Evolutionary Performance-Oriented Development of Parallel Programs by Composition of Components

Nasim Mahmood
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
nmtanim@cs.utexas.edu

Yusheng Feng
Institute for Computational Engineering and Sciences
University of Texas at Austin
Austin, Texas 78712
feng@ices.utexas.edu

James C. Browne
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
browne@cs.utexas.edu

ABSTRACT
This paper describes a method for evolutionary component-based development of families of parallel programs to attain performance goals on multiple execution environments for multiple family instances and an implementation of the method. It is based upon combining component-oriented development with integration of parallel/distributed execution and parallel/distributed simulation. Each component may have multiple representations at multiple levels of realization from analytical timing models to production code. Each component is encapsulated with an associative interface specifying its properties and behaviors which enables distinguishing among different implementations (or abstractions) of the same functional behavior. Evolutionary development evolves a program from an abstract performance model to a complete program and may continue evolution during runtime. Performance can be estimated at any stage of realization. The implementation is a compiler which composes parallel/distributed programs from components encapsulated with associative interfaces and a runtime system which supports integrated execution/simulation of parallel programs composed from components at different levels of abstraction and program evolution at runtime by component replacement. Case studies in the application of the evolutionary development method including performance results are given.

Categories and Subject Descriptors
C.4 [Performance of Systems]: Design studies, Modeling techniques; D.1.2 [Programming Techniques]: Automatic Programming – Program synthesis; D.1.3 [Concurrent Programming]

General Terms
Performance, Design, Languages.

Keywords
Parallel programming, performance modeling, component-oriented development, parallel/distributed simulation.

1. INTRODUCTION
1.1 Problem Statement
Designing and implementing parallel/distributed programs to meet performance requirements is still not an exact science. This problem is aggravated by the multiplicity of and constant change in parallel execution environments. Porting across execution environments with retention of efficiency often requires effort intensive redesign and re-implementation. Modern computational algorithms utilize adaptive methods where the behavior of the program may change substantially during its execution so that the performance (and accuracy) of programs optimized for the initial conditions of execution may deteriorate during execution. Conventional development methods for parallel programs where a program is fully developed before its performance properties are evaluated worsen the problem. Further, conventional monolithic program structures make evolution of parallel programs particularly difficult.

1.2 Approach and Method
This paper presents a method for design and implementation of instances of families of parallel/distributed programs whose performance properties can be predicted for arbitrary parallel/distributed execution environments and thus customized by application instance and execution environment and which can be adapted at runtime to maintain performance as program behavior and/or resource requirements change. The key insight is that the combination of a component-defined program structure where the components are self-describing enables use of components at multiple levels of abstraction in a parallel/distributed execution. The research presented here extends a method for composing parallel/distributed programs from components which has been previously reported [20] to incorporate design for performance based on evolution from abstract components to concrete components at both compile time and runtime.

Evolutionary development utilizes a component-oriented model of development and an integration of parallel/distributed
execution and parallel/distributed simulation. Each component may have multiple representations at multiple levels of realization from analytical timing models to production code. Each component is encapsulated with an interface which specifies its properties and behaviors and distinguishes among different representations of a component. Integration of parallel/distributed execution and parallel/distributed simulation enables a program to evolve from an abstract program where all components are represented as abstract timing models to a production code by systematically replacing the component models by more resolved instantiations. Performance can be estimated at any stage of realization. Evolution can be continued during runtime on a component by component basis.

The benefits of this approach include: (a) The abstract program has the same parallel structure as the concrete program thus eliminating a major source of uncertainty in the performance estimates. (b) It is straightforward to obtain simple but reasonably accurate models of most components, (c) The executions of programs realized with abstract components are very fast enabling a wide range of exploration of options (d) Automation of model construction though compiler composition of performance models removes much of the tedious effort of model development and (e) Optimal choices for component instantiations and structures are known at design time. Finally, the component-oriented approach facilitates runtime optimization.

1.3 Implementation

The implementation is a compiler which composes parallel programs from components encapsulated with associative interfaces and a runtime system which interprets associative interfaces and supports unified parallel/distributed execution/simulation of parallel programs composed from components at different levels of abstraction. The previously reported compositional compiler (P-COM\(^2\)) [20] generates a parallel/distributed programs as a precedence-constrained data dependence graph. Integration of execution behavior and parallel/distributed simulation is based on a formulation of parallel/distributed discrete event simulation as traversal of precedence constrained execution structures using a Lamport [19] clock.

There is an assumption, which has been verified in our experiments to data that the performance of parallel programs structured as data dependence graphs of components can be accurately modeled with simple timing models for the components and communication systems and analytic representation of contention for resources. Runtime adaptation/optimization is attained though monitoring on a component by component basis (the P-COM\(^2\) compiler, see Section 2, generates the monitor code), a component type specialized to analysis of monitor data and replacement of components though the dynamic linking capabilities provided by most operating systems.

---

1 Components are recursively defined. A component may be a composition of previously compiled components. This is necessary for a useful implementation but adds complexity to the definitions. No distinction will be made herein between primitive or composed components.

1.4 Contents of Paper

Section 2 describes the interface definition language, the compilation process and the runtime adaptation process. Section 3 describes the formulation and implementation of parallel/distributed simulation for precedence constrained execution structures. Section 4 first illustrates the approach with a simple example and then applies it to an hp-adaptive finite element code. Section 5 covers related research. Section 6 gives conclusions and suggests future research.

2. APPROACH: COMPONENT-ORIENTED DEVELOPMENT WITH SEMANTIC INTERFACES

This section sketches the concepts of component-oriented development, sketches the integration of parallel/distributed execution and parallel/distributed simulation and describes the implementation. A more complete specification of the interface definition language and the compositional compiler can be found in [20].

2.1 The Interface Definition Language

The concepts of the interface definition language which specifies the semantic properties of the components are sketched in the following.

**Component**: A component is one or more sequential computations, an interface which specifies the information used for selection and matching of components and a state machine which manages the interface, the interactions with other peers and the invocation of the sequential computations.

**Associative Interface**: An associative interface [4] encapsulates a component. It describes the behavior and functionality of a component. An associative interface consists of an accepts specification and a requires specification.

**Accepts Specification**: An accepts interface specifies the set of interactions in which a component is willing to participate. The accepts interface for a component is a set of three-tuples (profile, transaction, protocol).

- A profile is a set of attribute/value pairs. Components have a priori agreement on the set of attributes and values which can appear on the accepts and requires interface of a component.
- A transaction specification incorporates one or more function signatures including the data types, functionality and parameters of the unit of work to be executed and a state machine which manages the order of execution of the units of work. The state machine is defined in the form of conditional expressions over states and function signatures.
- A protocol defines a sequence of simple interactions necessary to complete the interaction specified by the profile. The most basic protocol is data-flow (continuations), which is defined as executing the functionality of a component and transmitting the output to a successor defined by the selectors at that component without returning to the invoking component

**Requires Specification**: A requires interface specifies the set of interactions which a component must initiate if it is to complete the interactions it has agreed to accept. The requires interface is a set of three-tuples (selector, transaction, protocol). A component can have multiple tuples in its requires interface to implement its required functionality.
• A **selector** is a conditional expression over the attributes of all the components in the domain.
• **Transaction** specifications are similar to those for accepts specifications.
• **Protocol** specifications are as given for accepts specifications.

**Start Component**: A start component is a component that has at least one requires interface and no accepts interface.

**Stop Component**: A stop component is a component that has at least one accepts interface and no requires interface.

**Adapt Component**: An adapt component contains the logic for utilizing the behavioral information measured in the execution of the code. The fact that the measured data can be analyzed in the context of the known semantics of the components in which the measurements are taken enables straightforward analysis and decision processes.

Extension of the interface to incorporate abstracted components requires only that each component have an attribute which specifies whether it is an abstract component for which the execution time is to be generated by the Lamport clock and a simulator or a concrete component where the execution time is the elapsed system time.

Properties of desired implementations such as degree of parallelism (number of replicas) for a given component are also specified in the associative interface as runtime determined parameters. It is often desirable for a component to retain state across executions.

### 2.2 Data Flow Graph Model of Execution

The data flow model of parallel computation which underlies the unification of execution and simulation formulates a parallel execution as a dynamic generalized data flow graph (GDFG) which is an extension of the data flow graphs in [23]. The nodes of the graph contain the actions of the program which may include a local sequential discrete event simulator. The arcs specify the dependence relations between the actions of the programs. Execution of the program is traversal of the graph. The nodes of the graph are defined as six tuples \((\text{input ports}, \text{firing rule}, \text{an initialization, a computation}, \text{routing rule}, \{\text{output ports}\})\). Input ports are containers for a typed object or data structure. A firing rule is a conditional expression over the values in the input ports of the node. A node is enabled for execution when its firing rule evaluates to true. A computation is the action associated with the node. The routing rule of a node assigns values to the output ports of a node as soon as the computation has completed an execution. A node once enabled remains enabled until the enabled execution begins. The execution of a node is run to completion. The arcs of the graph are infinite fifo queues which bind output ports of a source node to input ports of sink nodes. Execution of a program is accomplished by generation and traversal of the directed graph. The graph explicitly specifies which nodes can be executed in parallel.

### 2.3 Compilation Process

The conditional expression of a selector is a template which has slots for attribute names and values. The names and values are specified in the profiles of other components of the domain. Each attribute name in the selector expression of a component behaves as a variable. The attribute variables in a selector are instantiated with the values defined in the profile of another component. The profile and the selector are said to match when the instantiated conditional expression evaluates to true.

The source program for the compilation process is a Start component which implements initialization for the program and a requires interface which specifies the components implementing the first steps of the computation and one or more libraries to search for components. The libraries should include the components needed to compose a family of applications specified by a domain analysis. The components which are composed to form a program are dependent on the requires interface of the Start component.

The compilation process first parses the associative interface of the start component. The compiler then searches a specified list of libraries for components whose accepts interface matches with the requires interface of the start component. If the matching between the selector of one component and the profile of another component is successful, the compiler tries to match the corresponding transactions of the requires and accepts interface. The transactions are said to match when all of the following conditions are true. 1) The name of the two transactions is the same. 2) The number of arguments of each of the two transactions is the same. 3) The data type of each argument in the requires transaction is the same as that of the corresponding argument in the accepts transaction. 4) The sequencing constraint given by the conditional expression in the accepts transaction specification (the state machine) is satisfied. Finally the protocol specifications must be consistent.

The target language for the compilation process is a generalized data flow graph (GDFG) as described preceding. The GDFG has two special node types, a start node and a stop node. When compilation of the P-COM2 Start component is completed, it is converted into a start node [23] for the GDFG and each match of a requires interface to an accepts interface results in addition of a node to the data flow graph which is being incrementally constructed by the compilation process and an arc connecting the this new node to the node which is currently being processed by the compiler. If there is a replication clause in a transaction specification then at runtime the specified number of replicas of the matched component are instantiated and linked with data flow arcs. This searching and matching process for the requires interface is applied recursively to each of the components that are in the matched set. The composition process stops when no more matching of interfaces is possible which will always occur with a Stop component since a Stop component has no requires interface. Compilation of a P-COM2 Stop component results in generation of a stop node for the data flow graph. The compiler will signal an error if a requires interface cannot be matched with an accepts interface of a desired component. The generated GDFG is then compiled to a parallel program for a specific architecture by compilation processes implemented in the CODE [23] parallel programming system.

### 2.4 Runtime Adaptation

An h-p adaptive finite element code is an example of an application where the execution behavior may change material as it executes. The adaptive code may make many cycles through the basic loop of solution adaptation. The requirements of the solution process may change substantially as the solution mesh and approximating functions are locally or globally adapted. The
component-composition approach to application family development enables substitution of components implementing different algorithms during execution.

Most operating systems enable runtime linking of components to executable images. The requirement is to identify components which need to be replaced and to specify the properties of the component which is to be substituted for an existing component. Additionally the programs must be instrumented to acquire and analyze the execution properties and behavior of the components of the system.

Composition of a program from components enables and facilitates each of these tasks. Monitoring can be done on a component by component basis; components whose behavior is unlikely to vary need not be monitored. The monitoring code is readily generated by the compiler on a component by component basis. The required analysis and actions can be provided in a separate component or components which can also be selected on the basis of domain analysis or replaced at runtime if necessary.

3. PARALLEL/DISTRIBUTED SIMULATION

This section presents a data flow formulation of parallel/distributed discrete event simulation for simulation modeling of parallel/distributed systems which are formulated as precedence-constrained dynamic generalized data flow graphs and its integration with direct execution.

Sequential execution of discrete event simulation can be viewed as the generation and traversal of a dynamic, ordered list of events. Parallel execution of discrete event simulation can be viewed as generation and traversal of a directed graph of events. Parallel algorithms must partition generation and traversal of a dynamic ordered list of events into subsets while preserving a valid order of generation and traversal. Parallel/distributed executions of discrete event simulations are typically formulated under the constraint that traversal of the directed graph must conform to an order which would result from some sequential execution.

The parallel/distributed discrete event simulation model is formulated as a directed graph of nodes where the dependence relations among the nodes are an order-preserving subset of the nodes of the data flow graph of the actual system. In practice, the nodes with abstract models of the node computation are given the same firing rules as the nodes with the concrete code for the computation. Simulation time is generated by an extended Lamport clock [19] at each node in the graph. An arc carrying the simulation time of a source node to each sink node of the source node is added to the arc set of the data flow graph of the simulation model. If the firing rule is an "and" over several ports the start time for the execution of the node is taken to be the largest time from any data source in the Lamport clock computation of local time. If the firing rule is an "or" over multiple ports then the Lamport clock time computation is carried out for each invocation. The local clock for a node is updated to include the time (real or simulated) taken to execute the node computations and this local time is sent to nodes to which the node has an output arc. Causality is maintained in that the execution order will be an execution order which could have been generated by some serial execution of the actual system. No deadlock management algorithms (other than what is required for the actual system.) are necessary. Parallel speed-up of execution of the simulation is bounded by the parallel speed-up of the actual system.

4. ILLUSTRATION AND CASE STUDY

The first example a parallel solution of LaPlace’s equation indicates the accuracy to be expected when simple abstract performance models of components are used to predict performance of an application. The second example uses a mixed representation to determine a near optimal configuration for parallel execution of an hp adaptive finite element code.

4.1 Laplace Solver

The algorithm for the LaPlace solver in two dimensions is as follows:

1. The NxM matrix is partitioned row wise into P sub-matrices and the sub-matrices are sent to the P processors.
2. The shadow rows are communicated. After the communication the topmost and bottommost processor has a matrix of size N/P+1 x M and all other processors has a matrix of size N/P+2 x M.
3. Each processor performs a Jacobi iteration on its partition. A difference norm between the old values and the new values are calculated.
4. Each processor sends its value of the difference norm to a designated processor ("sum") which collects the P difference norms.
5. The “sum” processor decides whether to stop the iteration process and sends the decision message to each of the P processor.
6. If a process receives a stop iteration message it sends its partition to the “gather” processor.
7. The designated processor collects all the submatrices and compose them into a N x M matrix.
8. The solution is printed.

Five components can be identified from this algorithm a. Distribute which performs step 1 and 2, b. Jacobi: performs steps 3, 4 and 6, c. Sum which performs step 5, d. Gather which performs step 7 and e. Print which performs step 8.

Figure 1 shows the data flow graph of the program in terms of the components identified. The data flow graph (Figure 1) is shown for the case when the matrix is partitioned into three parts.
Distribute

\[ \text{Jacobi}[0] \rightarrow \text{Print} \]
\[ \text{Jacobi}[1] \rightarrow \text{Sum} \rightarrow \text{Gather} \rightarrow \text{Jacobi}[2] \]

**Figure 1: Data Flow Graph for Laplace Solver**

From the data flow graph, the data elements that have to be passed from one component to the other are identified. Abstract components are coded where the computation section is empty and is not yet implemented. The timing model for the component is added in the computation section of the abstract component to give an estimate of the runtime of the component. Communication is modeled using the size of the data elements being passed and the properties of the interconnection network. The complete program can then be run using the abstract components which gives an estimate of the runtime of the program. When the implementation of a component is complete, the concrete component can then be plugged into the program replacing the abstract component. The process of replacing an abstract component with a concrete component is continued until all the abstract components are replaced with concrete components. During the process the estimated runtime of the program gets more and more accurate and at the end of the process we have a fully functional program.

**Abstract Component Models**

All of the components in this family of applications are floating point intensive. Computation time for each component is modeled using an estimate of the number of floating point operations needed to implement the computation. The estimated time for the computation is computed by dividing that number with the FLOPS (Floating Point Operations per Second) of the processor. Normalization of the FLOPS rate for a single component is usually sufficient to give good accuracy for computation times. Communication time is modeled as the expected time to send a given number of bytes. Communication time for each message is computed as \( a + b \times x \) where \( a \) is a startup time for the communication to begin, \( b \) is the data transfer rate of the network and \( x \) is the given size of the data. The parameters \( a \) and \( b \) are estimated from measurements on the execution environment to be modeled. We have tried several versions of more sophisticated performance models for communication and have not found substantial increase in accuracy.

**Comparison of Model with actual runtime**

Table 1 shows a comparison of the estimated runtime and actual runtime for various matrix sizes and partition sizes. The measurements were taken on "Lonestar" a Cray/ Dell Linux cluster at the Texas Advanced Computer Center. The estimated runtime is for the program when all the components are abstract components. The estimated runtime is within 10% of the actual runtime in most of the cases.

<table>
<thead>
<tr>
<th>Matrix Size (nnxn)</th>
<th># of partitions</th>
<th>Estimated runtime (sec)</th>
<th>Actual runtime (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>2</td>
<td>27.979618</td>
<td>26.04458</td>
</tr>
<tr>
<td>1024</td>
<td>4</td>
<td>15.411232</td>
<td>14.234831</td>
</tr>
<tr>
<td>1024</td>
<td>8</td>
<td>9.275731</td>
<td>8.47888</td>
</tr>
<tr>
<td>1024</td>
<td>16</td>
<td>7.051624</td>
<td>6.31288</td>
</tr>
<tr>
<td>2048</td>
<td>2</td>
<td>107.157538</td>
<td>101.566281</td>
</tr>
<tr>
<td>2048</td>
<td>4</td>
<td>57.962647</td>
<td>54.137176</td>
</tr>
<tr>
<td>2048</td>
<td>8</td>
<td>47.306664</td>
<td>44.850613</td>
</tr>
<tr>
<td>2048</td>
<td>16</td>
<td>23.367203</td>
<td>21.459022</td>
</tr>
<tr>
<td>4096</td>
<td>2</td>
<td>432.709424</td>
<td>422.8589</td>
</tr>
<tr>
<td>4096</td>
<td>4</td>
<td>223.485333</td>
<td>218.343156</td>
</tr>
<tr>
<td>4096</td>
<td>8</td>
<td>178.698618</td>
<td>172.806012</td>
</tr>
<tr>
<td>4096</td>
<td>16</td>
<td>142.53143</td>
<td>136.246375</td>
</tr>
</tbody>
</table>

**4.2 Component Selection and Adaptation for an h-p Adaptive Finite Element Code**

**4.2.1 Description of the h-p Adaptive Finite Element Code**

h-p adaptive codes are an example of an application which may benefit from both customization at compile time and optimization at runtime. This case study is based on an h-p adaptive finite element code structure developed in [10][11][12]. These packages have a common data structure in one-, two-, and three-dimensional space. The data structure is hierarchical and has the ability to account for 1-irregularity in the finite element mesh. The major logical components include mesh generation, problem definition, shape function definition, and element routine, linear system of equation solver, error estimation module, and h-p adaptation module. All three packages follow the same refinement rule that uses middle node as an identifier of each element, and separates vertex nodes with other "bubble" nodes in the data structure [10][11]. The refinement strategy is a "two-grid" strategy. The fine mesh is obtained from the coarse mesh by halving element size and add one to the degree of the approximating polynomial. An improved mesh is obtained from comparison of the error in the coarse and fine meshes. We have used the one-dimensional code in this case study since it has the same structure as the two-D and three-D codes but is of considerably smaller size.

**4.2.2 Componentization of the h-p Adaptive Finite Element Code**

The set of components is determined by constructing a workflow diagram for the application in which each logical function is identified as a component.
solution of the linear systems which result from different sizes and structures for the different meshes. It may be advantageous to use a higher degree of parallelism for solution of the linear system for the fine mesh than for the coarse mesh. However, the linear system for one-dimensional finite element models are very sparse so that solution requires only modest computational work for their solution. So the overheads of communication might limit the effective degree of parallelism.

A system configuration which used concrete representations of all components except the linear solvers was executed on lonestar. For small matrices a direct solver is typically used and that was the case for the original code which we re-engineering into components. However, if the approximating polynomial is of high degree or the matrix is large, solution by an iterative method such as a conjugate gradient method can be much more efficient.

A wide range of experiments were executed ranging across mesh properties, types of linear equation solvers and degree of parallelism for the solution of the linear system from the fine mesh. Each experiment required only changing of values in requires interfaces and invocation of the compiler.

We report here the results of two experiments which lead to the important performance optimizations. The solution of the linear systems from the coarse and fine mesh were run in parallel in both of the experiments. Each of the two experiments used an initial mesh of 500 elements with the approximating polynomial for the finite elements being chosen to be of degree 2 and degree 8. The initial linear systems for the 500x2 mesh is 1001x1001 for the coarse mesh and 4001x4001 for the fine mesh while the initial linear systems for the 500x8 mesh are 4001x4001 and 9001x9001.

Experiment 1 used a direct solver for the coarse mesh and a parallel conjugate gradient solver for the fine mesh and varied the degree of parallelism for solution of the linear system from the fine mesh. For the preconditioned conjugate gradient method it is assumed that the total number of iterations required for convergence is proportional to the square root of the spectral condition number of the input matrix. Table 2 shows the results from Experiment 1.

Table 2: Estimated Execution Times for Experiment 1.

<table>
<thead>
<tr>
<th>Mesh (# of elements x polynomial degree)</th>
<th>Estimated Coarse Mesh Solution Time (sec)</th>
<th>Number of Processors for Fine Mesh Solution</th>
<th>Estimated Fine Mesh Solution Time (sec)</th>
<th>Estimated Total Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500x2</td>
<td>0.26</td>
<td>1</td>
<td>1.65</td>
<td>3.08</td>
</tr>
<tr>
<td>500x2</td>
<td>0.26</td>
<td>2</td>
<td>8.14</td>
<td>9.71</td>
</tr>
<tr>
<td>500x2</td>
<td>.026</td>
<td>4</td>
<td>27.49</td>
<td>29.76</td>
</tr>
<tr>
<td>500x8</td>
<td>13.82</td>
<td>1</td>
<td>3.93</td>
<td>18.43</td>
</tr>
<tr>
<td>500x8</td>
<td>13.82</td>
<td>2</td>
<td>11.93</td>
<td>18.47</td>
</tr>
<tr>
<td>500x8</td>
<td>13.82</td>
<td>4</td>
<td>23.15</td>
<td>27.74</td>
</tr>
</tbody>
</table>

From this experiment we conclude that there is no performance gain from parallel execution of the conjugate gradient solver on the linear system from the fine mesh and that the direct solver is a
bottleneck for larger matrices resulting from high degree approximating polynomials.

Experiment 2 replaces the direct solver for the coarse mesh with a serial implementation of the conjugate gradient solver and the parallel conjugate gradient solver for the fine mesh with this same serial conjugate gradient solver. The results of this experiment are given in Table 3.

Table 3: Estimated Execution Times for Experiment 2.

<table>
<thead>
<tr>
<th>Mesh (# of elements x polynomial degree)</th>
<th>Estimated Coarse Mesh Solution Time (sec)</th>
<th>Estimated Fine Mesh Solution Time (sec)</th>
<th>Estimated Total Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500x2</td>
<td>0.25</td>
<td>1.13</td>
<td>2.49</td>
</tr>
<tr>
<td>500x8</td>
<td>0.91</td>
<td>3.31</td>
<td>6.64</td>
</tr>
</tbody>
</table>

This experiment shows that the conjugate gradient solver is only marginally faster than the direct solver for the linear systems from meshes with low degree approximating polynomials but dramatically faster for meshes with high degree approximating polynomials.

These (and other) experiments suggest that a concrete configuration similar to the abstract configuration of Experiment 2 would be near optimal. Table 4 gives the execution times for the program with concrete components.

Table 4: Actual Execution Times for Optimal Configuration

<table>
<thead>
<tr>
<th>Mesh (# of elements x polynomial degree)</th>
<th>Coarse Mesh Solution Time (sec)</th>
<th>Fine Mesh Solution Time (sec)</th>
<th>Total Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500x2</td>
<td>0.22</td>
<td>1.19</td>
<td>2.42</td>
</tr>
<tr>
<td>500x8</td>
<td>0.86</td>
<td>3.23</td>
<td>6.25</td>
</tr>
</tbody>
</table>

The abstract performance model of the system gave quite accurate predictions of the performance of various system configurations and lead directly to a near-optimal system configuration.

4.2.4 Runtime Optimization

Runtime adaptation and optimization is accomplished by: (a) monitoring the behavior of components whose performance may degrade as the properties of the solution are modified by the adaptive algorithms (the compiler generates the monitoring code), (b) inserting an Adapt component to which the monitor data is directed to periodically analyze the behavior of the components, (c) modifying the requires interface which invokes the component to load and execute a different component.

We have conducted experiments where a direct solver for the coarse mesh was automatically replaced by a conjugate gradient solver when the mesh properties caused its execution time to exceed the execution time of the fine mesh solver. These experiments, and other experiments where we replaced the computational models in the code have been reported in a separate paper [21].

5. RELATED RESEARCH

There are at least four major research topics which intersect this paper: model-based prediction of performance, runtime optimization of parallel/distributed programs, component-oriented development and parallel/distributed simulation. The third and fourth topics are large research areas where the related research can only be lightly touched.

The most directly related research is MPI-SIM. MPI-SIM [24] predicts the performance of existing MPI programs by using direct execution to simulate sequential blocks of code and simulates a subset of MPI core functions. The simulator can run in parallel and a conservative synchronization algorithm together with a number of optimizations is used reduce the frequency and cost of synchronizations in the parallel simulator. But the simulator assumes the existence of program implementation and cannot predict the program performance at the design stage. It can, however, accurately predict the behavior of a program across multiple parallel execution environments and has been applied to several large scale parallel programs [9].

5.1 Model Based Performance Prediction

The survey paper in [3] gives a taxonomy of some existing model based performance prediction techniques. The paper classifies existing techniques in three dimensions where the dimensions are: the integration level of the software model with the performance model, the level of integration of performance analysis in the software lifecycle, and the methodology automation degree. Using the classification criterion our work falls in the category where the performance model is the same as the software model, the level of integration in the software lifecycle falls in the software design stage and the level of automation is high.

Predicting performance of computations using user input has been discussed in [25]. The user has to predict about the performance of a component and the techniques discussed in that paper can be used in asserting the prediction.

5.2 Runtime Adaptation and Optimization

There has been relatively little research utilizing dynamic replacement of components to enhance performance or robustness of scientific and engineering applications.

The Broadway annotational compiler [16] uses annotations for retaining domain specific semantics information. Using the information the compiler can choose domain specific optimization techniques. Using dynamic feedback techniques the compiler can choose dynamically the best implementation from multiple versions of optimized code. PCOM\textsuperscript{2} also uses semantic information in the form of attributes and their values. Using similar semantic information the PCOM\textsuperscript{2} compiler can choose the best component at compile time. The use of dynamic loading also enables our compiler to choose the best implementation at runtime.

Adve’s PCL [13] system enables runtime adaptation of task graphs at a finer level of granularity (the basic block level).

AspectIX [17] offers the ability to replace an implementation at runtime. The functional and configuration interface in AspectIX is
similar to the transaction and attributes of the profile in P-COM. The transaction provides the syntax of a component and the attributes express the semantics in the program domain. AspectIX uses interface information at runtime. P-COM integrates runtime and compile time composition.

5.3 Component-Oriented Development

The Common Component Architecture (CCA) project [1] is a major research and development project focused on composition of parallel programs from components. One primary goal of CCA is to enable composition of programs from components written in multiple languages. CCA has developed interface standards. The implementations of the CCA interface specifications are object-oriented. There are several tools, XCAT [15], Caffeine [7] and BABEL [2][5] implementing the CCA interface specification system. Component composition are either graphical or through scripts and make files. CCA components interact through two types of ports. The first type of port is the provides port. It is an interface that provides the information necessary to enable composition by compilation.

For additional related work in this category see [20].

5.4 Parallel/Distributed Simulation

Parallel/distributed simulation research has two main branches: conservative originating by (Chandy,Misra,Bryant) [6][8] and virtual time or optimistic originated by Jefferson [18]. These models for distributed simulation are all formulated in the context of the communicating sequential processes model of parallel computation where each process executes a local sequential discrete event simulator which processes both locally generated and externally generated events. Messages containing events to be processed by remote processes are sent on channels which implement fifo queues at the receiver. Each process maintains a local clock and each message incorporates the value of the local clock with each event which it sends for remote processing. Causality is maintained by each process constraining its execution to events which will precede in time any event which it may receive in a message. If event times are randomly generated, speed-up of the execution may be limited to log₂(n) where n is the number of processes.

Optimistic distributed simulation is also formulated in the CSP model of execution but allows processes to execute forward in time until a breach of causality is detected. Causality is maintained by processes recovering to a time before the breach of causality by rollback of message queues and state. Pure optimistic distributed simulation will process slow if breaches of causality occur more than occasionally. There has been much research on hybrid models of distributed simulation where processes “look ahead” to both progress beyond the time allowed by pure conservative simulation and to avoid most of the breaches of causality which might occur under optimistic execution. Bagrodia and his students [22][26] have carried several studies which use data flow graph based “look ahead” to improve the efficiency of parallel/distributed simulation. There have been many hybrid schemes many of which are described in Fujimoto’s comprehensive book.

The data flow precedence-constrained execution model used herein is different from the CSP-based execution model for distributed discrete event simulations in fundamental ways.

a) The simulation clock is derived from an execution order derived from the logic of the model rather than the simulation clock determining the order of execution.

b) Causality is not defined by simulation time but by consistency with a correct execution sequence which could occur in the actual system being modeled.

6. CONCLUSIONS AND FUTURE RESEARCH

The combination of a component-defined program structure where the components are self-describing and the integration of execution and simulation has been shown to enable: (a) automated support for evolutionary development of parallel/distributed programs from abstract design or performance models, (b) prediction of the performance properties of parallel/distributed programs for specific application instances and execution environments, (c) automated composition by compiler of programs which are near-optimal for given execution environments and application instances and (d) runtime adaptation/optimization of parallel/distributed programs on a component by component basis.

Future research will extend the capabilities of the compiler and runtime system to interface with hardware simulators, provide additional automation for component selection, integrate static analysis to provide estimates of component execution. Additional experimental studies on more complex systems are also in progress.

7. ACKNOWLEDGMENTS

This work was accomplished through support from ITR NSF grant number 0205181 , “A Computational Infrastructure for Reliable Computer Simulations.” NGS NSF grant number 0103725 “Performance-Driven Adaptive Software Design and Control.” Additional support came from the Defense Advanced Research Project Agency (DARPA) under Contract NBCH30390004.

8. REFERENCES


