

Self-Describing Components: A Programming Model for Multicore Systems

James C. Browne and Nasim Mahmood
Department of Computer Sciences
University of Texas
Austin, TX 78701
{browne,nmtanim@cs.utexas.edu}
512-471-9579
512-471-8885 (fax)

Abstract

Composition of self-describing components is suggested as a programming model for multicore systems and clusters of multicore systems. Self-describing components and an automated composition process are briefly specified and discussed. Solutions to problems such as definition of granularity of units of computation, formulation of parallel structures and resource constrained scheduling of multi-core processing nodes are sketched.

1. Introduction

It is commonly agreed that the conventional programming model where programmers explicitly program parallelism as segments of code or object methods through process and thread creation is inadequate for the emerging generation of multicore processors where a single node may have 4-64 processor cores. The conventional approach of decomposing shared data structures to determine parallel structures is error-prone. It will be much more difficult for programmers to evaluate the consequences of their design decisions. Close coupling of multiple resource types (memory bandwidth, cache, network bandwidth, processing power, etc.) may cause saturation of one resource of a multicore system while other resources are quite plentiful.

Parallel programming complexity is a multi-facet issue, it includes, but is not restricted to: (1) defining logical units of computation for parallel execution, (2) sharing and/or communicating data among executing units of computation (threads through locks assuming a shared memory model or communication by messages for distributed memory models), (3) finding the appropriate parallelism granularity for a given problem instance and execution environment, (4) mapping tasks to processes and/or threads, (5) adapting program structure across phases of computation with different requirements for parallelism and (6) for clusters of multicore systems, mapping tasks to nodes. Currently programmers are generally responsible for all of these tasks. Neither programming methodologies nor compilers provide much practical support or automation for any of these tasks. As parallel programming becomes the norm rather than the exception, automation of these tasks becomes essential.

The succeeding sections sketch how solutions to some of these problems including methodology support and automation of some tasks can be cast in terms of a parallel program development based on self-describing components. The P-COM² system [1,2 and 3] is an implementation of these concepts. P-COM² consists of a component specification language, a compiler which implements automated composition of parallel programs from self-describing components, a mapping process for assigning components to processors and a runtime system which supports component replacement and enables execution of programs where components may be abstract performance models or fully realized components.

2. Self-Describing Components

A self-describing component is a serial program that is encapsulated with a specification that specifies the properties (including resource use) and behaviors of the component. A component specification consists of an *accepts* interface and a *requests* interface. An *accepts* interface specifies the set of interactions in which a component is willing to participate and the behavioral contracts for each interaction. It incorporates a profile (a profile is a set of values for attributes in which the properties and behaviors of the component can be specified) which describes the properties and behaviors of the component including the differences between implementations of a given functionality, signatures for the set of transactions (functions) which it implements, a state machine to sequence the receipt of messages and initiation of interactions and a flow protocol. A *requests* interface specifies the set of interactions that a component must have satisfied if it is to complete the interactions it has agreed to accept. It incorporates a set of selectors (a selector is a conditional expression over the attributes which may appear in profiles) defining the profiles and transactions (function signatures) for the components it requires and may incorporate a state machine for sequencing the interactions. Space limitation precludes illustrating the component specification. Examples and more details of component specifications can be found in [1, 2, and 3] which can be found at the url <http://www.cs.utexas.edu/~nmtanim>.

3. Automated Composition

Composition begins with a *start* component which has a requests interface but no accepts interface. The start component binds values to a set of attributes which establish a given implementation of each component in its *requests* interface. 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 operations and protocol must also match for a component match to occur. A match results in the compiler generating code for a connector with properties specified in the protocol and the state machines. The component matching algorithm has been extended to implement containment relations on profile attributes. A containment relation can be defined for each attribute in a profile. A containment relation ($A \gg B$) specifies that the functionality of A is a superset of the functionality of B and that A can be substituted for B if a component implementing B is not available. Program composition terminates on an *end* component which has an accepts interface but no requests interface.

4. Identification of Granularity and Specification of Parallel Structures

The methodology gives an approach for specification of components as parameterized functions. Instances of parameterized components are the units of parallel computation. Granularity is commonly determined by values for size parameters for data structures of the component instance. The parameters determining the degree of parallelism can be adjusted at runtime to increase or decrease the amount of parallelism.

The compiler first generates a generalized data flow graph [4] where the components are nodes (nodes are assignable units of computation (ACUs)). The scheduler then assigns the nodes of the data flow graph to processors. The structure and dimensions of the data flow graph are determined by the compiler from parameters supplied by the programmer. The compiler can

generate either threaded code where the interactions among components are via shared variables or MPI-based message passing code or both. Programmers are thus freed from having to explicitly specify the details of parallel structure.

4. Scheduling

The resource requirements of a component can be included in the specifications for a component. The computation and communication requirements of a component can be described by expressions parameterized by the input and output arguments of the function signatures. For example, the computational requirements for a component implementing a matrix computation can be estimated by an expression of the computational complexity of the computation and a scaling constant. The communication requirements can be estimated by an expression over the size of the input and output arguments of the function signature and a scaling constant.¹ For data parallel computations these arguments will be known prior to a given execution. Let us assume for the sake of definiteness that the program is a data parallel program with a single phase and is not adaptive. A preprocessor can then generate resource requirement estimates for each ACU. Given this information a scheduler can attempt to optimize placement of ACUs to resources according to specified metrics. For example, a scheduler could estimate the memory access bandwidth requirement for each ACU as well as its compute requirements and determine how many of a given ACU to map to a given multicore node. A more sophisticated analysis could estimate cache requirements.

The current scheduling system in P-COM² uses a set of heuristics to minimize communication costs but otherwise does not use resource requirement information. This simple scheduler could be extended with more sophisticated algorithms which utilize information on execution environments and component resource requirements. It is therefore straightforward to incorporate near-optimal scheduling² in P-COM².

5. Adaptation

The compiler generates monitoring code on a component by component basis. This data can be directed to an *adapt* component which determines when degrees of parallelism should be changed or when a component is not performing properly and should be replaced by a different implementation. Components can be replaced by use of the dynamic loading facilities provided by most operating systems.

6. Summary and Conclusions

A programming model based on self-describing components enables automated solutions to some of the more difficult tasks of parallel programming and is particularly well adapted to programming of multicore systems.

7. Related Research

The most directly related research is [5] which applies component-oriented programming methods where the parallelization constructs are specified in an annotation language which is

¹ We have found that even these simple models yield highly accurate performance predictions when carefully applied.

² There is a vast literature on optimal scheduling of data flow graphs to processor configurations.

translated to Java threads/locks, etc. The programmer must write the parallelization code and not capabilities for optimal scheduling are given.

But there is a vast array of related research. Component-oriented development [6] is probably the most active branch of research in software engineering. Service-oriented architectures and some grid programming systems are also based on concepts very similar to component-oriented development. Brief surveys of the most related research can be found in [1,2 and 3].

8. Acknowledgements

This research was supported by NSF Grant Number 0438967 “SoD Collaborative Research: Constraint-based Architecture Evaluation” and NSF Grant Number CNS-0540033 “A Dynamic Data-Driven System for Laser Treatment of Cancer.

9. References

- [1] Mahmood, N., Deng, G., and Browne, J. C. Compositional Development of Parallel Programs, *Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pp. 109-126, College Station, TX, 2-4 October 2003.
- [2] Mahmood, N., Feng, Y., and Browne, J. C. A Case Study in Application Family Development by Automated Component Composition: h-p Adaptive Finite Element Codes, *Proceedings of the International Conference on Computational Science (ICCS'05)*, pp. 347-354 Atlanta, GA, 22-25 May 2005.
- [3] Mahmood, N., Feng, Y., and Browne, J. C. Evolutionary Performance-Oriented Development of Parallel Programs by Composition of Components, *Proceedings of the 5th International Workshop on Software and Performance (WOSP'05)*, pp. 239-248, Palma de Mallorca, Spain, 11-15 July 2005.
- [4] 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.
- [5] P. Palatin, Y. Lhuilier and O. Temem CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006) pp. **247-258**
- [6] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.