Interpretive Performance Prediction for Parallel Application Development

Manish Parashar
Dept. of Electrical & Computer Engineering
Rutgers, The State University of New Jersey
Piscataway, NJ 08854-8058
parashar@caip.rutgers.edu

Salim Hariri
Dept. of Electrical & Computer Engineering
University of Arizona
Tucson, AZ 85721-0104
hariri@ece.arizona.edu

Abstract

Application software development for High-Performance Parallel Computing (HPC) is a non-trivial process; its complexity can be primarily attributed to the increased degrees of freedom that have to be resolved and tuned in such an environment. Performance prediction tools enable a developer to evaluate available design alternatives and can assist in HPC application software development.

In this paper we first present a novel "interpretive" approach for accurate and cost-effective performance prediction. The approach has been used to develop an interpretive HPF/Fortran 90D application performance prediction framework. The accuracy and usability of the performance prediction framework are experimentally validated. We then outline the stages typically encountered during application software development for HPC and highlight the significance and requirements of a performance prediction tool at relevant stages. Numerical results using benchmarking kernels and application codes are presented to demonstrate the application of the interpretive performance prediction framework at different stages of the HPC application software development process.

Keywords: *HPC application software development, Interpretive performance prediction, HPF/Fortran 90D application development.*

1 Introduction

Development of efficient application software for High-Performance Parallel Computing (HPC) is a non-trivial process and requires a thorough understanding not only of the application but also of the target computing environment. A key factor contributing to this complexity is the increased degrees of freedom that have to be resolved and tuned in such an environment. Typically, during the course of parallel application software development, the developer is required to select between available algorithms, 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 quiet large and selecting the most appropriate one can be a formidable task.

Evaluation tools enable a developer to visualize the effects of various design alternatives. Conventional evaluation techniques typically require extensive experimentation and data collection. Most existing evaluation tools post-process traces generated during an execution run. This implies instrumenting source code, executing the application on the actual hardware to generate traces, post-processing these traces to gain insight into the execution and overheads in the implementation, refining the implementation and then repeating the process. The process is repeated until all possibilities have been evaluated and the most suitable options for the problem have been identified. Such a development overhead can be tedious if not impractical.

Performance prediction tools provide a more practical and cost-effective means for evaluating available design alternatives and making design decisions. These tools, in symbiosis with other development tools, can be effectively used to complete the feedback loop of the "develop-evaluate-tune" cycle in the HPC application software development process.

In this paper we first present a novel interpretive approach for accurate and cost-effective performance prediction that can be effectively used during HPC software development. The essence of the approach is the application of *interpretation* techniques to performance prediction through an appropriate characterization of the HPC system and the application. An interpretive HPF/Fortran 90D application performance prediction framework has been implemented using the interpretive approach and is part of the NPAC[1] HPF/Fortran 90D application development environment. The accuracy and usability of the framework are experimentally validated.

Next, we outline the stages typically encountered during HPC application software development and highlight the significance and requirements of a performance prediction tool at relevant stages. Numerical results obtained using application codes and benchmarking kernels are then presented to demonstrate the application of the performance prediction framework to different stages of the application software development process outlined.

The rest of the paper is organized as follows: Section 2 introduces the interpretive approach to performance prediction. Section 3 then describes the HPF/Fortran 90D performance prediction framework and presents numerical results to validate the accuracy and usability of the interpretive approach. Section 4 outlines the HPC software development process and highlights the significance of performance prediction tools. Section 5 presents experiments to illustrate the application of the framework to different stages of the HPC software development process. Section 6 presents some concluding remarks.

[1]Northeast Parallel Architectures Center

2 Interpretive Performance Prediction

Interpretive performance prediction is an accurate and cost-effective approach for compile-time estimation of application performance. The essence of the approach is the application of *interpretation* techniques to performance prediction through an appropriate characterization of the HPC system and the application. A *system abstraction* methodology is defined to hierarchically abstract the HPC system into a set of well defined parameters which represent its performance. A corresponding *application abstraction* methodology is defined to abstract a high-level application description into a set of well defined parameters which represent 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 1 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 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 more detailed discussion of the performance interpretation approach can be found in [1].

Figure 1: Interpretive Performance Prediction

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 2 illustrates the system abstraction process using 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). System characterization thus proceeds recursively down the system hierarchy, generating SAU's of finer granularity at each level. The process terminates when the required granularity of parameterization is achieved. This choice is usually driven by a tradeoff between accuracy and cost-effectiveness.

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 to abstract the control structure of the application forming an *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 a *Synchronized Application Abstraction Graph (SAAG)*. The machine specific filter then incorporates machine specific information (such as introduced compiler transformations/optimizations which are specific to the particular machine) into the SAAG based on the mapping that is being evaluated. Figure 3 illustrates the application abstraction process using a sample application description.

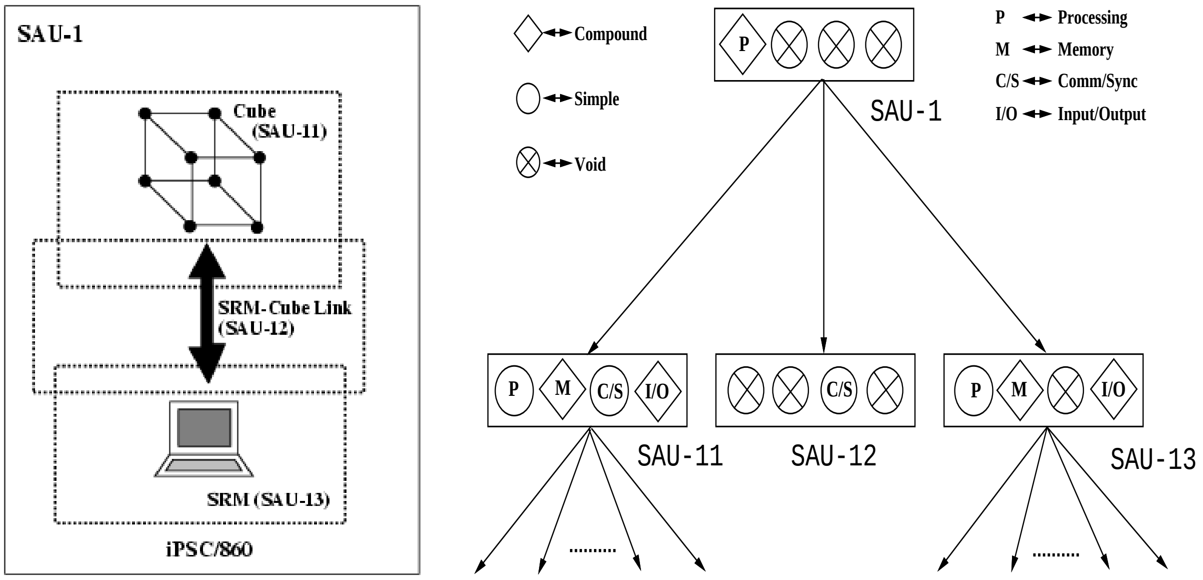
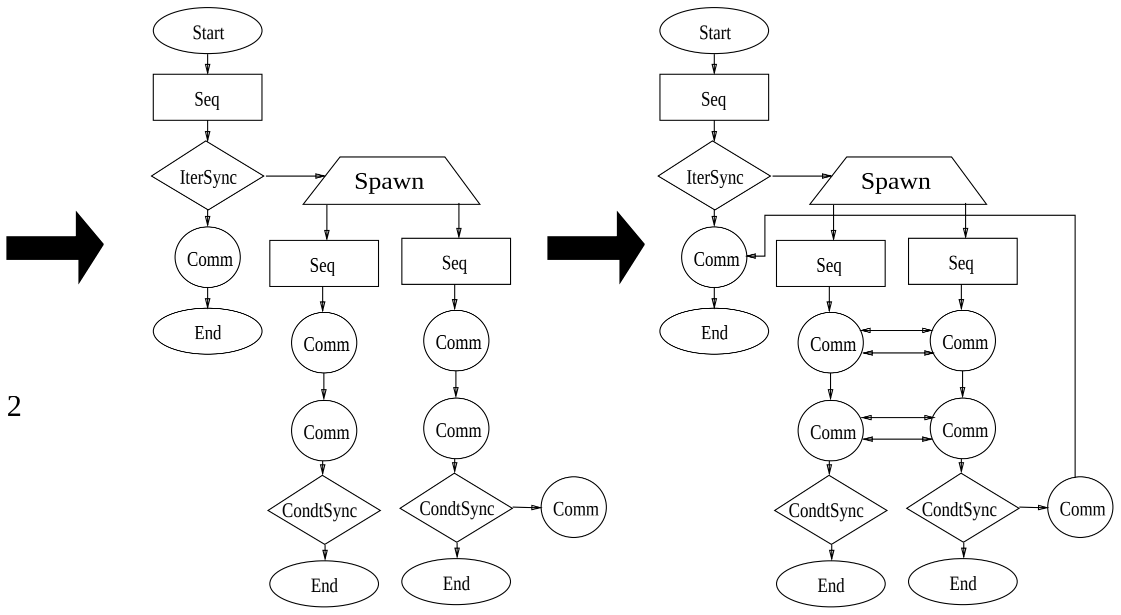


Figure 2: System Abstraction Process

```
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 3: Application Abstraction Process

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 a SAAG to predict the performance of the corresponding application. The interpretation function 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 briefly described below. A more detailed discussion of these models and the complete set of interpretation functions can be found in [1].

Modeling Communication/Synchronization: Communication or synchronization operations in the application are decomposed during interpretation into three components (as shown in Figure 4):

- **Call Overhead:** This represents fixed overheads associated with the operation.
- **Transmission Time:** This is the time required to actually transmit the message from source to 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.

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

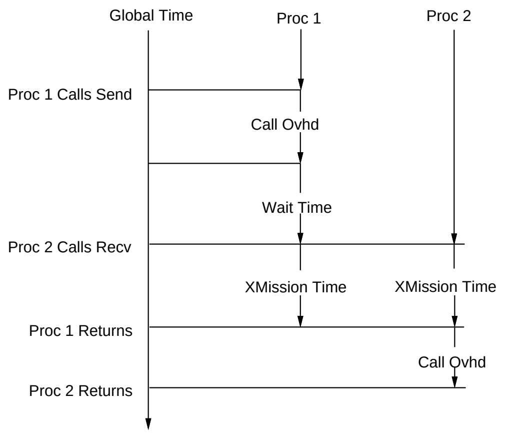


Figure 4: Interpretation Model for Communication/Synchronization AAU's

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 as 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 recursively 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 i^{th} 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 3.5.

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 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.

2.5 Related Research in Performance Prediction

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 [2] while probabilistic models for parallel programs based on queueing theory have been presented in [3]. 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 [4]. 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. [5]. 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) [6] 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. [7]. Balasundaram et al. [8] use "training routines" to benchmark the performance of the architecture and then use this information to evaluate different data decompositions.

The PPPT system [9] 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 [10] 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 [11] 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.

3 A HPF/Fortran 90D Performance Prediction Framework

3.1 An Overview of HPF/Fortran 90D

High Performance Fortran (HPF) [12] 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.

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 5.

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.

3.2 ESP: The HPF/Fortran 90D Performance Prediction Framework

ESP (see Figure 6) is an interpretive framework for HPF/Fortran 90D application performance prediction. It uses the interpretive approach outlined above to provide accurate and cost-effective performance prediction of HPF/Fortran 90D. ESP has been implemented as a part of the HPF/Fortran 90D application development environment [13] developed at the NPAC, Syracuse University.

The design of ESP is is based on the HPF source-to-source compiler technology [14] 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:

```
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)
```

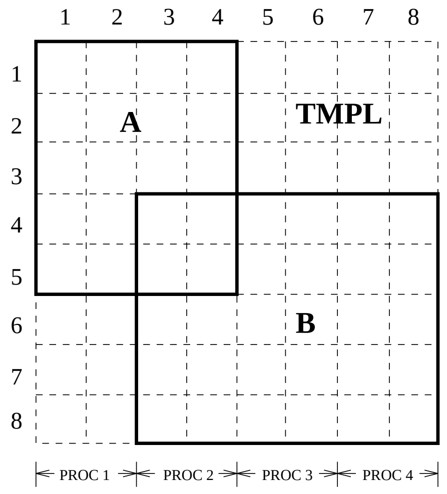
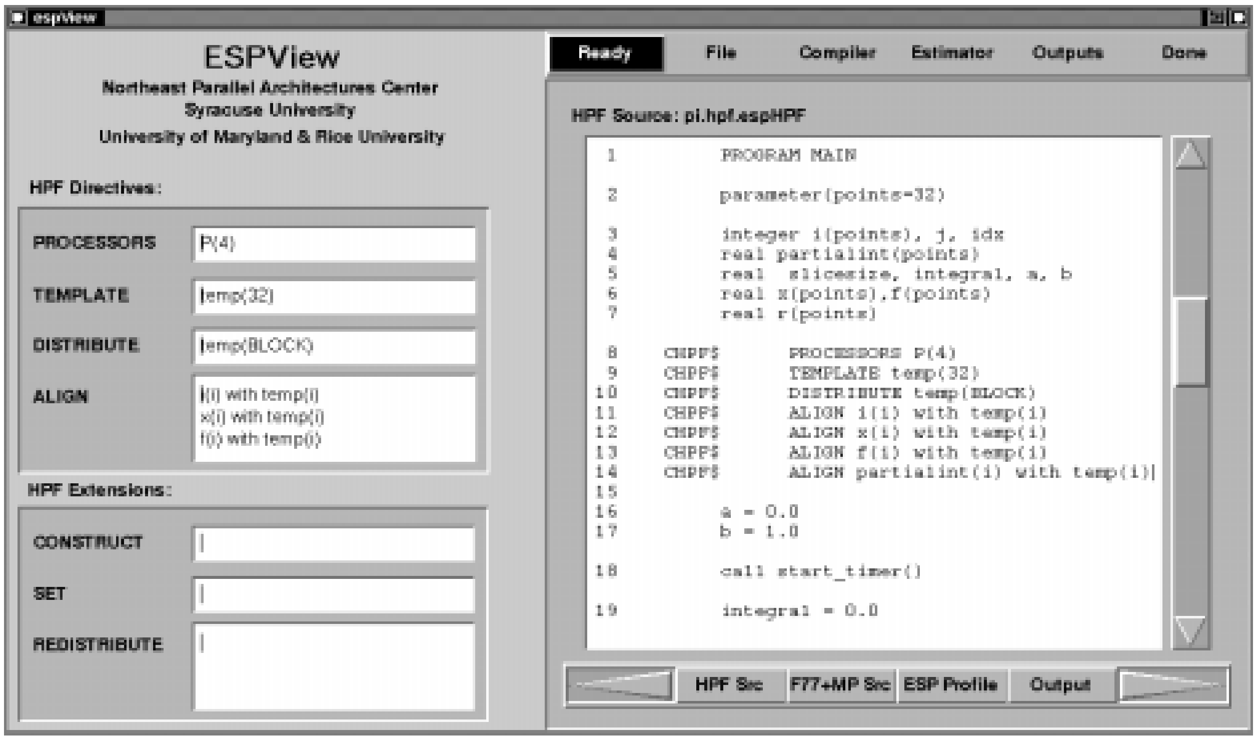


Figure 5: HPF/Fortran 90D Directives



HPF Directives:

PROCESSORS	P(4)
TEMPLATE	temp(32)
DISTRIBUTE	temp(BLOCK)

ALIGN	i(i) with temp(i)
	x(i) with temp(i)
	f(i) with temp(i)

HPF Extensions:

CONSTRUCT	
SET	
REDISTRIBUTE	

HPF Source: pi.hpf.espHPF

```
          PROGRAM MAIN

          parameter(points=32)

          integer i(points), j, idx
          real partialint(points)
          real  slicesize, integral, a, b
          real x(points),f(points)
          real r(points)

      CHPF$       PROCESSORS P(4)
      CHPF$       TEMPLATE temp(32)
      CHPF$       DISTRIBUTE temp(BLOCK)
      CHPF$       ALIGN i(i) with temp(i)
      CHPF$       ALIGN x(i) with temp(i)
      CHPF$       ALIGN f(i) with temp(i)
      CHPF$       ALIGN partialint(i) with temp(i)

          a = 0.0
          b = 1.0

          call start_timer()

          integral = 0.0
```

Figure 6: The HPF/Fortran 90D Application Development Environment

3.3 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.

3.4 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 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 metrics 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 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 such as ParaGraph [2]. The user can then use the capabilities provided by the package to analyze the performance of the application.

3.5 Experimental Evaluation of ESP

The experimental evaluation presented in section has the following objectives:

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

2. 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 8 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 HPF/Fortran 90D Benchmark Suite [15]. 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 partitioning 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 2.

[2]Developed at Oak Ridge National Laboratory, http://www.ornl.gov

Name	Description
	Livermore Fortran Kernels (LFK)
LFK 1	Hydro Fragment
LFK 2	ICCG Excerpt (Incomplete Cholesky; Conj. Grad.)
LFK 3	Inner Product
LFK 9	Integrate Predictors
LFK 14	1-D PIC (Particle In Cell)
LFK 22	Planckian Distribution
	Purdue Benchmarking Set (PBS)
PBS 1	Trapezoidal rule estimate of an integral of f(x)
PBS 2	Compute $e^* = \sum_{i=1}^{n} \prod_{j=1}^{m} \left(1 + \frac{0.5}{-\|i-j\|+0.001}\right)$
PBS 3	Compute $S = \sum_{i=1}^{n} \prod_{j=1}^{m} a_{ij}$
PBS 4	Compute $R = \sum_{i=1}^{n} \frac{1}{x_i}$
PI	Approximation of π 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

Table 2: Validation Application Set

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%
Financial	32 - 512	1 - 8	1.1%	4.6%
Laplace (Blk-Blk)	16 - 256	1 - 8	0.2%	4.4%
Laplace (Blk-X)	16 - 256	1 - 8	0.6%	4.9%
Laplace (X-Blk)	16 - 256	1 - 8	0.1%	2.8%

Table 3: Accuracy of the Performance Prediction Framework

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 multiple runs. The results obtained are summarized in Table 3. Error values listed are percentages of the measured time and represent maximum/minimum absolute errors over all problem sizes and 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%.

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. The objectives 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.

3.5.2 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, graphical user interface to work with and requires no special hardware other than a conventional desktop workstation. 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 obtained measured execution times) consisted of editing code, compiling and linking using a cross compiler (compiling on the front end 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 application (for different problem decompositions) using measurements and the performance interpreter are shown in Figure 7. Experimentation using the interpretive approach required approximately 10 minutes for each of the three implementations. Experimentation using measurements however, took a minimum 27 minutes (for the (BLOCK,*) decomposition) and required almost 1 hour for the (*,BLOCK) case. Clearly, the measurements approach can be very tedious and time consuming, 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.

4 The HPC Application Software Development Process

In this section we outline the HPC application software development process as a set of stages (see Figure 8) typically encountered by an application developer. The input to development process is the application specification generated either from the problem statement itself (if it is a new problem) or from existing code (when porting of dusty decks). The final output is a running application. Feedback loops are present at some stages for step-wise refinement and tuning. The stages are briefly listed below. A detailed description of each stage as well as the nature and requirements of support tools that can assist the developer can be found in [16].

4.1 Inputs

The input to the software development process is the application specification in the form of a functional flow description of the application and its requirements. The application specification corresponds to the "user requirement document" in a traditional life-cycle models. Supporting tools at this stage include expert system based tools and intelligent editors, both equipped with a knowledge base to assist in analyzing the application. In Figure 8 these tools are included in the "Application Specification Filter" module.

4.2 Application Analysis Stage

The function of the application analysis stage is to thoroughly analyze the input application specification with the objective of achieving the most efficient implementation. The output of this stage is a detailed process flow graph (the "Parallelization Specification") where the nodes of the graph represent functional modules and the edges represent interdependencies. The key functions performed by this include: (1) functional module creation, i.e. identification of functions that can be executed in parallel; (2) functional module classification, i.e. identification of standard functions; and (3) module synchronization, i.e. analysis of mutual interdependencies. This stage corresponds to the "design phase" in standard software life-cycle models and its output corresponds to the "design document".

4.3 Application Development Stage

The application development stage receives a process flow graph as input and generates an implementation which can then be compiled and executed. The key functions performed by this stage include: (1) algorithm development, i.e. assist the developer in identifying functional components in the input flow graph and selecting appropriate algorithmic implementations; (2) system level mapping, i.e. help the developer in selecting the appropriate HPC system and system configuration for the application; (3) machine level mapping, i.e. help the developer appropriately mapping functional component(s) onto processor(s) in the selected HPC configuration; and (4) implementation & coding, i.e. handle code generation and code filling of selected templates so as to produce a parallel program which can then be compiled and executed on the target system.

A key component of this stage is the design evaluator that assists the developer in evaluating different options available and identifying the option that provides the best performance. The design evaluator estimates the performance of the current design on the target system and provides insight into computation and communication costs, existing idle times and overheads. The estimated performance can then be used to identify regions where further refinement or tuning is required. The key features of the design evaluator are: (1) the ability to provide evaluations with desired accuracy, with minimum resource requirements and within a reasonable amount of time; (2) the ability to automate the evaluation process; and (3) the ability to perform the evaluation without having to run the application on the target system(s).

4.4 Compile-Time & Run-Time Stage

The compile-time/run-time stage handles the task of executing the parallelized application generated by the development stage to produce the required output. The compile-time portion of this stage consists of optimizing compilers and tools for resource allocation and initial scheduling. The responsibility of the run-time portion include handling dynamic scheduling, dynamic load balancing, migrations, and irregular communications.

4.5 Evaluation Stage

In the evaluation stage, the developer retrospectively evaluates the design choices made during the development stage and looks for ways to improve the design. This stage performs a thorough evaluation of the execution of the entire application, detailing communication and computation times, communication and synchronization overheads and existing idle times. That is, it uses application performance debugging to identify regions in the implementation where performance improvement is possible. The evaluation methodology enables the developer to investigate the effect of various run-time parameters like system load and network contention on performance, as well as the scalability of the application with machine and problem size. The key feature of this stage is the ability to perform evaluation with the desired accuracy and granularity, while maintaining tractability and non-intrusiveness.

4.6 Maintenance/Evolution Stage

In addition to the above described stages encountered during the development and execution of HPC applications, there is an additional stage in the life-cycle of this software which involves its maintenance and evolution. The functions of this stage include monitoring the operation of the software and ensuring that it continues to meet its specifications with changes in system configuration.

5 Application of the Interpretive Framework to HPC Software Development

The interpretive performance prediction framework can be effectively used at different stages of the software development process outlined in Section 4. In this section we present experiments performed using the current implementation of the ESP HPF/Fortran 90D performance prediction framework to illustrate its application to HPC software development.

5.1 Application Development Stage

The Design Evaluator module of the Application Development Stage is responsible for evaluating the different implementation and mapping alternatives available to the other modules of this stage. To illustrate the application of the interpretive framework to this stage, we demonstrate how the framework can be used to select an appropriate problem decomposition and mappings for a given system configuration. This is achieved by comparing the performance of the Laplace solver application for 3 different distributions (HPF DISTRIBUTE directive) of the template, namely (BLOCK,BLOCK), (BLOCK,X) and (X,BLOCK), and corresponding alignments (HPF ALIGN directive) of the data elements to the template. These three distributions (on 4 processors) are shown in Figure 9 and the corresponding HPF/Fortran 90D descriptions are listed in Table 4.

Figures 10-13 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 configuration. For the Laplace solver, the (Block,X) 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.

The key requirement of the design evaluator module is that it provides the ability to obtain evaluations with the desired accuracy, with minimum resource requirements and within a reasonable amount of time; the ability to automate the evaluation process; and the ability to perform the evaluation within an integrated workstation environment without running the application on the target computers. In the above experiment, performance interpretation was source driven and can be automated into an intelligent capable of selecting appropriate decompositions and mappings. Further, as demonstrated in Section 3.5.2, performance interpretation is performed on a workstations and requires a fraction of the experimentation time. The interpretive framework thus can be effectively used to provide the functionality of the Design Evaluator Module in the Design Evaluation stage of the HPC software development process.

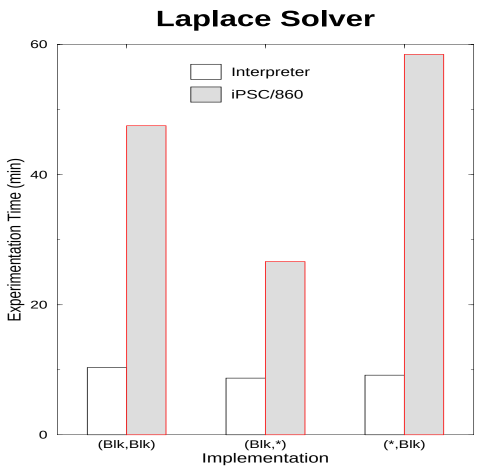


Figure 7: Experimentation Time - Laplace Solver

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

Table 4: Possible Distributions for the Laplace Solver Application

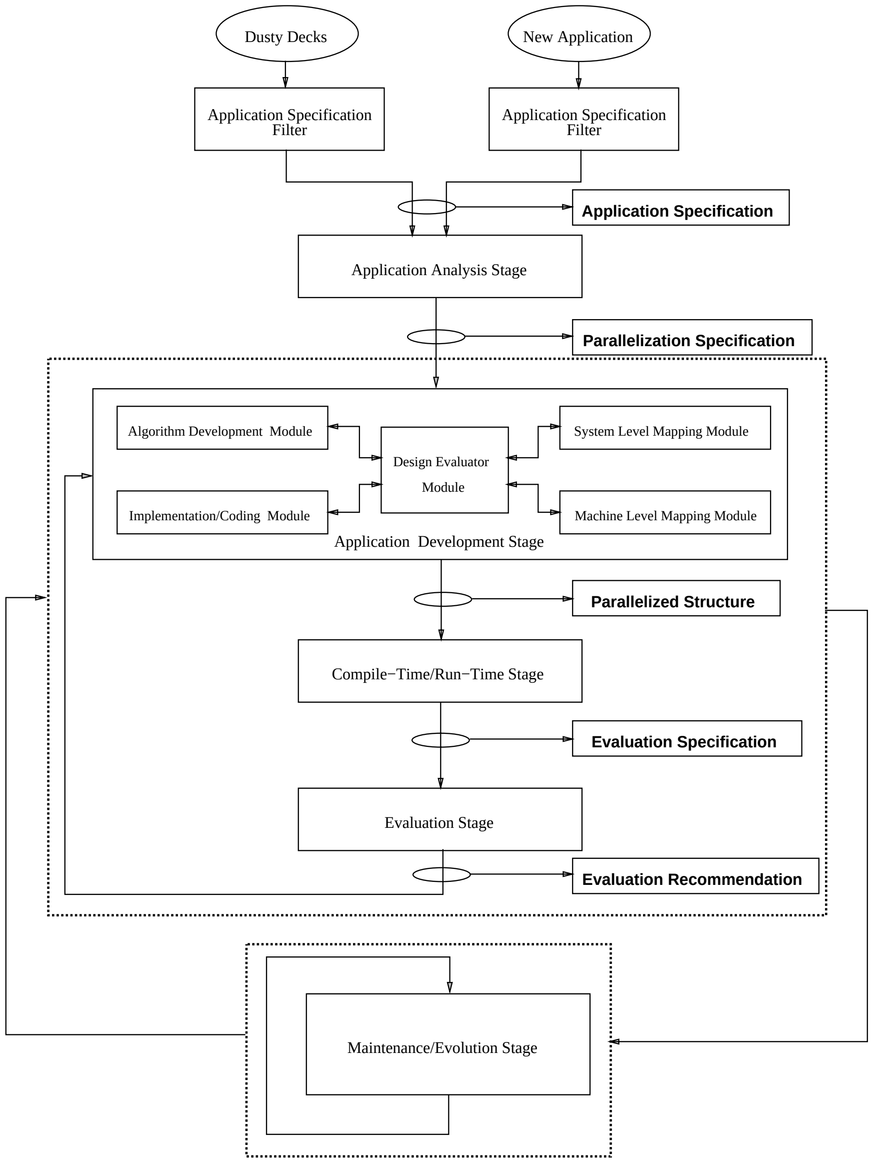

Dusty Decks
New Application
Application Specification Filter
Application Specification Filter
Application Specification
Application Analysis Stage
Parallelization Specification
Algorithm Development Module
Design Evaluator Module
System Level Mapping Module
Implementation/Coding Module
Machine Level Mapping Module
Application Development Stage
Parallelized Structure
Compile−Time/Run−Time Stage
Evaluation Specification
Evaluation Stage
Evaluation Recommendation
Maintenance/Evolution Stage

Figure 8: The HPC Software Development Process

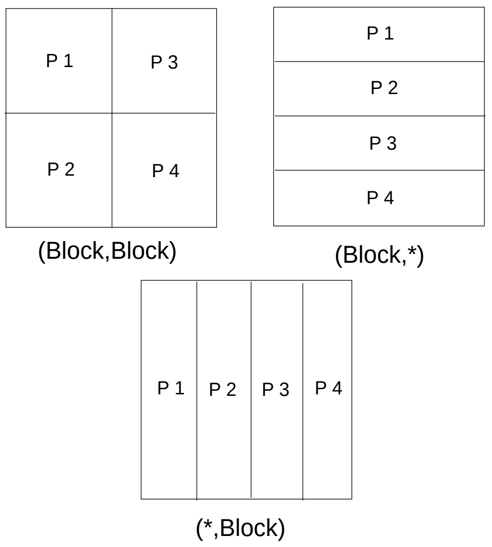


Figure 9: Laplace Solver - Data Distributions

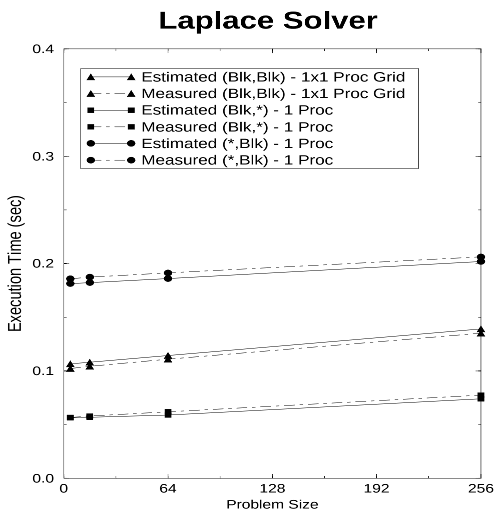


Figure 10: Laplace Solver (1 Proc) - Estimated/Measured Times

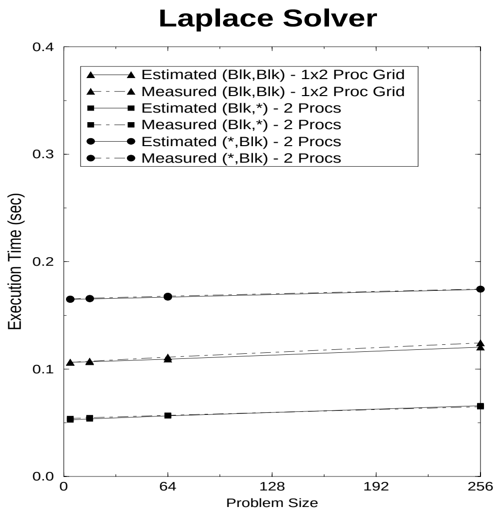


Figure 11: Laplace Solver (2 Procs) - Estimated/Measured Times

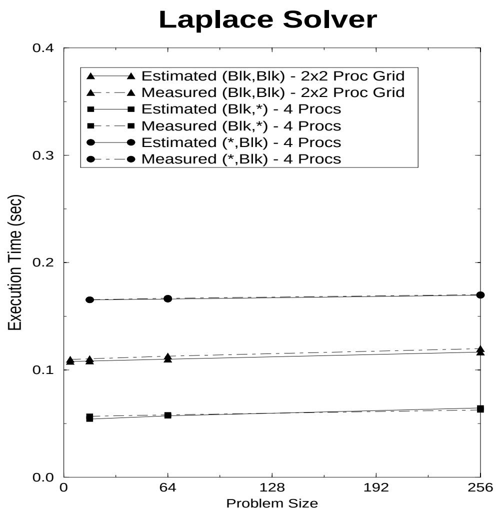


Figure 12: Laplace Solver (4 Procs) - Estimated/Measured Times

5.2 Evaluation Stage

The Evaluation stage of the HPC software development process is responsible for performing a thorough evaluation of the implementation with two key objectives:

- Identify regions of the implementation where performance improvement is possible by performance debugging the implementation and analyzing the contribution of different parts of the application description and view their computation time/communication time breakup.
- Investigate the scalability of the application with machine and problem size as well as the effect of system and run-time parameters on its performance. This enables the developer to test the robustness of the design and to modify it to account for different run-time scenarios.

The key requirement of this stage is the ability to perform the above evaluations with the desired accuracy and granularity, while maintaining tractability, non-intrusiveness, and cost-effectiveness. The use of the interpretive framework to the Evaluation stage of the HPC software development process is illustrated by the following experiments:

1. Application performance debugging.
2. Evaluation of application scalability.
3. Experimentation with system and run-time parameters.

5.2.1 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.

N-Body Computations: Figure 15 shows the performance profile for two phases of the n-body application. Phase 1 (see Figure 14) 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 forces interactions. Overhead time represents parallelization overheads. Similar profiles can be obtained at smaller granularities (upto a single line of code).

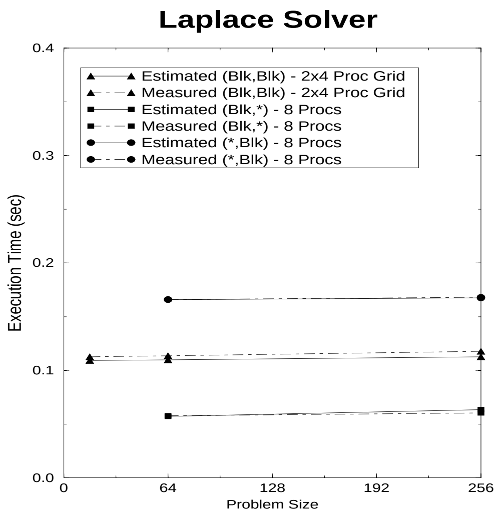


Figure 13: Laplace Solver (8 Procs) - Estimated/Measured Times

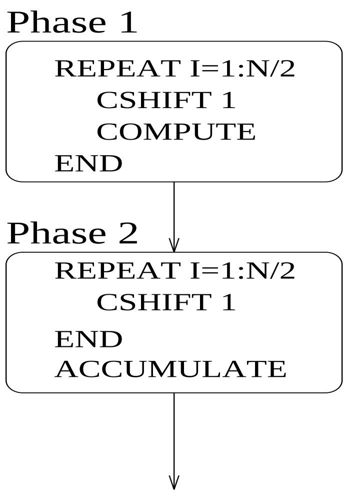


Figure 14: N-Body - Application Phases

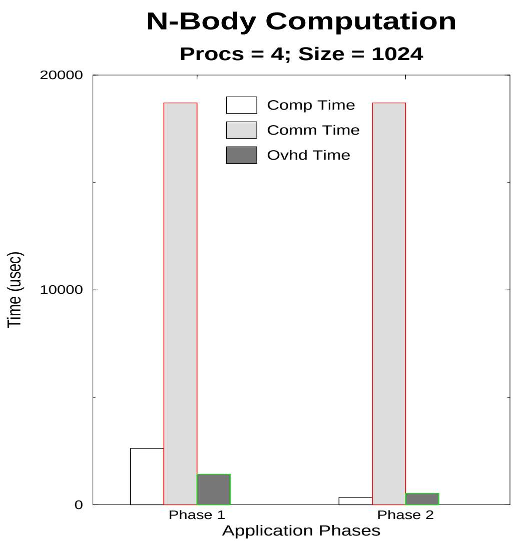


Figure 15: NBody Computation - Interpreted Performance Profile

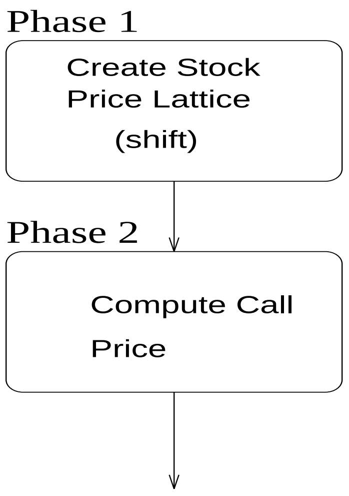


Figure 16: Financial Model - Application Phases

Parallel Stock Option Pricing: A performance profile for the parallel stock option pricing application is shown in Figure 17. This application has two phases as shown in Figures 16. 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 modification. Using the interpretive framework, this information is available, at all levels required, during application development.

5.2.2 Application Scalability Evaluation

Figures 18, 19, & 20 plot the scalability of three applications (PI, NBody and Financial) with problem and well as system sizes. Both, measured and estimated times are plotted to show that estimated times provide sufficiently accurate scalability information.

5.2.3 Experimentation with System/Run-Time Parameters

The 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 the each processor in the iPSC/860 system on application performance. The results are shown in Figure 21. Such an evaluation enables the developer to visualize how the application will perform on a faster (prospective) machine or alternately if it has be run on a slower processor. It can also be used to evaluate the benefits of upgrading to a faster processor system.

Effect of Varying Network Load: Figure 22 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 23.

Effect of Varying Interconnection Bandwidth: The effect of varying the interconnect bandwidth on the application performance is shown in Figure 24. The increase/decrease in application execution times is greater for larger processor configurations as illustrated in Figure 25.

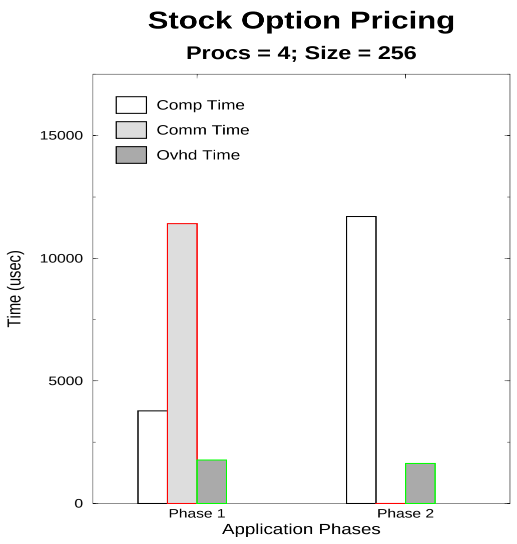


Figure 17: Financial Model - Interpreted Performance Profile

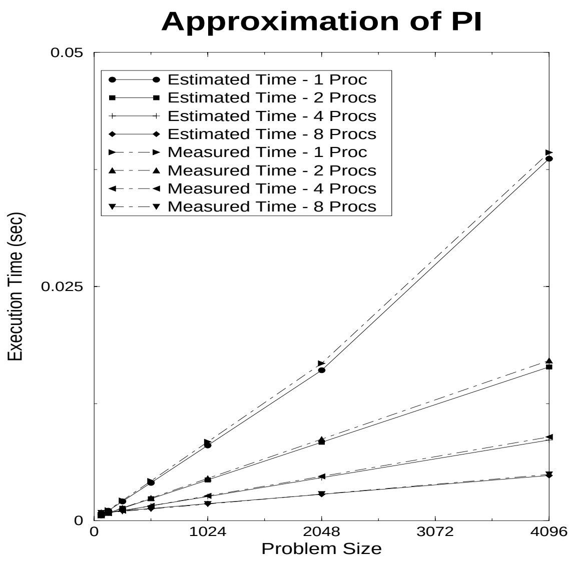


Figure 18: PI - Scalability with Problem/System Size

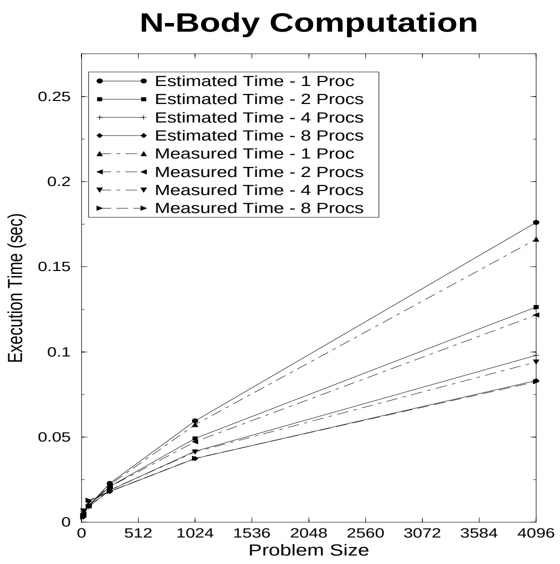


Figure 19: N-Body - Scalability with Problem/System Size

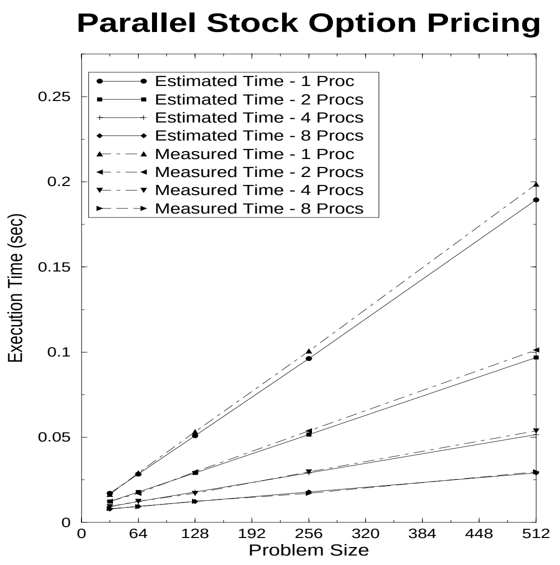


Figure 20: Financial Model - Scalability with Problem/System Size

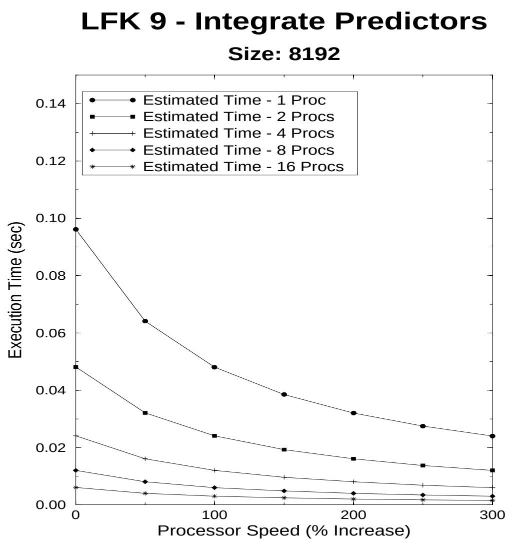


Figure 21: Effect of Increasing Processor Speed on Performance

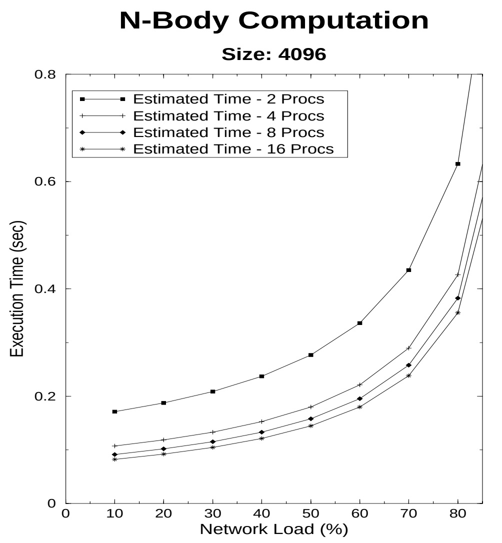


Figure 22: Effect of Increasing Network Load on Performance

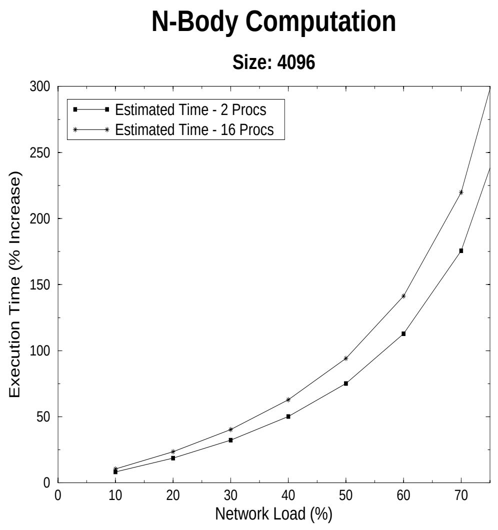


Figure 23: Effect of Varying Network Load on Performance (% Change in Execution time)

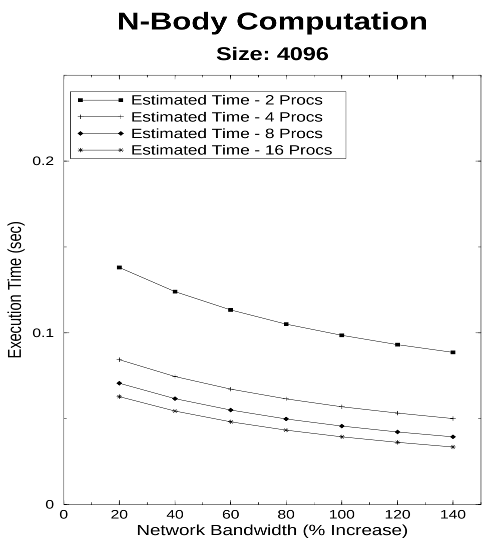


Figure 24: Effect of Increasing Network Bandwidth on Performance

Experimentation with Larger System Configurations: In this experiment we experiment with larger system configurations that physically available (i.e. 16 & 32 processors). The results are shown in Figures 26 & 27. 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 different system parameters not only allows the user to evaluate the application during the Evaluation stage, but can also be used during the Maintenance/Evolution stage to check whether the application meets its specification with changes in the system configuration.

6 Conclusions

Software development in any high-performance parallel computing environment is non-trivial and the development of efficient application software capable of exploiting available computing potentials depends to a large extent on the availability of suitable tools and application development environments. Evaluation tools enable a developer to visualize the effects of the various design alternatives and make appropriate design decisions, and thus form a critical component of such a development environment.

In this paper we first presented a novel interpretive approach for accurate and cost-effective performance prediction that can be effectively used during HPC application software development. A source-driven HPF/Fortran 90D performance prediction framework based on the interpretive approach has been implemented as part of the NPAC HPF/Fortran 90D integrated application development environment. The accuracy and usability of the interpretive performance prediction framework were experimentally validated.

We then outlined the stages typically encountered during application software development in a HPC environment and highlighted the significance and requirements of a performance prediction tool at the relevant stages. Numerical results using benchmarking kernels and application codes were presented to demonstrate the application of the performance prediction framework to different stages of the application software development process.

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. We are also working on expanding to the HPF/Fortran 90D application development environment to incorporate a wider set of tools so as to span the stages of the HPC application software development process.

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

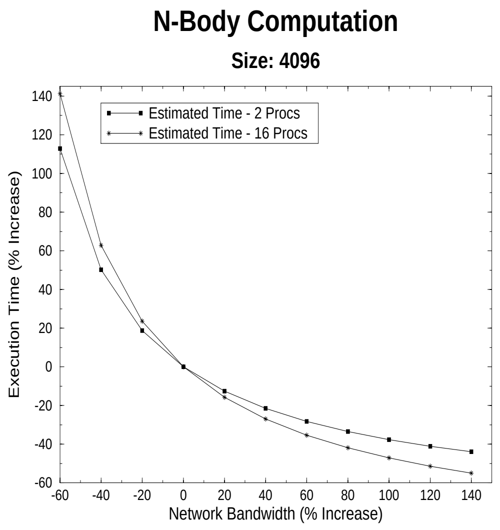


Figure 25: Effect of Varying Network Bandwidth on Performance (% Change in Execution time)

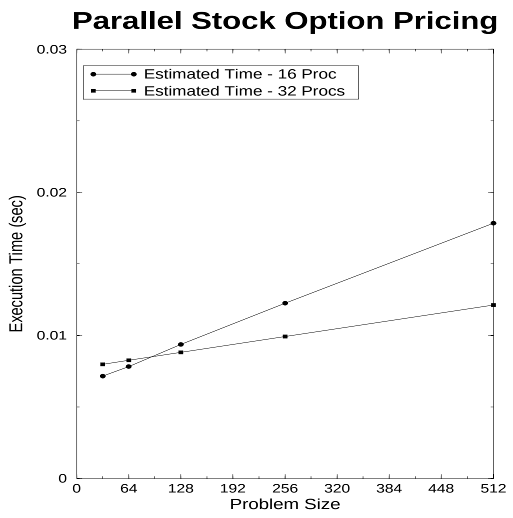


Figure 26: Experimentation with Larger System Configurations Financial Model

via WWW at `http://www.ticam.utexas.edu/~parashar/public_html/ESP/`.

[2] 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.

[3] Philip Heildelberger and Kishore S. Trivedi, "Analytic Queueing Models for Programs with Internal Concurrency", *IEEE Transactions on Computers*, **C-32**(1):73–82, Jan. 1983.

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] Manish Parashar, Salim Hariri, Tomasz Haupt, and Geoffrey C. Fox, "Design of an Application Development Toolkit for HPF/Fortran 90D", *Proceedings of the International Workshop on Parallel Processing*, Dec. 1994.

[14] 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.

[15] A. Gaber Mohamed, Geoffrey C. Fox, Gregor von Laszewski, Manish Parashar, Tomasz Haupt, Kim Mills, Ying-Hua Lu, Neng-Tan Lin, and Nang Kang Yeh, "Application Benchmark Set for Fortran-D and High Performance Fortran", Technical Report SCCS-327, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY 13244-4100., June 1992, Available via WWW at `http://www.npac.syr.edu`.

[16] Manish Parashar, Salim Hariri, Tomasz Haupt, and Geoffrey C. Fox, "A Study of Software Development for High Performance Computing", in Karsten M. Decker and Renè M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*. Birkhauser Verlag, Basel, Switzerland, Aug. 1994.

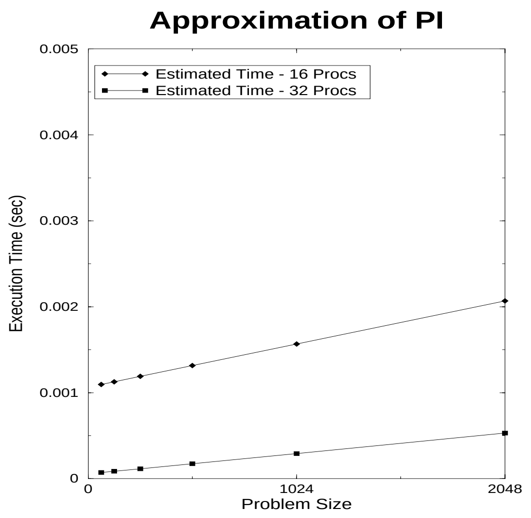


Figure 27: Experimentation with Larger System Configurations - Approximation of PI