Compositional Development of Parallel Programs

Nasim Mahmood, Guosheng Deng, and James C. Browne

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
{nmtanim,gsdeng,browne}@cs.utexas.edu

Abstract. This paper presents a programming model, an interface definition language (P-COM^2) and a compiler that composes parallel and distributed programs from independently written components. P-COM^2 specifications incorporate information on behaviors and implementations of components to enable qualification of components for effectiveness in specific application instances and execution environments. The programming model targets development of families of related programs. One objective is to be able to compose programs which are near-optimal for given application instances and execution environments. Component-oriented development is motivated for parallel and distributed computations. The programming model is defined and described and illustrated with a simple example. The compilation process is briefly defined and described. Experience with one more complex application, a generalized fast multipole solver is sketched including performance data, some of which was surprising.

1 Introduction

This paper presents a language (P-COM^2)[1] and a compiler that composes parallel and distributed programs from independently written components and illustrates their application. P-COM^2 is an interface definition language which incorporates information on behaviors and implementations of components to enable qualification of components for effectiveness in specific application instances and execution environments. The general strategy is somewhat similar to composition of programs in the Web Services paradigm but the goals are quite different. A component is a serial program which is encapsulated by an associative interface [8,11] which specifies the properties of the component. The composition implemented by the compiler is based on matching of associative interfaces and generates as final output either an MPI program or multi-threaded code for a shared memory multi-processor. The CODE [26] parallel programming system is used as an intermediate language and is the immediate target language of the compositional compiler.

Component-oriented software development is one of the most active and significant threads of research in software engineering [1,10,15,29]. There are many motivations for raising the level of abstraction of program composition from individual statements to components with substantial semantics. It is often the case that there is a family of applications which can be generated from a modest number of appropriately-defined components. Optimization and adaptation for different execution environments is readily accomplished by creating and maintaining multiple versions of components rather than by direct modifications of complete

[1] P-COM^2 stands for Parallel COMposition from COMponents.

applications. Programs generated and maintained as compositions of components are much more understandable and thus much more readily modifiable and maintainable.

Even though there are additional benefits to component-oriented development in the distributed and parallel domain[2], there has been relatively little research on component based programming in the context of high performance parallel and distributed programming. (Section 8 summarizes related work.) The execution environments for parallel programs are much more diverse than those for sequential programs. It is often necessary to maintain multiple versions of parallel programs for different execution environments. Program development by composition of components enables adaptation of parallel programs to different execution environments and optimization for different application instances by replacement of components. Adaptive control of parallel and distributed programs [3] is also enabled by replacement of components. Management of adaptations such as degree of parallelism and load balancing are readily accomplished at the component level. Parallelism is most often determined by the number of instances of a component which are executing in parallel (SPMD parallelism). The P-COM^2 language and the compiler explicitly make provision for dynamic SPMD parallelism It has also been found that viewing programs as compositions of components tends to lead to programs with better structuring and better performance even for sequential versions.

P-COM^2 approaches component-oriented development of parallel and distributed programs from a different perspective than most other projects. The principal concerns and goals for the P-COM^2 project have been to enable automation or at least partial automation of composition through a compiler, to develop a mechanism enabling runtime adaptation of parallel and distributed programs at the component level [3] and to enable performance-oriented, evolutionary development of parallel and distributed programs. This paper covers the first topic, compiler-implemented composition. The P-COM^2 interface definition language incorporates information on component properties and behaviors as well as function/procedure/method interfaces including an implicit state machine to sequence invocations of components with internal state. Additionally the P-COM^2 system targets development of families of programs with instances of the family targeting given application instances or given execution environments.

The P-COM^2 language and compiler have been used in implementing some substantial programs. One of the applications is to construct components and compose programs for solving linear equations using a fast multipole solver (FMM). The FMM code can be formulated in either a memory intensive or computation intensive formulation and at points in between. It is complex to write a parameterized program spanning these options but they are readily composed from parameterized components. The compiler has also been applied in the composition of parallel method of lines (MOL) codes for solving time dependent partial differential equations. MOL also has a great number of possible configurations and runtime adaptations.

The remainder of the paper is organized in the following way. Section 2 explains some terms and concepts used in the compiler. Next, the programming model, the language and the compilation process are described in section 3, 4, and 5 respectively. Then a simple program, a macro-parallel FFT algorithm [32], is used to introduce the programming model, the programming language (which is an interface definition language) and the compilation process in section 6. The components and compilation process and a short discussion of the FMM code is given in Section 7. Section 8 discusses related work in this area. The paper is concluded and some future directions are discussed in Section 9.

---

[2] CORBA, Web Services, etc. which are very much component-oriented development systems, are not commonly used for development of parallel or high performance applications.

2 Definition of Terms and Concepts

Domain Analysis: Domain analysis [5] identifies the components from which a family of programs in the domain can be constructed and identifies a set of attributes in which the properties and behaviors of the components can be defined. It is usually the case that applications require components from multiple domains.

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. An interaction, which may be initiated as an incoming message (or set of messages) or as an invocation of a transaction, will trigger an action which is associated with some state of the state machine. The action may include execution of a sequential computation.

Sequential Computation: A computation is a unit of work that implements some atomic functionality. A computation is a sequential program which refers only to its own local variables and its input variables.

Associative Interface: An associative interface [8] encapsulates a component. It describes the behavior and functionality of a component. One of the most important properties of associative interfaces is that they enable differentiation among alternative implementations of the same component. These interfaces are called "associative" because selection and matching is similar to operations on content-addressable memories. An associative interface consists of an **accepts** specification and a **requests** 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 requests 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 transaction can be enabled or disabled based on its current state and its current state can be used in runtime binding of the components. Multiple transactions controlled by the state machine can be used to represent complex interactions such as precedence of transactions, "and" relationships among transactions acting as a barrier and "or" relationships between transactions representing alternative ways of executing the component.
- 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. More complex interaction protocols such as call-return and persistent transactions are planned but not yet implemented.

Requests Specification: A requests interface specifies the set of interactions which a component must initiate if it is to complete the interactions it has agreed to accept. The requests interface is a set of three-tuples (selector, transaction, protocol). A component can have multiple tuples in its requests 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 requests interface and no accepts interface. Every program requires a start component. There can be only one start component in a program which provides a starting point for the program.

Stop Component: A stop component is a component that has at least one accepts interface and no requests interface. A stop component is also a requirement for termination of a program. There can be more than one stop component of a program denoting multiple ending points for the program.

3 Programming Model

The domain-based, component-oriented programming model targets development of a family of programs rather than a single program. The programming model has two phases: development of families of components and specification of instances from the family of programs which can be instantiated from the sets of components.

3.1 Component Development

The set of components which enables construction of a family of application programs may include components which utilize different algorithms for different problem instances or different implementation strategies for different execution environments. A program for a given problem instance or given execution environment is composed from appropriate components by selecting desired properties for the components and the properties of the execution environment in the Start component. The steps for developing components are:

a. Domain Analysis – Execute the necessary domain analyses. It is usually the case that applications require components from multiple domains.
b. Component Development – Specify and either design and implement or discover in existing libraries, the family of components identified in the domain analysis in an appropriate sequential procedural language.
c. Encapsulate – Encapsulate the components in the P-COM^2 interface definition language using the attributes identified in the domain analysis to specify associative interfaces for the components. The interfaces must differentiate the components by identifying their properties in terms of the attributes defined in the domain analysis.

3.2 Program Instance Development

The steps in specifying a given instance of an application are:

a. Analyze the problem instance and the target execution environment. Identify the attributes and attribute values which characterize the components desired for this problem instance and execution environment.
b. Identify the components from which the application instance will be composed. If the needed components are not available then some additional implementations of components may be necessary together with an extension of the domain analysis.
c. Identify the dependence graph of the application instance. The dependence graph is expressed in terms of the components identified. Specify the number of replications desired for parallelism and for fault-tolerance. Incorporate these specifications into the

component interfaces or as parameters in the Start component if parameterized parallelism has been incorporated into the component interfaces.

d. Define a Start component which initializes the replication parameters, sets attribute values needed to ensure that the desired components are selected and matched.
e. Define at least one Stop component.

4 The Interface Definition Language- P-COM^2

The fundamental concepts underlying the interface definition language were given in Section 2. The syntax will be illustrated in the example in Section 6. Here we discuss what is expressed in the interfaces specifiable in the language.

The language is rooted on the domain analyses for the program family. The domain analyses specify problem domain knowledge. It is expected that an application developer should be able, once familiar with the concepts of domain analysis, to generate domain analyses for a family of codes in her/his area of expertise. The associative interfaces define the behaviors of the components and will usually give properties of a given component's implementation of its functionality. Properties of desired implementations such as degree of parallelism 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. There may be precedence or sequencing relations among the transactions implemented by a component. Precedence and sequencing information is also specified in the interface as an implicit state machine implemented as a conditional expression over the states of the components and the transaction specifications. Finally, the protocol specification enables choice among interaction modes (Although only one is currently implemented).

5 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 with a sequential computation which implements initialization for the program and a requests 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 set of components which is composed to form a program is primarily dependent on the requests interface of the start component.

The target language for the compilation process is a generalized data flow graph as defined in [26]. A node in this data flow graph consists of an initialization, a firing rule, a sequential computation and a routing rule for distribution of the outputs of the computation. There are two special node types, a start node and a stop node. Acceptable data flow graphs must begin with a start node and terminate on a stop node.

The compilation process starts by parsing the associative interface of the start component. The compiler then searches a specified list of libraries for components whose accepts interface matches with the requests interface of the start component. The matching process is actually not much more than a sophisticated type matching. 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 requests 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 requests 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.

When compilation of the start component is completed, it is converted into a start node [26] for the data flow graph which will represent the parallel program and each match of a requests 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 requests 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 requests interface. Compilation of a P-COM^2 stop component results in generation of a stop node for the data flow graph. The compiler will signal an error if a requests interface cannot be matched with an accepts interface of a desired component. The data flow graph which has been generated is then compiled to a parallel program for a specific architecture by compilation processes implemented in the CODE [26] parallel programming system.

6 Example Program

This section presents an example program showing the complete process of developing a parallel program for the fast Fourier transformation (FFT) of a matrix in two dimensions from simple components. The algorithm presented is an adaptation of Swarztrauber's multiprocessor FFT algorithm [32]. This problem is simple enough to cover in detail and illustrates many of the important concepts such as stateful components and precedence constraints. Given an N x M matrix of complex numbers where both N and M are powers of 2, we want to compute the 2D FFT of the complex matrix. This 2D FFT can be described in terms of 1D FFTs, which helps in parallelizing the algorithm. Let us assume that there are P available processors where P is also a power of 2. In this case the domain analysis is straightforward and is an analysis of the algorithm itself. The steps of the algorithm are following:

a) Partitioning the matrix row wise (horizontally) into P submatrices, one for each processor.
b) Sending these submatrices to each of the P processors for computation. The size of each the submatrix is N/P x M.
c) Each processor performs a 1D FFT on every row of the submatrix that it received.
d) Collecting these 1D FFT's and then transposing the N x M matrix. The resulting matrix is of size M x N.
e) Splitting the M x N matrix row wise into P submatrices. The size of each of the submatrix is M/P x N.
f) Sending these submatrices to the each of the P processors for computation.
g) Again each processor performs a 1-D FFT on every row of the submatrix that it received.

h) Collecting all the submatrices from the P processors and transposing the M x N matrix to get an N x M matrix. The resulting N x M matrix is the 2D FFT of the original matrix.

This simple analysis suggests that all of the instances of this algorithm can be created from composing instances of three components: a one-dimensional FFT component, a component which partitions and distributes matrices and a component which merges rows or columns to recover a matrix and which may optionally transpose the recovered matrix. Let us name the components as fft_row, distribute, and gather_transpose respectively. One could as well formulate the algorithm with separate components for merge and transpose but that could introduce additional communication. Or the algorithm can use any 1D FFT algorithm to calculate the 2D FFT of the matrix. Additionally the choice of implementation for transposition of an array may vary with execution environment. Note that each of the components above can be reused as each of them is actually used twice in the algorithm. These components could reasonably be expected to be found as "off the shelf" component which can be found and reused from linear algebra and fft libraries. Other than the above three components we need a component that will read/initialize the matrix and one component to print out the final result. Let us name the component as initialize and print. The component to read/initialize the array may be the Start component and the print component may be the Stop component. The Start component will be written to specify the set of component instances which will be composed for a given data set and target execution environment.

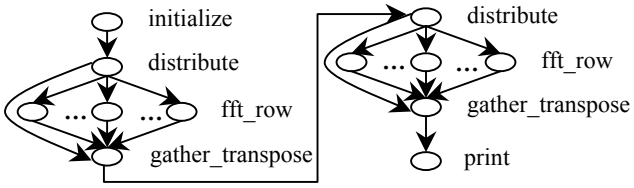

Fig. 1. Data Flow Graph of 2D FFT Computation

The depencence graph of the program in terms of these components is shown in Figure 1. This data flow graph suggests an optimization of creating a new component which combines the functions of distribute and gather_transpose. This depending on the mapping of nodes to processors, could eliminate two transmissions of the large matrix. As shown in Figure 1, parallelism can be achieved through the use of multiple fft_row components. Note that the gather_transpose component has to keep track of its state as it sends data to the distribute component on its first execution and to the print component after its second execution.

Once we have identified the components, the next step is to complete the domain analysis by defining a list of attributes through which we can describe the functions, behaviors and implementations of a component and their instantiations. When some service is required it is described in terms of the attributes in the format of accepts and requests interfaces.

The two domains from which this computation is composed are the matrix and fft domains. There is a generic attribute "Domain" which is required for multi-domain problems. The matrix domain has these distinct attributes:

a) Function: an attribute of type string. Describes its function.
b) Element_type: an attribute of type string. Describes the type information of the input matrix.
c) Distribute_by_row: an attribute of type boolean. Describes whether the component partitions the matrix by row or by col.

The fft domain has these attributes:

a) Input: an attribute of type string. Describes the input structure.

b) Element_type: an attribute of type string. Describes the type information of the input.
c) Algorithm: an attribute of type string.
d) Apply_per_row: an attribute of type boolean. Describes whether to apply the FFT function per row or per column.

The completed domain analysis for the components is shown in Figure 2. Once the domain analysis is done, we encapsulate the components in associative interfaces using the attributes and transactions.

As shown in Figure 3, the requests interface of the initialize component specifies that it needs a component that can distribute a matrix row-wise. The interface passes real and imaginary parts of the matrix, the dimension of the matrix and the total number of processors to the distribute component using the transaction specification. The data type mat2 is defined as a two dimensional array data type.

Fft_row a) Domain: fft b) Input: matrix c) Element_type: complex d) Algorithm: 1d-fft e) Apply_per_row: true	Gather_transpose a) Domain: matrix b) Function: gather c) Element_type: complex d) Combine_by_row: true e) Transpose: true
Distribute a) Domain: matrix b) Function: distribute c) Element_type: complex d) Distribute_by_row: true	Print a) Domain: print b) Input: matrix c) Element_type: complex

Fig. 2. Domain Analysis of the Components

Figure 4a shows the accepts interface of the distribute component. This distribute component assumes that the matrix which it partitions and distributes will be merged. This is specified in Figure 4b. The first selector interfaces to the gather_transpose component providing the size of each of the submatrices, the total number of submatrices to collect at the gather_transpose component and also state information which is needed in the gather_transpose component. The second selector in Figure 4b specifies that it needs p instances of the fft_row component and distributes the submatrices to each of the replicated components along with their size. The construct "index [p]" is used to specify that multiple copy of the fft_row component are needed. The construct "[]" with the transaction argument is used to transmit different data to different copies of component. For different transmission patterns, different constructs may be used in the language of the interface. Note that the number of instances of the fft_row component is determined at runtime.

```
selector:
  string domain == "matrix";
  string function == "distribute";
  string element_type == "complex";
  bool distribute_by_row == true;
transaction:
  int get_matrix(out mat2 grid_re,out mat2 grid_im, out
                 int n, out int m, out int p);
protocol: dataflow;
```

Fig. 3. Requests Interface of Initialize Component

```
profile:
  string domain = "matrix";
  string function = "distribute";
  string element_type = "complex";
  bool distribute_by_row = true;
transaction:
  int get_matrix(in mat2 grid_re,in mat2 grid_im, in int n,
                 in int m, in int p);
protocol: dataflow;
```

Fig. 4a. Accepts Interface of distribute component

Figure 5a specifies that this implementation of fft_row component uses the "Cooley-Tukey" algorithm [13]. The fft_row component requires no knowledge of how many copies of itself are being used. From Figure 5b, we can see that the instance number of the fft_row component is passed to the gather_transpose component using the variable "me".

Figure 6a illustrates the use of the ">" operator between the transactions to describe the precedence relationship between the transactions. The second transaction cannot execute until the first transaction is completed. The gather_transpose component collects the submatrices one by one through the second transaction in the interface. P-COM2 incorporates precedence ordering operations sufficient to express simple state machines for management of interactions among components.

```
selector:
  string domain == "matrix";
  string function == "gather";
  string element_type == "complex";
  bool combine_by_row == true;
  bool transpose == true;
transaction:
  int get_p(out int n/p, out int m, out int p,
            out int state);
protocol: dataflow;
{selector:
  string domain == "fft";
  string input == "matrix";
  string element_type == "complex";
  string algorithm == "Cooley-Tukey";
  bool apply_per_row == true;
transaction:
  int get_grid_n_m(out mat2 out_grid_re[],out mat2
                   out_grid_im[], out int n/p, out int m);
protocol: dataflow;
}index [ p ]
```

Fig. 4b. Requests Interface of distribute component

As shown in Figure 6b, the first requests interface of the gather_transpose component is used to connect to the distribute component. The second interface connects to the print component. The variable “state” is used to enable one of the transactions based on the current state of the gather_transpose component.

```
profile:
  string domain = "fft";
  string input = "matrix";
  string element_type = "complex";
  string algorithm = "Cooley-Tukey";
  bool apply_per_row = true;
transaction :
  int get_grid_n_m(in mat2 grid_re,in mat2 grid_im,in int n,
                   in int m);
protocol: dataflow;
```

Fig. 5a. Accepts Interface of Fft_row Component

```
selector:
  string domain == "matrix";
  string function == "gather";
  string element_type == "complex";
  bool combine_by_row == true;
  bool transpose == true;
transaction:
  int get_grid_n_m_inst(out mat2 out_grid_re,out mat2
                        out_grid_im, out int me);
protocol: dataflow;
```

Fig. 5b. Requests Interface of Fft_row Component

```
profile:
  string domain = "matrix";
  string function = "gather";
  string element_type = "complex";
  bool combine_by_row = true;
  bool transpose = true;
  transaction:
  int get_p(in int n, in int m, in int p,in int state);
   >
  int get_grid_n_m_inst(in mat2 grid_re,in mat2 grid_im,
                        in int inst);
protocol: dataflow;
```

Fig. 6a. Accepts Interface of Gather_transpose Component

```
selector:
  string domain == "matrix";
  string function == "distribute";
  string element_type == "complex";
  bool distribute_by_row == true;
transaction:
  %{  state == 1, gathered == p }%
  int get_matrix(out mat2 out_grid_re,out mat2 out_grid_im,
                 out int m, out int n*p, out int p);
protocol: dataflow;
selector:
  string domain == "print";
  string input == "matrix";
  string element_type == "complex";
transaction:
  %{  state == 2, gathered == p }%
  int get_grid_n_m(out mat2 out_grid_re,out mat2
                   out_grid_im, out int m,out int n*p);
protocol: dataflow;
```

Fig. 6b. Requests Interface of Gather_transpose Component

7 Case Study - A Generalized Fast Multipole Solver

The Fast Multipole Method (FMM) [20,21], which solves the N-body electrostatics problems in O(N) rather than $O(N^2)$ operations, is central to fast computational strategies for particle simulations. The FMM is also useful for iterative solution of linear algebraic equations associated with approximate solution of integral equations. There the FMM is used for O(N) matrix-vector multiplication. In order to adapt the FMM for applications in fluid and solid mechanics, the classical electrostatics problem must be replaced with a generalized electrostatics problem [17,18]. Such problems involve vector and tensor valued charges, which means that one generalized electrostatics problem is equivalent to several classical electrostatics problems, which share the same geometry. In particular, FLEMS code [17] relies

on the generalized electrostatics problem that is equivalent to 13 classical electrostatics problems.

We have performed a domain analysis for the FMM for generalized (multiple charge type) electrostatics. For example, the FMM tree has certain attributes, such as its depth and its number of charges per cell and the application component has an attribute with values that select between classical and generalized electrostatics. For generalized electrostatics the number of charge types is an attribute. For each attribute, the analysis defines a range of legal values. Components for a family of FMM codes for generalized electrostatics were derived from the FLEMS FMM implementation. These components were given associative interfaces that define their properties and behaviors and were annotated with domain attributes and architectural attributes. An instance of the component family can be specified by providing specific values for each attribute. An example of an attribute that would lead to different implementations is the number of charge types to be processed simultaneously.

There are a family of space-computation tradeoffs which can be applied in the matrix-structured formulation [30] of the FMM algorithm which can be chosen to optimize the code for a given execution environment and problem specification. These include:

- Simultaneous computation of cell potentials for multiple charge types.
- Use of optimized library routines for vector-matrix multiply.
- Use of optimized library routines for matrix-matrix multiply.
- Loop interchange over the two outer loops to improve locality (Within a component).
- Number of terms in the multipole expansion.

There are many variants of these structures and interactions among them. The original FMM implementation in the FLEMS code is approximately 4500 lines in length with the logic distributed throughout the code. Manual construction of optimized versions for even a modest number of execution environments would lead to rather complex code. But a small number (eight) of components characterized by the number of charges which are simultaneously computed and the number of terms in the multipole expansion suffice to realize an important subset of execution environment optimized codes.

The FMM includes five translation theorems:

- Particle charge to Multipole (P2M is applied at the finest partitioning level)
- Multipole to Multipole (M2M is applied at all partitioning levels, from the finest to the coarsest)
- Multipole to Local (M2L is applied at all partitioning levels)
- Local to Local (L2L is applied at all partitioning levels, from the coarsest to the finest)
- Local to Particle potential and forces (L2P is applied at the finest partitioning level)

Two kinds of components are needed structure the FMM computation framework. The first category comes directly from the FMM algorithm. The five translation theorems, charges-to-multipole, multipole-to-multipole, multipole-to-local, local-to-local, local-to-potential and force, and direct-interaction calculation belong to this category. The second category contains the communication components, distribute and collect which actually also derive from the FMM algorithm since they implement distribution and collection according to the interaction lists for each partition of the domain..

The data flow graph for the FMM code for two processors is shown in Figure 7.

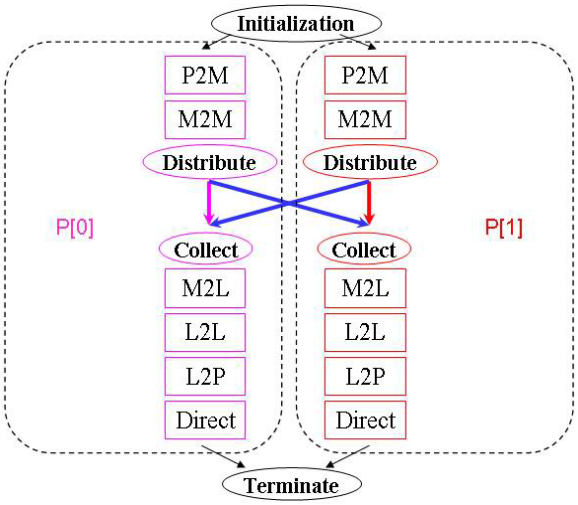

Fig. 7. Data flow Graph of FMM code

An extensive set of performance studies were made comparing the original and componentized sequential codes. Preliminary results are reported [16] and a more detailed paper is in preparation. The performance of the sequential componentized code, contrary to conventional wisdom, is up to 15 times faster than the original implementation which had itself been optimized by several generations of students and post-doctoral fellows. This surprising result is largely due to specialization of functionality based on selection of optimal components and replacing loop implementations of matrix-matrix multiply by BLAS implementations of matrix-matrix multiply. Table 1 shows a small sample of the performance data obtained. The data was taken on a Linux cluster of Pentium III's at 1.8 Gigahertz and a 100MB Ethernet interconnect. There are approximately half a million charges in this system. There are two factors to be noted: (i) Speedup is near-linear for the small number of processors and (ii) the time increases less than linearly with the number of charge types due to the change due to optimizations local to components.

Table 1. Performance data for tree depth of four.

Number of Charge Types	Run time on 2 processors (Seconds)	Run time on 4 processors (Seconds)	Run time on 8 processors (Seconds)
5	413.84	215.52	121.11
12	561.53	305.50	254.14

8 Related Research

There has been relatively little research on component based programming in the context of parallel and distributed program. Darwin [25] is a composition and configuration language for parallel and distributed programs. Darwin uses a configuration script to compose programs from components. This composition process is effectively manual. In our approach, the composition information encapsulates the components themselves, as a result the compiler can choose the required component automatically.

The component-based software development environment [23,28] of the SciRun project feature powerful graphical composition of data flow graphs of components which are compiled to parallel programs. H2O [31] is a component-oriented framework for composition of distributed programs based on web services. Triana [33] is a graphical development environment for composing distributed programs from components targeting peer to peer execution environments. The G2 [24] composes distributed parallel programs from web services through Microsoft .Net. Armada [27] composes distributed/parallel programs specialized to data movement and filtering.

The Common Component Architecture (CCA) project [6] 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, [19] Ccaffeine [14] and BABEL [7,9] 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. The provides port is an interface that components provide to other components. The second type of port is the uses port. It is an interface through which components connects with other components which they require. These port type exhibit some similarities to the accepts and requests transaction specifications. However, the details and implementations are quite different as we have focused on incorporation of the information necessary to enable composition by compilation.

ArchJava [4] annotates ports with provides and requires methods which helps the programmer to better understand the dependency relations among components by exposing it to the programmer. The accepts and requests interface of a P-COM^2 component incorporate signatures as do ArchJava provides and requires. The accepts and requests interfaces also include profiles and precedence specification carrying semantic information and enabling automatic program composition. The attribute name/value pairs in profiles are used for both selecting and matching components thereby providing a semantics-based matching in addition to type checking of the matching interfaces.

The use of associative interface has been reported earlier in the literature. Associative interface is used in one broadcast based coordination model [12]. This model uses run time composition, whereas our paper presents compile time composition. Associative interfaces have also been reported in composition of performance modeling [11].

9 Conclusion and Future Research

This paper has presented a programming model, a programming system and a compiler for composing distributed and parallel programs from independently written components. The conceptual foundations are domain analysis, support for families of programs, integration and automation of discovery and linking and management of components with state.

The component-based development method described and illustrated in this paper is not intended for development of small or "one-off" applications. The investment of effort in domain model development and characterization and encapsulation of components is not trivial and these software engineering methods are not typically a part of the development process for high performance applications. The target applications are those where several instances of an application are to be developed, where the application may need to be optimized for several different execution environments or where the application is expected to evolve over a substantial period of time. In such cases the investment of effort in domain model development and characterization and encapsulation of components can be expected to show

return. That being said, the parallel programs which have been developed to demonstrate and evaluate the method show good performance and are readily evolvable.

We are currently investigating the feasibility of combining runtime [12] and compile-time composition of associative interfaces. We plan to implement a hybrid graphical composition and compiler-based composition system. We also plan to integrate the compositional compiler with the Broadway annotational compiler [22] to overcome the problem of "too many components." Finally we are working on additional applications including an hp-adaptive finite element code.

Acknowledgements

This research was supported by the National Science Foundation under grant number 0103725 "Performance-Driven Adaptive Software Design and Control" and grant number 0205181, "A Computational Infrastructure for Reliable Computer Simulations." The experiments were run on the facilities of the Texas Advanced Computation Center.

References

1. Achermann F., Lumpe M., et al., Piccola - a Small Composition Language, in *Formal Methods for Distributed Processing - A Survey of Object-Oriented Approaches*, pp. 403-426, Cambridge University Press, 2001.
2. Adve V. S., Bagrodia R., et al., POEMS: End-to-end Performance Design of Large Parallel Adaptive Computational Systems, in *IEEE Transactions on Software Engineering*, vol. 26(11): pp. 1027-1049, November 2000.
3. Adve V., Akinsanmi A., et al., Model-Based Control of Adaptive Applications: an Overview, in *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, April 2002.
4. Aldrich J., Chambers C., et al., ArchJava: connecting software architecture to implementation, in *Proceedings of the 22nd International Conference on Software Engineering,* pp. 187-197, May 2002.
5. Arango G., Domain Analysis: From Art Form to Engineering Discipline, in *Proceedings of the Fifth International Workshop on Software Specification and Design*, pp. 152-159, 1989.
6. Armstrong R., Gannon D., et al., Toward a Common Component Architecture for High-performance Scientific Computing, in *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation*, pp. 115-124, August 1999.
7. Babel: Components @ LLNL, `http://www.llnl.gov/CASC/components/babel.html.`
8. Bayerdorffer B., Associative Broadcast and the Communication Semantics of Naming in Concurrent Systems, *Ph.D. Dissertation,* Dept. of Computer Sciences, University of Texas at Austin, December 1993.
9. Bernholdt D. E., Elwasif W. R., et al., A Component Architecture for High-Performance Computing, in *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*, June 2002.
10. Birngruber D., Coml: Yet another, but simple component composition language, in *Workshop on Composition Languages*, *WCL'01*, pp. 1–13, September 2001.
11. Browne J. C. and Dube A., Compositional Development of Performance Models in POEMS, in *International Journal of High-Performance Computing Applications*, vol. 14(4), Winter 2000.

12. Browne J. C., Kane K., et al., An Associative Broadcast Based Coordination Model for Distributed Processes, in *Proceedings of COORDINATION 2002,* LNCS 2315, pp. 96-110, 2002.
13. Cooley J. W. and Tukey J. W., An Algorithm for the Machine Computation of the Complex Fourier Series, in *Mathematics of Computation*, vol. 9, pp. 297-301, April 1965.
14. Ccaffeine - a CCA component framework for parallel computing, `http://www.cca-forum.org/ccafe/`
15. Czarnecki K. and Eisenecker U. W., Components and Generative Programming, in *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering,* Springer-Verlag LNCS 1687, pp. 2-19, 1999.
16. Deng G., New approaches for FMM implementation, *Masters Thesis*, Dept. of Manufacturing Systems Engineering, University of Texas at Austin, August 2002.
17. Fu Y., Klimkowski K. J., et al., A fast solution method for three-dimensional many-particle problems of linear elasticity, in *International Journal for Numerical Methods in Engineering*, vol. 42(7): pp. 1215-1229, 1998.
18. Fu Y. and Rodin G. J., Fast solution method for three dimensional Stokesian many-particle problems, in *Commun. Numer. Meth.* Engng, vol. 16(2): pp. 145-149, 2000.
19. Govindaraju M., Krishnan S., et al., Merging the CCA Component Model with the OGSI Framework, in *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (CCGrid2003)*, pp. 182-189, May 2003.
20. Greengard L. and Rokhlin V., A fast algorithm for particle simulations, in *Journal of Computational Physics*, vol. 73(2): pp. 325-348, 1987.
21. Greengard L. and Rokhlin V., A new version of the fast multipole method for the Laplace equation on three dimensions, in *Acta Numerica*, vol. 6: pp. 229-270, 1997.
22. Guyer S. and Lin C., An Annotation Language for Optimizing Software Libraries, in *Proceedings of the Second Conference on Domain Specific Languages*, pp. 39-53, October 1999.
23. Johnson C.R., Parker S., et al., Component-Based Problem Solving Environments for Large-Scale Scientific Computing, in *Journal on Concurrency and Computation: Practice and Experience*, vol. 14, pp. 1337-1349, 2002.
24. Kelly W., Roe P., et al., An Enhanced Programming Model for Internet Based Cycle Stealing, in *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1649-1655, June 2003.
25. Magee J., Dulay N., et al., Structuring parallel and distributed programs, in *Software Engineering Journal*, vol. 8(2): pp. 73-82, March 1993.
26. Newton P. and Browne J. C., The CODE 2.0 Graphical Parallel Programming Language, in *Proceedings of the ACM International Conference on Supercomputing*, July 1992.
27. Oldfield R. and Kotz. David, Armada: a parallel I/O framework for computational grids, *Future Generation Computer Systems*, vol. 18(4), pp. 501-523, 2002.
28. Parker S. G., A Component-based Architecture for Parallel Multi-Physics PDE Simulation, in *Proceedings of the International Conference on Computational Science*, Springer-Verlag LNCS 2331, pp. 719-734, April 2002.
29. Seiter L., Mezini M., et al., Dynamic component gluing, in *OOPSLA Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems*, November 1999.
30. Sun X. and Pitsianis N, A Matrix Version of the Fast Multipole Method, in *Siam Review*, vol. 43(2): pp. 289-300, 2001.
31. Sunderam V. and Kurzyniec D., Lightweight Self-Organizing Frameworks for Metacomputing, in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 (HPDC'02)*, pp. 113-124, July 2002.

32. Swarztrauber P. N., Multiprocessor FFTs, in *Journal of Parallel Computing*, vol. 5: pp. 197-210, 1987.
33. Taylor I., Shields M., et al., Distributed P2P Computing within Triana: A Galaxy Visualization Test Case, in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003.