted metrics is to serve either as the first-cut performance estimate of an application or as a relative performance measure to be used as a basis for design tuning. In either case, the interpreted performance is accurate enough to provide the required information.

5.2 Application of the Interpretive Framework to HPC Application Development

The application of the interpretive performance prediction framework to HPC application development is illustrated by validating its utility for the following: (1) selection of appropriate HPF/Fortran 90D directives based on the predicted performance, (2) experimentation with larger system configurations, varying system parameters, and with different run-time scenarios, and (3) analyzing different components

Name	Problem Sizes (data elements)	System Size (# procs)	Min Abs Error (%)	Max Abs Error (%)
LFK 1	128 - 4096	1 - 8	1.3%	10.2%
LFK 2	128 - 4096	1 - 8	2.5%	18.6%
LFK 3	128 - 4096	1 - 8	0.7%	7.2%
LFK 9	128 - 4096	1 - 8	0.3%	13.7%
LFK 14	128 - 4096	1 - 8	0.3%	13.8%
LFK 22	128 - 4096	1 - 8	1.4%	3.9%
PBS 1	128 - 4096	1 - 8	0.05%	7.9%
PBS 2	256 - 65536	1 - 8	0.6%	6.7%
PBS 3	256 - 65536	1 - 8	0.8%	9.5%
PBS 4	128 - 4096	1 - 8	0.2%	3.9%
PI	128 - 4096	1 - 8	0.00%	5.9%
N-Body	16 - 4096	1 - 8	0.09%	5.9%
Finance	32 - 512	1 - 8	1.1%	4.6%
Laplace (Blk,Blk)	16 - 256	1 - 8	0.2%	4.4%
Laplace (Blk,*)	16 - 256	1 - 8	0.6%	4.9%
Laplace (*,Blk)	16 - 256	1 - 8	0.1%	2.8%

Table 4: Accuracy of the Performance Prediction Framework

(BLOCK,BLOCK)	(BLOCK,*)	(*,BLOCK)
PROCESSORS PRC(2,2)	PROCESSORS PRC(4)	PROCESSORS PRC(4)
TEMPLATE TEMP(N,N)	TEMPLATE TEMP(N)	TEMPLATE TEMP(N)
DISTRIBUTE TEMP(BLOCK,BLOCK)	DISTRIBUTE TEMP(BLOCK)	DISTRIBUTE TEMP(BLOCK)
ALIGN A(i,j) with TEMP(i,j)	ALIGN A(i,*) with TEMP(i)	ALIGN A(*,j) with TEMP(j)
ALIGN B(i,j) with TEMP(i,j)	ALIGN B(i,*) with TEMP(i)	ALIGN B(*,j) with TEMP(j)
ALIGN C(i,j) with TEMP(i,j)	ALIGN C(i,*) with TEMP(i)	ALIGN C(*,j) with TEMP(j)
ALIGN D(i,j) with TEMP(i,j)	ALIGN D(i,*) with TEMP(i)	ALIGN D(*,j) with TEMP(j)

Table 5: Possible Distributions for the Laplace Solver Application

of the execution time and their distributions with respect to the application. The experiments performed are described below:

5.2.1 Appropriate Directive Selection

To demonstrate the utility of the interpretive framework in selecting HPF compiler directives we compare the performance of the Laplace solver for 3 different distributions (DISTRIBUTE directive) of the template, namely (BLOCK,BLOCK), (BLOCK,*) and (*,BLOCK), and corresponding alignments (ALIGN directive) of the data elements to the template. These three distributions (on 4 processors) are shown in Figure 8 and the corresponding HPF/Fortran 90D descriptions are listed in Table 5. Figures 9-12 compare the performance of each of the three cases for different system sizes using both, measured times and estimated times. These graphs can be used to select the best directives for a particular problem size and system
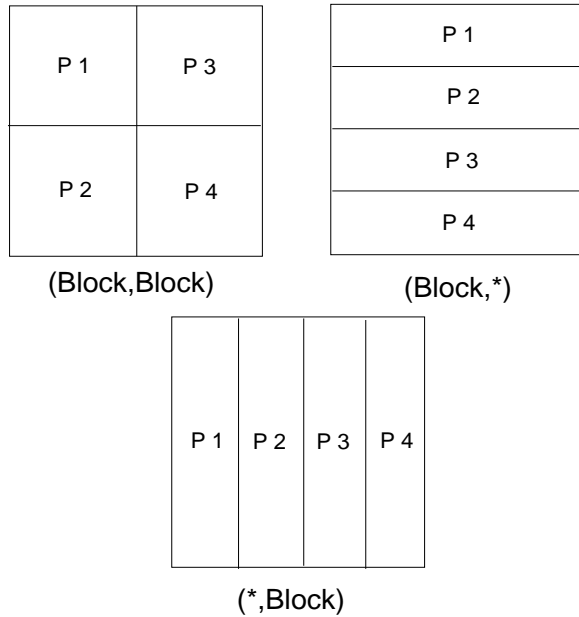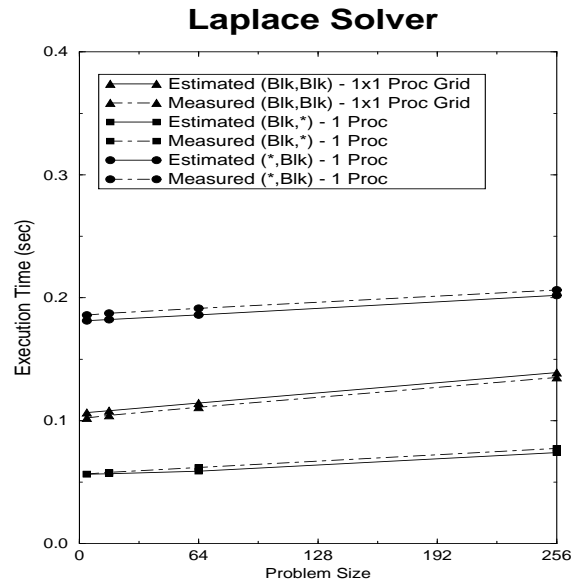
Figure 8: Laplace Solver - Data Distributions

Figure 9: Laplace Solver (1 Proc) - Estimated/Measured Times

configuration. For the Laplace solver, the (BLOCK,*) distribution is the appropriate choice. Further, since the maximum absolute error between the estimated and measured times is less than 1%, the directive selection can be accurately made using the interpretive framework. Using the interpretive framework is also significantly more cost-effective as will be demonstrated in Section 5.3.

In the above experiment, performance interpretation was source driven and can be automated. This exposes the utility of the framework as a basis for an intelligent compiler capable of selecting appropriate directives and data decompositions. Similarly, it can also enable such a compiler to select code optimizations such as the granularity of the computation phase per communication phase in the loosely synchronous computation model.

5.2.2   Experimentation with System/Run-Time Parameters

Results presented in this section demonstrate the use of the interpretive framework for evaluating the effects of different system and run-time parameters on the application performance. The following experiments were conducted:

Effect of Varying Processor Speed: In this experiment we evaluate the effect of increasing/decreasing the speed of each processor in the iPSC/860 system on application performance. The results are shown in Figure 13 for speeds 2 times (100% processor speed increase), 3 times (200% processor speed increase), and 4 times (300% processor speed increase) the i860 processor speed. Such an evaluation enables the developer to visualize how the application will perform on a faster (prospective) machine or alternately if
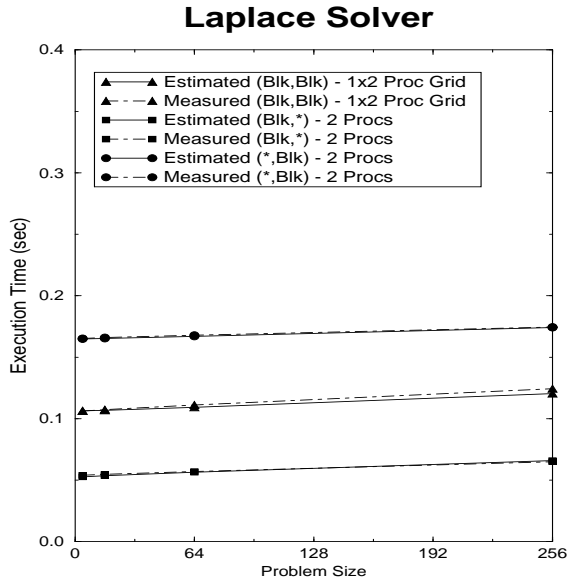
Laplace Solver

Figure 10: Laplace Solver (2 Procs) - Estimated/Measured Times
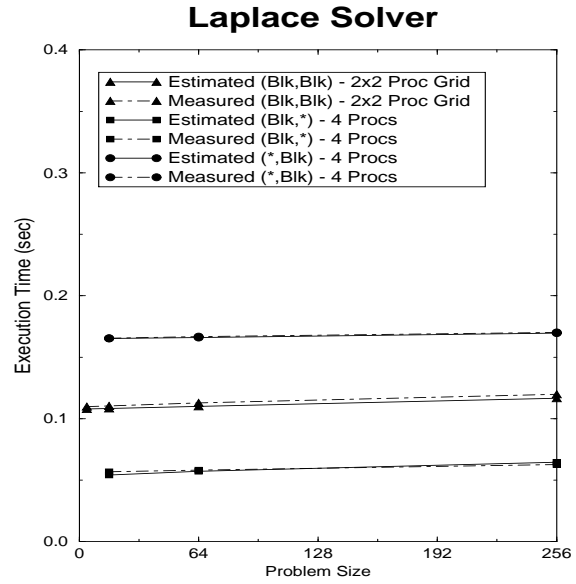
Laplace Solver

Figure 11: Laplace Solver (4 Procs) - Estimated/Measured Times
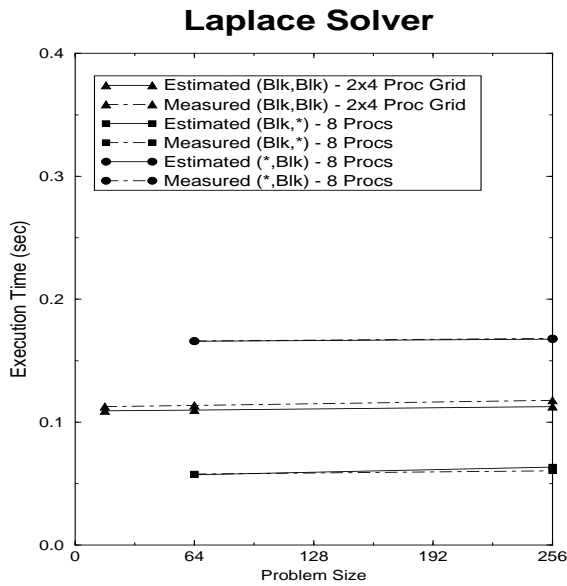
Laplace Solver

Figure 12: Laplace Solver (8 Procs) - Estimated/Measured Times
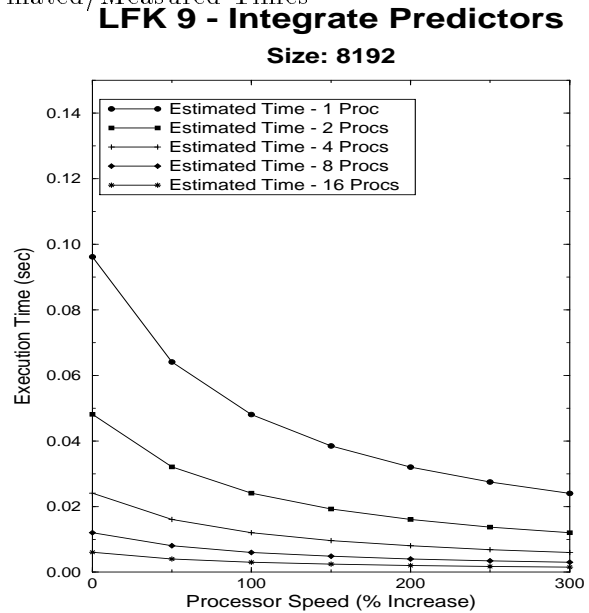
LFK 9 - Integrate Predictors

Size: 8192

Figure 13: Effect of Increasing Processor Speed on Performance

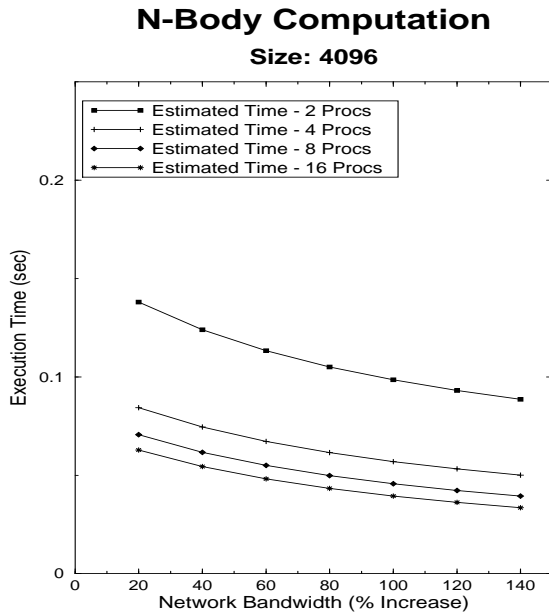it has be run on a slower processor. It can also be used to evaluate the benefits of upgrading to a faster processor system.

Figure 14: Effect of Increasing Network Bandwidth on Performance
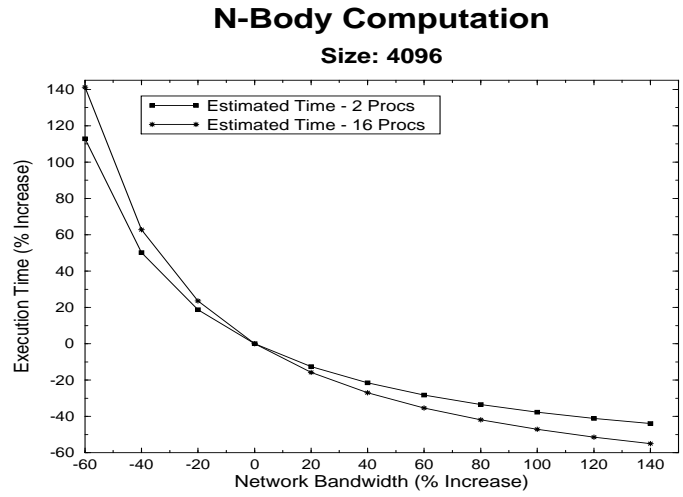
Figure 15: Effect of Varying Network Bandwidth on Performance (% Change in Execution time)

Effect of Varying Interconnection Bandwidth: The effect of varying the interconnect bandwidth on the application performance is shown in Figure 14. The increase/decrease in application execution times is greater for larger processor configurations as illustrated in Figure 15 (negative percentages indicate a decrease in execution time or network bandwidth).

Effect of Varying Network Load: Figure 16 shows the interpreted effects of network load on application performance. It can be seen that the performance deteriorates rapidly as the network gets saturated. Further, the effect of network load is more pronounced for larger system configurations as illustrated in Figure 17.

Experimentation with Larger System Configurations: In this experiment we experiment with larger system configurations than physically available (i.e. 16 & 32 processors). The results are shown in Figures 19 & 18. It can be seen that the first application (Approximation of Π) scales well with increased number of processors; while in the second application (Parallel Stock Option Pricing), larger configurations are beneficial only for larger problem sizes.

The ability to experiment with system parameters not only allows the user to evaluate the application characteristics, but also enables the evaluation of new and different system configurations. This exposes the potential of the framework as a design evaluation tool for system architects. Experimentation with run-time parameter enables the developer to test the robustness of the design and to modify it to account for different run-time scenarios.
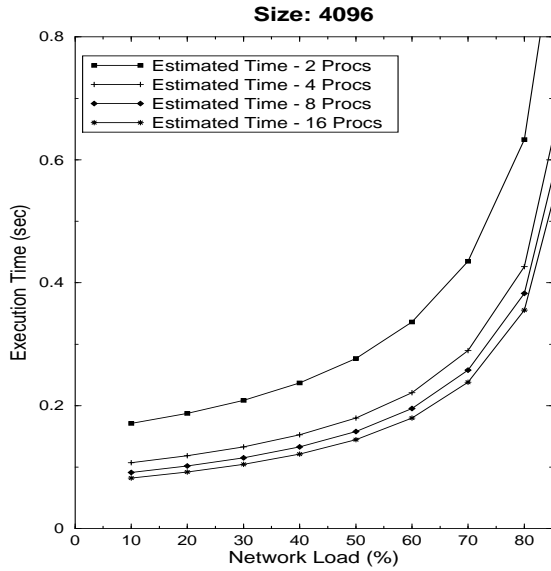
N-Body Computation

Size: 4096

Figure 16: Effect of Increasing Network Load on Performance
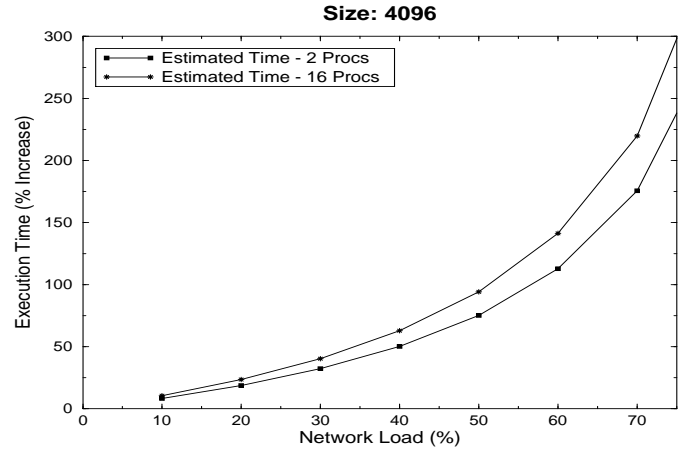
N-Body Computation

Size: 4096

Figure 17: Effect of Varying Network Load on Performance (% Change in Execution time)
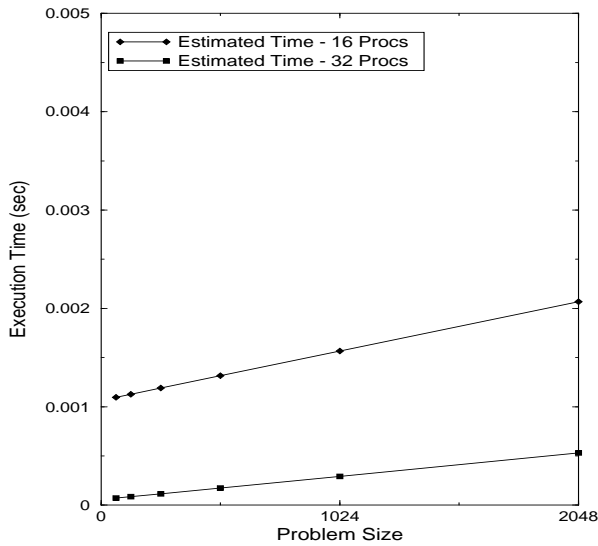
Approximation of PI

Figure 18: Experimentation with Larger System Configurations - Approximation of PI
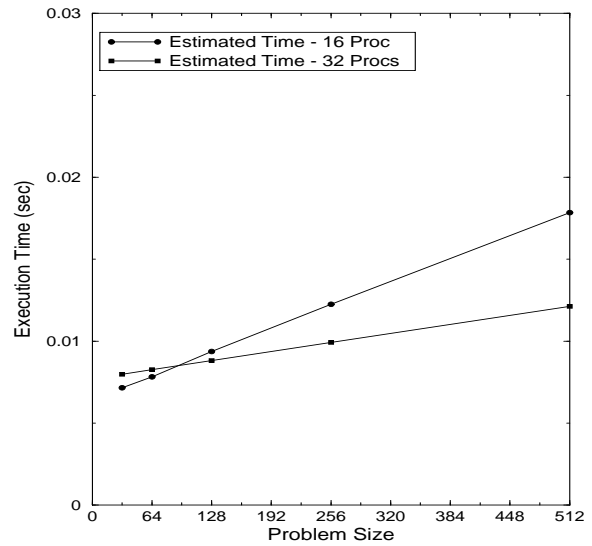
Parallel Stock Option Pricing

Figure 19: Experimentation with Larger System Configurations - Financial Model

5.2.3   Application Performance Debugging

The metrics generated by the interpretive framework can be used to analyze the performance contribution of different parts of the application description and to view their computation time/communication time breakup. This is illustrated below using two applications.

Phase 1

REPEAT I=1:N/2
    CSHIFT 1
      COMPUTE
END

Phase 2

REPEAT I=1:N/2
    CSHIFT 1
END
  ACCUMULATE
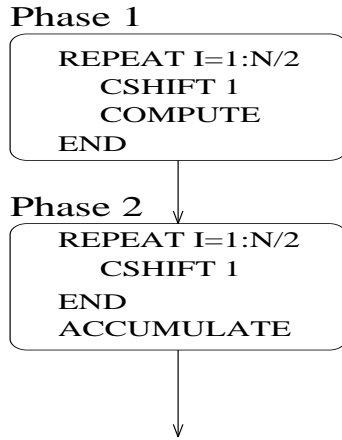
Figure 20: N-Body - Application Phases
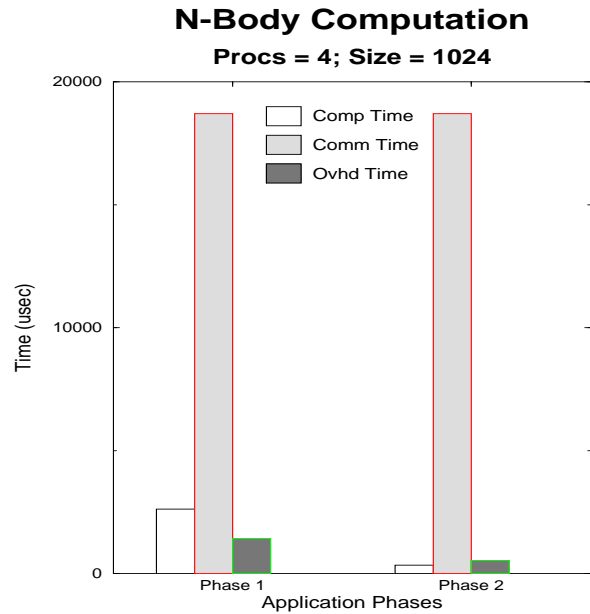
Figure 21: NBody Computation - Interpreted Performance Profile

N-Body Computations: Figure 21 shows the performance profile for two phases of the n-body application. Phase 1 (see Figure 20) represents the forward movement of data around the virtual processor ring while Phase 2 represents accumulation of force data at the original processors. For n processors, each phase requires $n/2$ circular shifts of the data; consequently their communication profiles are similar. However, Phase 1 performs more computation as it computes the force interactions. Overhead time represents parallelization overheads. Similar profiles can be obtained at smaller granularities (up to a single line of code).

Parallel Stock Option Pricing: A performance profile for the parallel stock option pricing application is shown in Figure 23. This application has two phases as shown in Figures 22. Phase 1 creates the (distributed) option price lattice while Phase 2, which requires no communication, computes the call prices of stock options.

Application performance debugging using conventional means involves instrumentation, execution and data collection, and post-processing this data. Further, this process requires a running application and has to be repeated to evaluate each design. Using the interpretive framework, this information is available, at all levels required, during application development.

5.3   Validating Usability of the Interpretive Framework

The interpreted performance estimates for the experiments described above were obtained using the interpretive framework running on a Sparcstation 1+. The framework provides a friendly menu-driven,
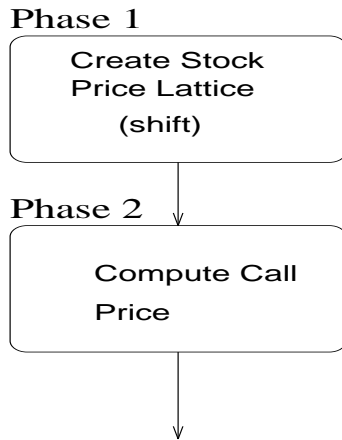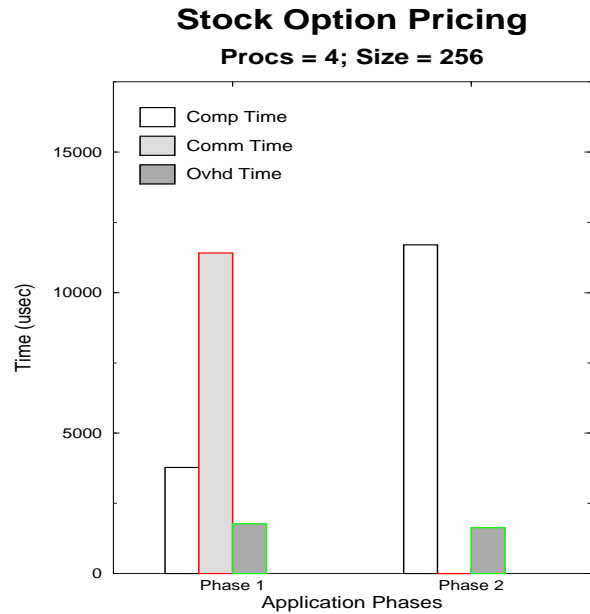
Figure 22: Financial Model - Application Phases

Figure 23: Financial Model - Interpreted Performance Profile

graphical user interface to work with and requires no special hardware other than a conventional workstation and a windowing environment. Application characterization is performed automatically (unlike most approaches) while system abstraction is performed off-line and only once. Application parameters and directives were varied from within the interface itself. Typical experimentation on the iPSC/860 (to obtain measured execution times) consisted of editing code, compiling and linking using a cross compiler (compiling on the front end (or SRM) is not allowed to reduce its load), transferring the executable to the iPSC/860 front end, loading it onto the i860 node and then finally running it. The process had to be repeated for each instance of each experiment. Relative experimentation times for different implementation of the Laplace Solver (Section 5.2.1) using measurements and the performance interpreter are shown in Figure 24. Experimentation using the interpretive approach required approximately 10 minutes for each of the three implementation. Experimentation using measurements however, took a minimum 27 minutes (for the (Blk,*) implementation) and required almost 1 hour for the (*,Blk) case. Clearly, the measurements approach is not feasible, specially when a large number of options have to be evaluated. Further, the iPSC/860, being an expensive resource, is shared by various development groups in the organization. Consequently, its usage can be restrictive and the required configuration may not be immediately available. The comparison above validates the convenience and cost-effectiveness of the framework for experimentation during application development.
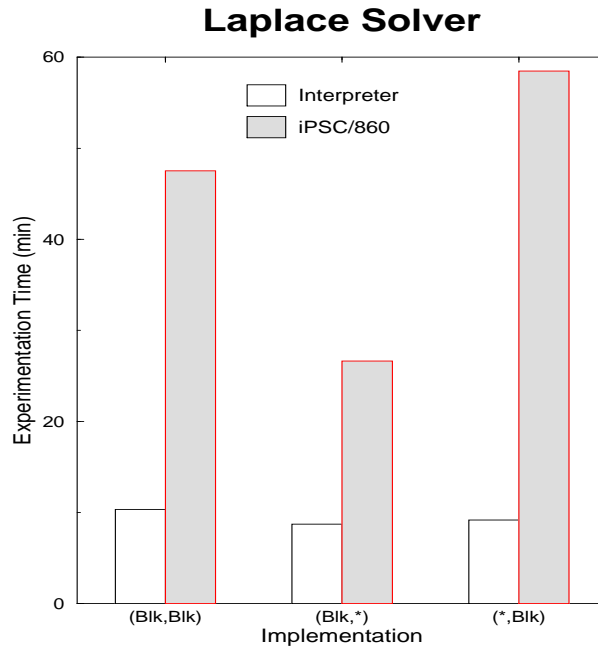
Laplace Solver

Figure 24: Experimentation Time - Laplace Solver

6 Related Work

Existing approaches and models for performance prediction on multicomputer systems can be broadly classified as analytic, simulation, monitoring or hybrid (which make use of a combination of the above techniques along with possible heuristics and approximations).

A general approach for analytic performance prediction for shared memory systems has been proposed by Siewiorek et al. in [4] while probabilistic models for parallel programs based on queueing theory have been presented in [5]. An analytic performance prediction technique based on the approximation of parallel flow graphs by sequential flow graphs has been proposed by Qin et al. in [6]. The above approaches require users to explicitly model the application along with the entire system. A source based analytic performance prediction model for Dataparallel C has been developed by Clement et al. [7]. The approach uses a set of assumptions and specific characteristics of the language to develop a speedup equation for applications in terms of system costs.

A simulation based approach is used in the SiGLe system (Simulator at Global Level) [8] which provides special description languages to describe the architecture, application and the mapping of the application onto the architecture.

An evaluation approach based on instrumentation, data collection and post-processing has been proposed by Darema et al. [9]. Balasundaram et al. [10] use 'training routines" to benchmark the performance of the architecture and then use this information to evaluate different data decompositions.

The PPPT system [11] uses monitoring techniques to profile the execution of the application program on

a single processor, and to derive sequential program parameters such as conditional branch probabilities, loop iteration counts, and frequency counts for each statement type. The user is required to provide a characteristic set of input data for this profiling run. Obtained information is then used by the static parameter based performance prediction tool to estimate performance information for the parallelized (SPMD) application program on a distributed memory system.

A hybrid approach is presented in [12] where the runtime of each node of a stochastic graph representing the application is modeled as a random variable. The distributions of these random variables are then obtained using hardware monitoring.

The layered approach presented in [13] uses a methodology based on application and system characterization. The developer is required to characterize the application as an execution graph and define its resource requirements in this system.

7   Conclusions and Future Work

Evaluation tools form a critical part of any software development environment as they enable the developer to evaluate different design choices available at various stages of application development, and make the most appropriate selection. These tools, in symbiosis with other development tools, complete the feedback loop of the "develop-evaluate-tune" cycle.

In this paper, we described a novel interpretive approach for accurate and cost-effective performance prediction on high performance computing systems. A comprehensive characterization methodology is used to abstract the system and application components of the HPC environment into a set of well defined parameters. An interpreter engine then interprets the performance of the abstracted application in terms of the parameters exported by the abstracted system. A source-driven HPF/Fortran 90D performance prediction framework based on the interpretive approach has been implemented as part of the HPF/Fortran 90D integrated application development environment. The current implementation of the environment framework is targeted to the iPSC/860 hypercube multicomputer system.

Numerical results using benchmarking kernels and application from the NPAC HPF/Fortran 90D Benchmark Suite were presented to validate the accuracy, utility, and usability of the performance prediction framework. The use of the framework for selecting appropriate compiler directives, for application performance debugging and for experimentation with run-time and system parameters was demonstrated.

We are currently working on developing an intelligent HPF/Fortran 90D compiler based on the source based interpretation model. This tool will enable the compiler to automatically evaluate directives and transformation choices and optimize the application at compile time. Future development of the framework will involve moving it to high performance distributed computing systems and exploiting its potential as a system design evaluation tool.

References

[1] Manish Parashar, *Interpretive Performance Prediction for High Performance Parallel Computing*, PhD thesis, Syracuse University, 121 Link Hall, Syracuse, NY 13244-1240, July 1994, Available via WWW at `http://godel.ph.utexas.edu/Members/parashar/ESP/esp.html`.

[2] High Performance Fortran Forum, *High Performance Fortran Language Specifications, Version 1.0*, Jan. 1993, Also available as Technical Report CRPC-TR92225 from Center for Research on Parallel Computing, Rice University, Houston, TX 77251-1892.

[3] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka, "Compiling HPF for Distributed Memory MIMD Computers", in David Lilja and Peter Bird, editors, *Impact of Compilation Technology on Computer Architecture*. Kluwer Academic Publishers, 1993.

[4] Dalibor F. Vrsalovic, Daniel P. Siewiorek, Zary Z. Segall, and Edward F. Gehringer, "Performance Prediction and Calibration for a Class of Multiprocessors", *IEEE Transactions on Computers*, $37(11):1353-1365$, Nov. 1988.

[5] Philip Heildelberger and Kishore S. Trivedi, "Analytic Queueing Models for Programs with Internal Concurrency", *IEEE Transactions on Computers*, $\mathbf{C\text{-}32}(1):73-82$, Jan. 1983.

[6] Reda A. Ammar and Bin Qin, "A Technique to Derive the Detailed Time Costs of Parallel Computations", *Proceedings of the 12th Annual International Computer Software and Application Conference*, pp. 113–119, 1988.

[7] Mark J. Clement and Micheal J. Quinn, "Analytic Performance Prediction on Multicomputers", Technical report, Department of Computer Science, Oregon State University, Mar. 1993.

[8] F. Andre and A. Joubert, "SiGLe: An Evaluation Tool for Distributed Systems", *Proceedings of the International Conference on Distributed Computing Systems*, pp. 466–472, 1987.

[9] Frederica Darema, "Parallel Applications Performance Methodology", in Margaret Simmons, Rebecca Koskela, and Ingrid Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, chapter 3, pp. 49–57. Addison-Wesley Publishing Company, 1988.

[10] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer, "A Static Performance Estimator in the Fortran D Programming System", in Joel Saltz and Piyush Mehrotra, editors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, pp. 119–138. Elsevier Science Publishers B.V., 1992.

[11] Thomas Fahringer and Hans P. Zima, "A Static Parameter based Performance Prediction Tool for Parallel Programs", *Proceedings of the 7th ACM International Conference on Supercomputing, Japan*, July 1993.

[12] Franz Sötz, "A Method for Performance Prediction of Parallel Programs", in H. Burkhart, editor, *Joint International Conference on Vector and Parallel Processing, Proceedings, Zurich, Switzerland*, pp. 98–107. Springer, Berlin, LNCS 457, Sep. 1990.

[13] E. Papaefstathiou, D. J. Kerbyson, and G. R. Nudd, "A Layered Approach to Parallel Software Performance Prediction: A Case Study", *Massively Parallel Processing Applications and Development, Delft*, 1994.

A   Accuracy of the Interpretation Framework

Estimated and measured execution times corresponding to the results summarized in Table 4 are plotted in Figures 25-37 below:
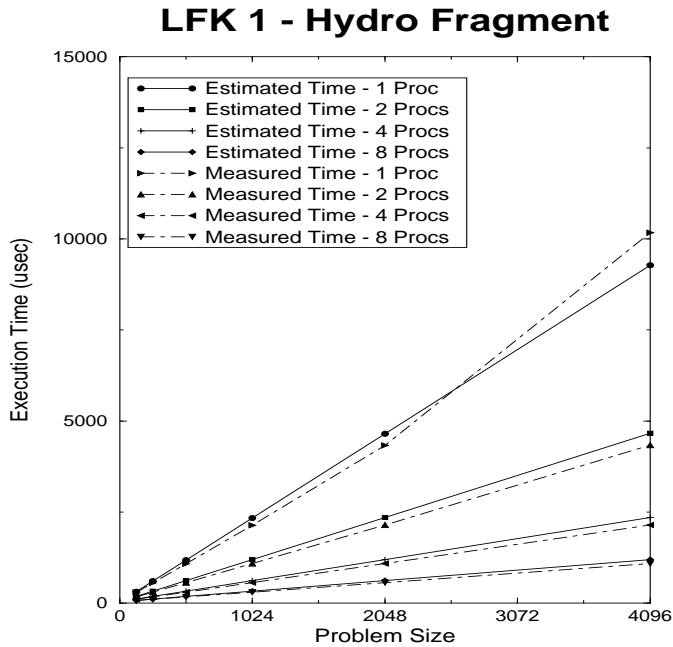
LFK 1 - Hydro Fragment

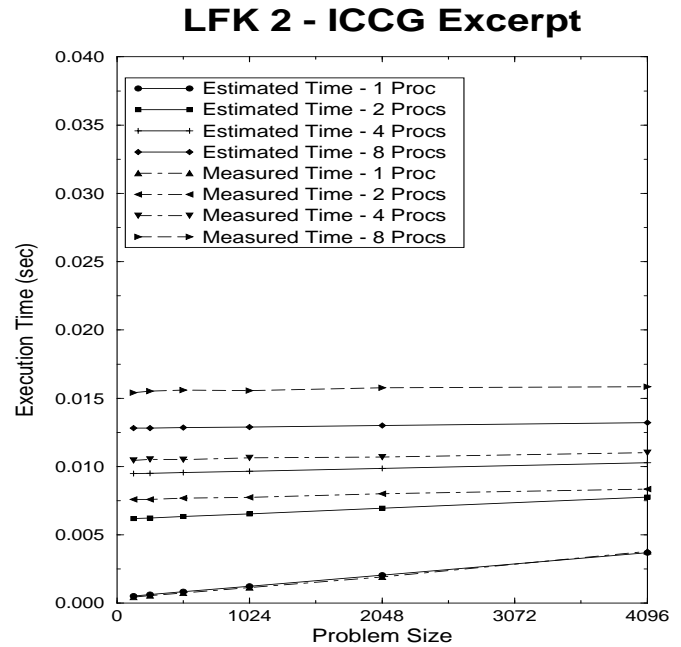Figure 25: LFK 1 - Estimated/Measured Times

LFK 2 - ICCG Excerpt

Figure 26: LFK 2 - Estimated/Measured Times

LFK 3 - Inner Product
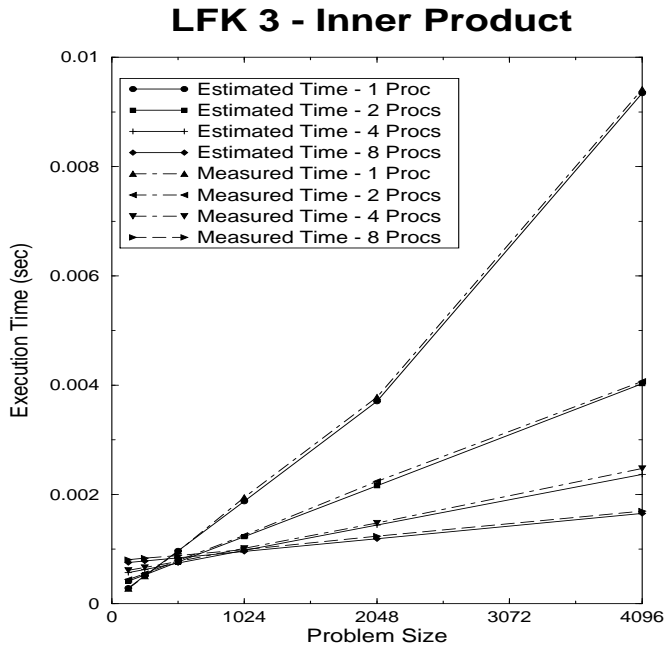
Figure 27: LFK 3 - Estimated/Measured Times

LFK 9 - Integrate Predictors
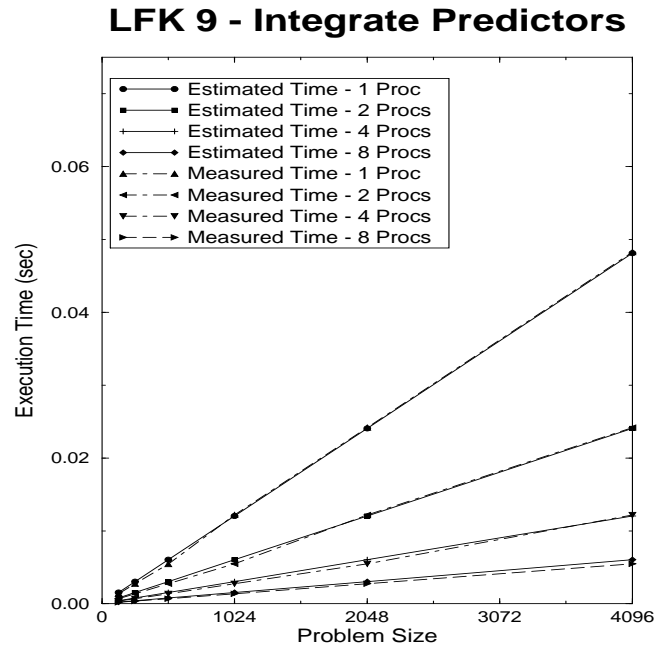
Figure 28: LFK 9 - Estimated/Measured Times

LFK 14 - 1-D PIC
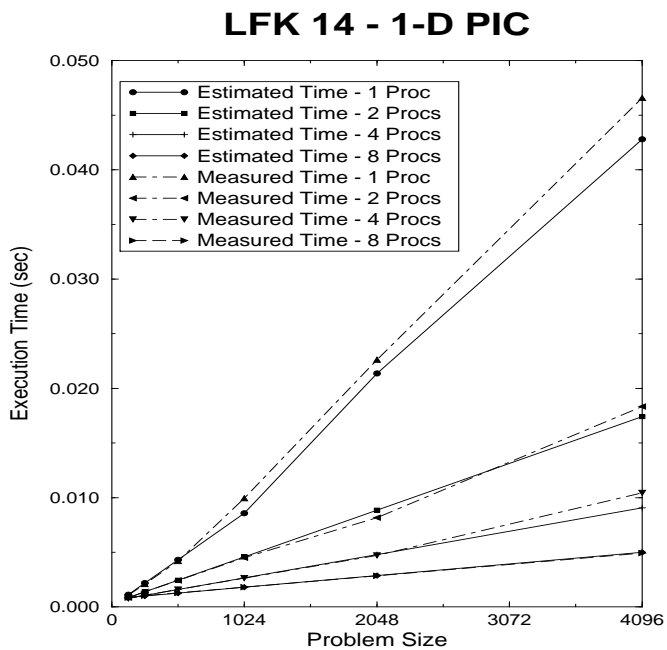
Figure 29: LFK 14 - Estimated/Measured Times

LFK 22 - Planckian Distribution
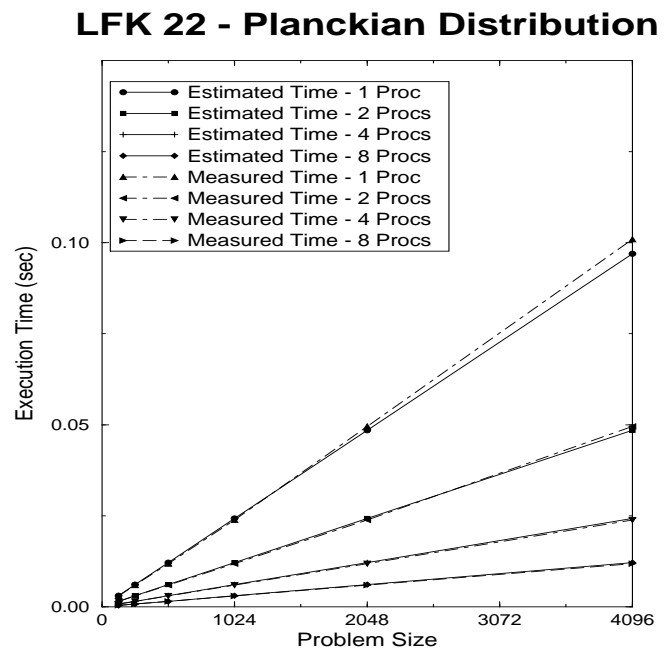
Figure 30: LFK 22 - Estimated/Measured Times

PBS 1 - Trapezoidal Rule
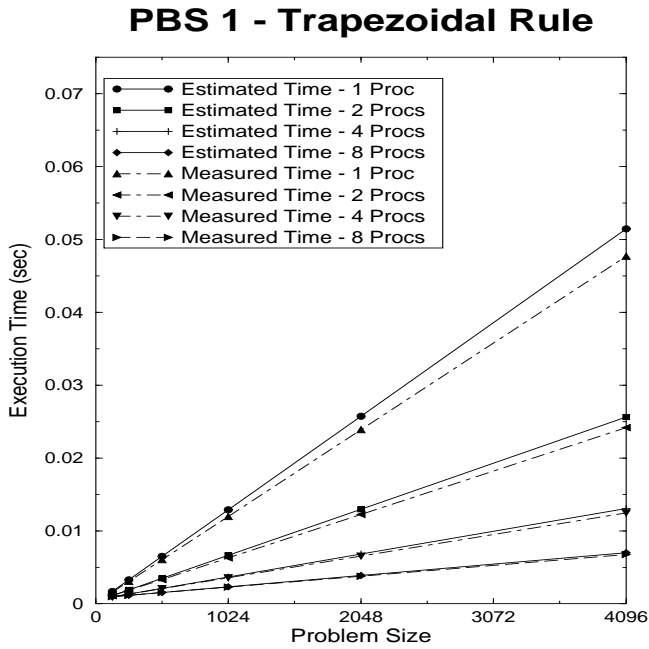
Figure 31: PBS 1 - Estimated/Measured Times

PBS 2 - Computation of e*
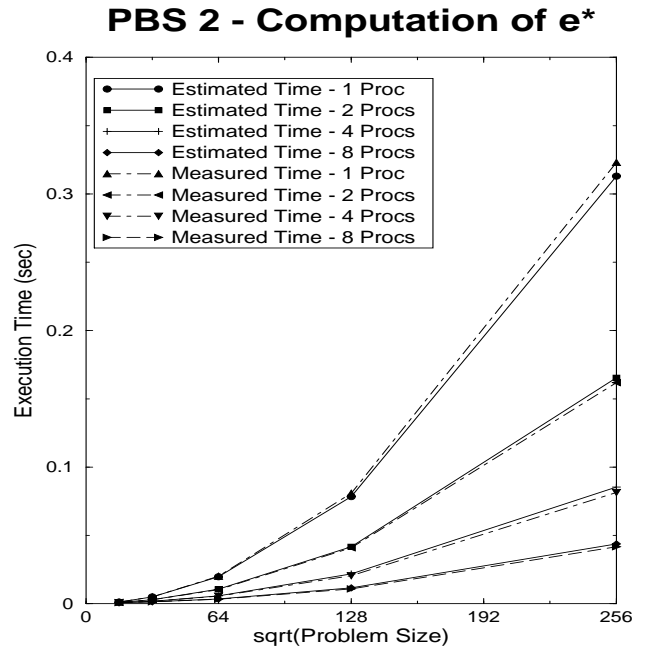
Figure 32: PBS 2 - Estimated/Measured Times

PBS 3 - Computation of S
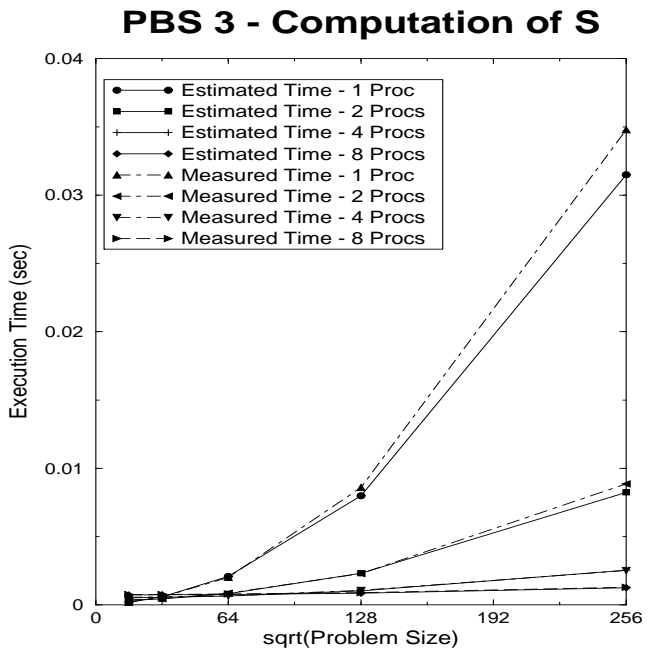
Figure 33: PBS 3 - Estimated/Measured Times

PBS 4 - Computation of R
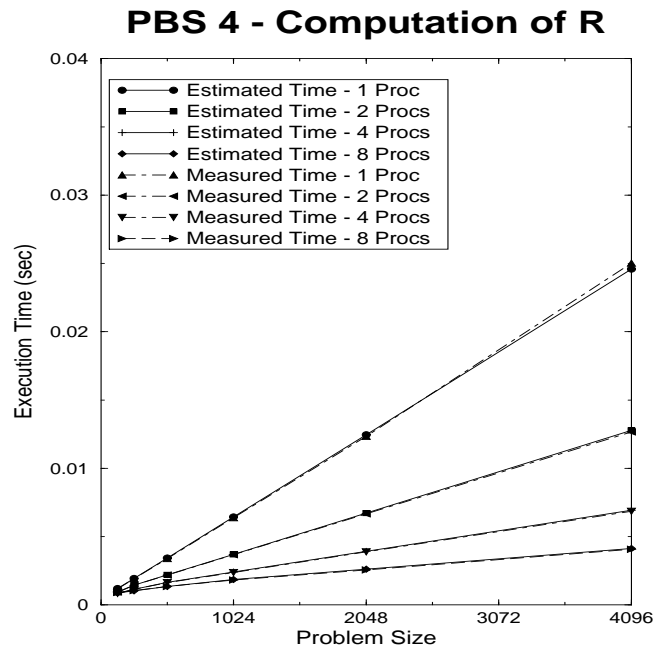
Figure 34: PBS 4 - Estimated/Measured Times

Approximation of PI
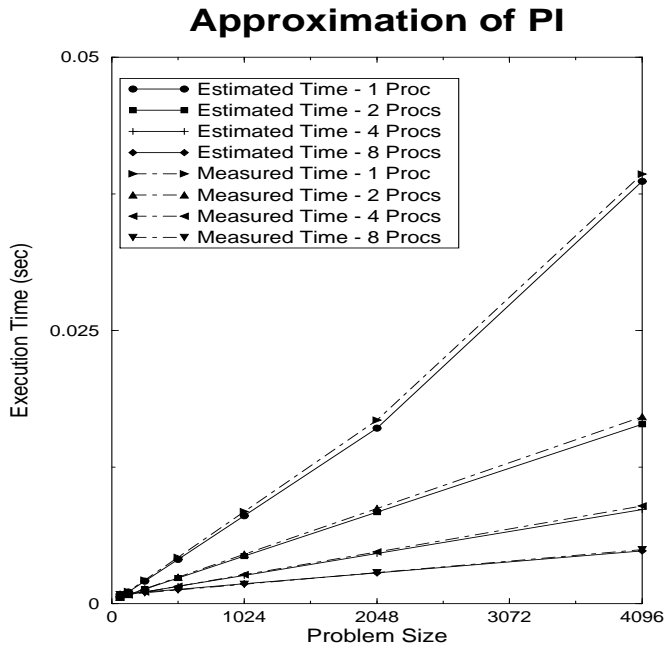
Figure 35: PI - Estimated/Measured Times

N-Body Computation
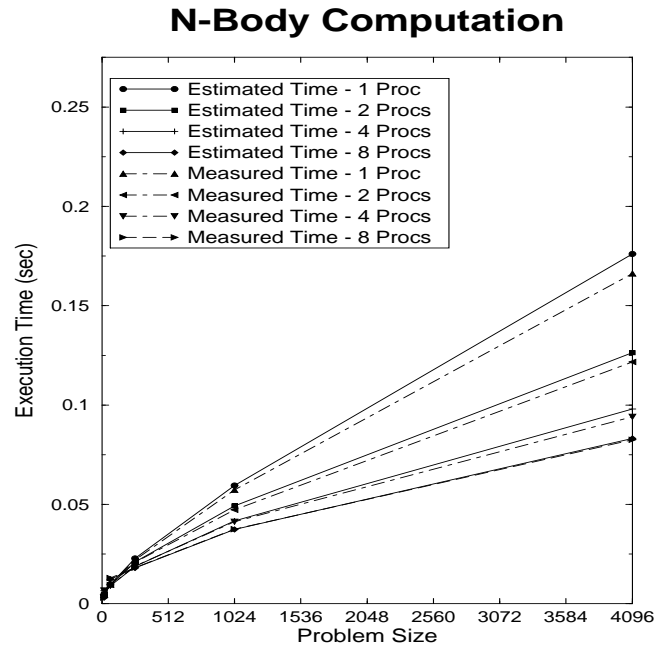
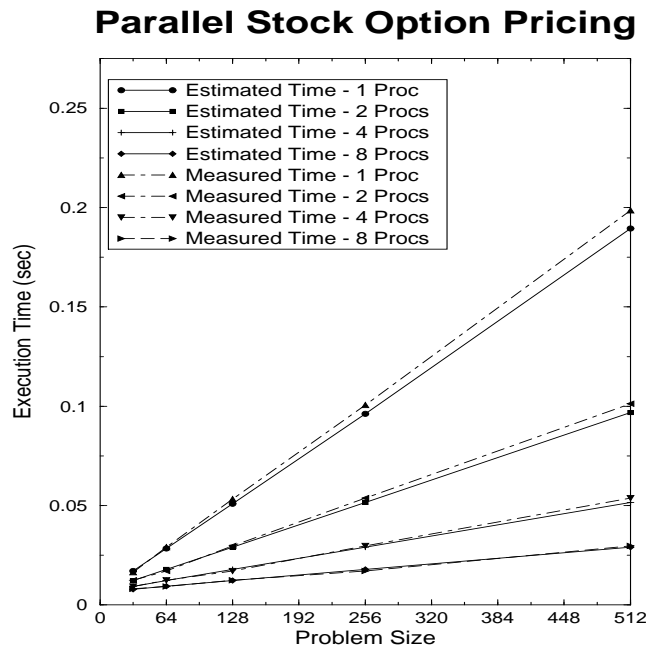Figure 36: N-Body - Estimated/Measured Times

Parallel Stock Option Pricing

Figure 37: Financial Model - Estimated/Measured Times