High Performance Cluster Computing using Component-oriented Distributed Systems

V. K. Murthy
School of Business Information Technology
RMIT University, Melbourne 3000, Victoria , Australia
kris.murthy@rmit.edu.au

Abstract

Component oriented distributed computing uses a collection of different types of components to achieve high performance in solving a problem by identifying each component as an object. The usefulness of the transactional paradigm in Component Oriented Programming (COP) is explained. We also describe a simplified version of COP (a master-slave-like computation) that is suitable for high performance cluster computing and indicate how to implement this paradigm using MPI.

1. Introduction

One of the fundamental themes of software engineering is the reuse or sharing of the parts of software already available. In early days of computing, this took the form of subroutines which provided code reuse in application. The subroutine libraries then provided code sharing across applications. Then the development of object oriented method permitted not only code reuse but also tailorable code via inheritance, encapsulation and polymorphism. The advent of the Client/server model then permitted sharing data across different platforms, while remote procedure call (RPC) enabled us to share code across platforms. Currently distributed object technologies such as CORBA and DCOM permit sharing tailorable code across platforms [14,15,18-20]. The notion of reuse, sharing and tailorability of codes and data across platforms led to the natural evolution of stand alone objects called " components" that are platform and language independent so that they can be plugged and played across networks, applications, languages , tools and operating systems.

Component technology is evolving as a key technology in Software engineering aiding the development of very complex software that can be tested and maintained easily. It is also playing an important role in various areas of research and development for advanced software, particularly in object oriented programming, Object oriented data and knowledge bases (OODKB) and object oriented software engineering [3,4,14,15,16,18-21]. The diversity of these areas suggests that there are underlying basic principles and issues that are common to a wide range of component-based software development. This paper addresses these basic principles and the issues involved in developing efficient software based on component oriented systems and seamless programming with heterogeneous components.

In object oriented computing (OOC) a problem is modelled as a set of cooperating objects, and is solved by exchanging messages among objects. In concurrent programming (CP) , a problem is modelled as a set of cooperating processes. Therefore OOC and CP have a similar structure; objects correspond to processes and message passing corresponds to inter-process communication. A process is not a self-contained module. In order to facilitate modular programming, object oriented programming combines the object and process into an integrated unit (of data and procedures) which is self contained; hence called a component (C). A component oriented program (COP) is interpreted as a collection of interacting components that steps through a program and manipulates data. Each component maintains its own share of data and has its own program piece to manipulate it. That is each component combines datastructure and functionality. The components are active and behave like actors in a movie, each following its own script and interacting with other components. A function call is separated into two tasks: a message passing to a component and the component executes in response a required procedure. The selection of a method to a message is called binding. By modelling a program as a set of cooperating heterogeneous component programs we can obtain portability across platforms and interoperability across operating systems and languages. This paradigm uses the technical skills arising from the three major areas in computer science: object-oriented methodology, concurrent programming [1, 3, 6, 7, 8, 17- 21], and the transactional / workflow paradigm.

2. Distributed Systems and Components

Distributed systems are essentially multi-tier client server systems in which the number of clients and servers are potentially very large and the distinction between the client and server becomes diffuse, each playing the dual role. In addition, the distributed systems offer directory services that enable objects to locate other objects, transactions and related business services. Therefore, it is convenient to define a distributed system in the most general way as one made up of components with the following properties [14,15,17-21]:

1.Marketable object: A component is a marketable entity which is a self contained, shrink-wrapped object.

2.Grain size: It is not a complete application; yet it can perform a set of limited tasks within an application domain and can be combined with other components to form a complete application. In this sense the grain size can vary from a fine grained object such as a C++ object to a medium grained object like a Graphic User Interface (GUI) control.

3.Building Block: It can be used like a building block that can be chiselled and used in a variety of ways to achieve unpredictable combinations for different applications.

4.Well-specified Interface: The interface exposes the component to the external world and hence should be well specified using an interface definition language (IDL). The interface defines the protocol of communication between two separate components of a system. The interface describes what services are provided by a component and the protocol for using those services.

5. Toolability: It permits tailorability (chiselling) and provides facilities for drag and drop and other visual assembly techniques.

6. Event Notification: It has the capability to notify an event to the external world, if some interesting event arises.

7. Configuration and Property management: Components have states. Property is a well defined attribute that can be read in order to modify the state of the component.

8. Scripting: Interface can be controlled via scripting languages.

9.Metadata: Contains information about itself: interfaces, properties, events, quality of services and contracts- that is its claims as to what it can be used for.

10. Interoperability: It can be invoked as an object across address spaces, languages, operating systems and tools. It is a system independent software entity.

11.Communication Topology: The components are connected by a communication network of a well-defined static / dynamic topology

12. Heterogeneous and recursive: In particular, we do not want any restriction on the nature of the component or its grain size of functionality - they can be different computers or software objects; that is they are heterogeneous.

13. Multi-threadedness and Serializability [9]: Since a component server can be accessed by multiple clients at a time it is essential that the component has the capability to start new thread of execution for each new client. Otherwise the response time can be very poor for many applications involving internet or E-Commerce services [10].

14. Persistence: In the database context we require to have persistence of the state.

15. Security: It must protect itself and its resources from outside intrusion, provide access controls, authentication of itself and its clients, and maintain audit trails.

3. Components are chiselled Objects

It is important to remember that components are chiselled out of objects that are not bound to a particular platform or a computer language. They are specifically designed for distributed applications. This inherent object infrastructure permits the components to be autonomous self-managing and collaborative (competitive or cooperative). In the discussion below we can therefore use components and objects interchangeably.

3.1. Definitions of objects

1. The state variables of an object are the variables which represent the internal persistent state of the object.

2. An object accepts a message which matches some message pattern and satisfies the corresponding constraint.

3. When a message arrives, message patterns and constraints are examined. After accepting the message

the object executes a sequence of actions described in the corresponding behaviour description part (script).

3.2. Properties of objects

The Concurrent objects (CO) have the following important properties:

1. They can react autonomously to changes in internal state and to events in its environment.
2. They are capable of executing multiple activities concurrently, including event detection.
3. They respond to detected events asynchronously.
4. Persistence: means that the state of the object should survive a session in which it was generated.
5. They can return values as a reply to a message received. The reactive capability of an active object is specified in terms of production rules or event-condition action rules. An event determines when the rule should be fired, the conditions whether the action should be executed and the action part determines how the object should respond.
6. Every message sent by an object arrives at the destination in a finite time and gets stored as a unique queue in a buffer associated with that destination object.
7. There is no global clock. However, timestamps are assumed to be generated and the clocks are synchronized using cause-effect relationship among objects while passing messages [1], [9], [17-20].

4. Components and Transactional Paradigm

The transactional paradigm (TP) has the following features:

1. In concurrent object programming, the rule conditions are matched and rule actions are performed on each object locally as well as on the external objects. The state of computation consists of a collection of named values in an active set of objects, where the names correspond to variables and the values are assigned from the problem domain. A state maps the variable to its corresponding value. The initial state specifies the initial condition of the problem, while the final state specifies the result. The rule actions activate each object through a set of internal actions and acts on the external objects through message passing. The internal and external actions should have the four properties - called ACID properties: Atomicity (indivisibility and either all or no actions or carried out), Consistency (before and after the execution of a transaction), Isolation (no interference among the actions), Durability (recovery under failure and achieving consistency). The transactional approach provides for the ACID properties [1,9, 21].
2. Also TP provides for cooperation among competing actions or processes, by resolving conflicts among the objects due to data dependence and resource dependence.
3. We can deal with both passive objects and active processes and achieve a very complex set of computations using the syntactic model of the transaction.
4. The serializability notion to ensure total temporal order is well-defined and so concurrent (or partially ordered) operations can take place using the well-known concurrency control techniques - such as locks, timestamps to indicate priority and obsolescence.
5. The logic of transactional paradigm takes into account the side effects due to performing action x before and doing action y after. That is action x serves as precondition for action y, and realizing action y is a post condition for action x. In practical terms, it sets up an a priori consistency and this ensures serializability of transactions in component based systems.
6. The notion of serializability is essentially concerned with the conflict equivalence of an interleaved schedule to a serial schedule (namely, the conflicting actions in the non- rolledback transactions are performed in the same temporal order). Hence it ensures a priori consistency in a competitive environment .
7. Also TP provides for reccovery when there is a failure.

5. Components and Condition-Event System

The Condition- event system (C-E System) [7] with the syntax:
ON event IF (precondition) DO (action) occupies a prominent place in component-oriented computations. Each object uses a script that consists of a set of rewrite rules consisting of a left-hand-side expression (LHS) and a right-hand side- actions (RHS). Given any message string that matches the LHS, the corresponding RHS actions are implemented. Thus we may carry out many different operations. The C-E system within an object operates in three-phase cycles: matching, selecting and execution. The cycle halts when a termination condition is reached . The task of match phase is similar to query matching - that is unification of the rules with the database. This phase returns a conflict set that satisfies the conditions of different rules. In the select phase we select those compatible rules after conflict resolution. In the execution phase all selected rules are fired and actions are implemented.

5.1. Achieving different types of parallelism

In component oriented programming (COP) we can achieve parallelism thus:
1. Concurrent multiple activation of independent objects: Although the message transmission is sequential, when the receiver objects are different the activation of the receiver objects can overlap in time.
2.While messages are sequentially received , a message can be sent simultaneously to several objects.
3.Parallelism between the actions of an object which sends a request message and the actions of an object that receives a message can be permitted depending upon the context.
4.The nature of internal production rules, events and actions determine whether an object reacts deterministically, nondeterministically or probabilistically [12,13]. This enables us to assign probabilities for applying the rule, assign strength to each rule by using a measure of its past success, introduce a support for each rule by using a measure of its likely relevance to the current situation.

The above four factors provide for competition and cooperation among the different rules. Also, the introduction of probabilistic choices in an object system would provide a computational model (such as the genetic algorithm) to simulate evolutionary biological, chemical and physical systems based on intermittent feedback from the environment and understand how intelligent behaviour can emerge from probabilistic interactions between many objects.

6. Formalizing Component-Oriented Systems

We now formalize a Component-oriented system (COS): A COS is a -tuple: (O,T, s(0)) where :O is a finite set of m objects; T is a finite set of global transactions; s(0) is the initial state. Every object O(j) is characterised by a pair (V(j) ,Op(j)) where V(j) is the set of all possible values for the object (its domain) and Op(j) is the set of local operations that can be performed on that object O(j). Each operation Op(j) is a partial function taking input values from the domain and outputting values from the domain V(j), thus changing the O(j) to a new state. The set of all possible state values is called the phase space of O(j) and its elements are called phase space elements(pse) of O(j) .

A global transaction (called External transaction or EXTRAN) T(ij) is defined as a transaction between two objects O(i) and O(j) ; this consists of a message sent from O(i) to execute a desired transaction in O(j); this message is received by O(j) . O(j) has a behaviour specified by: Pre(T(ij)), G(j), C(j), Post (T(ij)), where Pre() and Post() are respectively the pre and post states that are active before and after the transaction T(ij). G(j) is a guard of O(j) and C(j) is the command function consisting of operations that map values to values in local domains (note that the operations used in G(j) and C(j) are assumed to be defined) and sending messages. Thus the script specifies what message O(j) can accept and what actions it performs when it receives the message while in state Pre(T(ij)) to satisfy the post condition post(T(ij)). The Extran T(ij) can trigger in O(j) numeric, symbolic or database computations; hence, it provides for "Heterogeneous Computing". Each Extran T(ij) triggers a set of serializable computations in O(j) either in a total order or in a partially order depending upon whether parallelism , concurrency and interleavings are possible locally within O(j). If the object O(j) is "made up" of subobjects , we may have to execute a long transaction consisting of nested local transactions (called internal transaction - INTRAN). After executing Intran the system reaches a new state s' from old state s such that : $s' = s - pre(T(ij)) \cup post\ T(ij)$, using the command set C(j). It is possible to systematically derive a COP using a set of rules [5,8,11, 21].

7. Transactional Execution

In COP the state of computation consists of a collection of named values in an active database, where the names correspond to variables and the values are assigned from the problem domain. A state maps the variable to its corresponding value. The initial state specifies the initial condition of the problem, while the final state specifies the result. The rule actions activate the database D through an internal transaction (INTRAN) and acts on the external objects through an external transaction (EXTRAN) . In order to execute these transactions concurrently they must satisfy the following conditions:
1.The set of objects accessed by any two different EXTRAN are pairwise- disjoint.
2. The set of local states used by two different INTRAN are pairwise disjoint.
Condition 1 is well-known for those familiar with database transaction handling;
Condition 2 arises from the mutual exclusion of processes used in concurrent programming.

8. Performance Analysis of COP

The performance of a COP depends upon the choice a suitable topological sort among the objects that minimizes a certain objective. The usual scheduling objectives depend on the completion times of the jobs in the schedule, which also depends on the availability of resources.

For improved performance of a COP we must consider the optimal scheduling problem of a set of inter transactions on a set of objects where transactions are executed, and also a set of additional resources - such as registers/ cache that are required during their execution. A topological sort of a conflict multigraph provides only an abstract partial order among the different transactions that result in a serializable COP. Since it is not unique, we need to choose that topological sort which is optimal and allocate transactions to objects in such a way to minimize traffic and computation time.

9. Simplified COP

We proposed a very general component-oriented programming paradigm which includes many different computational features. In practice, many of these features can be suppressed and the structure of a COP and the corresponding protocols can be greatly simplified to suit a common application area using the following features :

1. Each component has a well defined metadata and contract for a given application.
2. Each object can be active or inactive.
3. Initially all objects are inactive except for a specified one (called the seeding object), which initiates the protocol (computation).
4. An active object can do local computation, send and receive messages and can spontaneously become inactive.
5. An inactive object becomes active if and only if it receives a message.
6. Each object may retain its current state or revise its state as a result of receiving a new message . If it revises its state, it communicates its revised state to other concerned objects.

10. Using MPI for COP

MPI [6,18] is a standard message passing interface for parallel applications and library programming. The CO programming paradigm outlined here can be implemented using MPI. The basic content of MPI is point to point communication between pairs of objects and collective communication within groups of objects. These respectively correspond to Extran and Intran in our formalism. Also MPI contains advanced message passing features .The Extran features can be realised using the various point to point message passing routines with the basic operations *send* and *receive*. Here each object can execute its own code in SIMD (single instruction multiple data mode) or MIMD (multiple-instruction multiple data mode) or SPMD (single program multiple data mode) that is an extension of SIMD and a restriction on MIMD.The collective routines can realise all Intran features that provides for barrier synchronization, broadcast, gather, scatter , and reduction operations (prefix operations such as- max, min, sum, product, exor) that can perform a parallel reduction operation over every group of processes. Also since MPI provides for topological structure for process groups (within an object), we can use this approach to map processors to local processes with a specified topology. In the current MPI version there is no facility for global serialization. The timestamping technique can be incorporated to ensure global serialization and also rollback for recovery in the MPI.

11. Scalability, Performance and Problem Domain Knowledge

We illustrate the use of component-based computation and scalability for the Generalized matrix inversion which is very important for a wide variety of applications. We can achieve supercomputer performance for matrix inversion of large rectangular matrices. However. for most problems, it is essential to have the problem-domain knowledge for achieving best performance, as illustrated below, where the knowledge of the conditioning number of the matrix is essential. For rectangular matrices (mxn) or singular matrices , we can define a Moore-Penrose generalized inverse X satisfying all of the four properties: $AXA = A$; $XAX = X$; $(AX)^t = AX$; $(XA)^t = XA$, where A^t is the transpose of A. Moore-Penrose inverse is denoted by A^+which is unique.If A is nonsingular $A^+ = X = A^{-1}$ and is unique. Here we use the matrix squaring algorithm [2] to find the rank of A and also find A^+.

11.1. Matrix Squaring Algorithm

This algorithm computes both the generalized inverse A^+ as well as the rank of a rectangular real matrix A (m x n) using successive squaring of the associated matrix T which is an (m+n) x (m+n) matrix given by:

$$T = \begin{bmatrix} P & Q \\ 0 & I \end{bmatrix}$$

where P= (I-bA*A) and Q= bA*, where A* is the complex conjugate transpose of A and b is a relaxation parameter in the range 0 < b < 2/ (maximum eigenvalue of A*A.A suitable choice for b is 1/trace (A*A) where trace (A*A) denotes the sum of diagonal elements of A*A .Starting with T(0)= T , and T(i+1) = $T^2(i)$, by successive squaring we obtain:

$$T(k) = \begin{bmatrix} M & N \\ 0 & I \end{bmatrix} .$$

Here N = A^+, rank (A) = n - Trace (M) and T(k) = T^{2^k}.

The number of iterations (squarings) k reflects the amount of computational work needed. This is dependent upon the condition number of A [2].

11.2. Matrix G-Inversion in a Component Cluster

In order to carry out the G-Inversion using the above algorithm in a cluster of workstations, we will use the master/slave organization of the components. The task of the Master is to distribute the matrix T to slave components and to get back from them the squared matrix. The slaves essentially have the contract to perform the matrix-vector multiplication .The protocol for the ***Master and Slave algorithm*** is given below:

Master:

1. Create Slave components (slaves).
2. Send matrix T to all the slaves
3. Send two consecutive columns of T to each slave (This speeds up the process, since a slave can start doing the next multiplication instead of waiting).
4. Receive a column of the squared matrix SQ(T).
5. Send a new column of T to the slave which has sent the column of SQ(T).
6. Repeat steps 4 and 5 until whole matrix has been sent and whole SQ(T) has been received.
7. Repeat Steps 1 to 6 replacing T by SQ(T), k times, until T(k) is computed.
8. Issue termination message to Slaves.

Slave:

1. Receive matrix T from master.
2. Receive two columns of T from the Master.
(The earlier column is multiplied by T; while following column is held in the message buffer of the component)
3. Multiply T by a column of T, to obtain a column of SQ(T).
4. Send Column SQ(T) to Master.
5. Repeat Steps until the Termination message is received from the Master.

The overlapping of processing and communication is achieved by sending two columns at one time from master so that the slave can begin computing one column and after sending the result can compute the product of T with the other column. any new column arriving would be stored in the buffer.

Computing SQ(T) :

Let us assume that there are P+1 components (processors) in the cluster made up of one Master and P Slaves. First let us compute the work load for computing SQ(T) of an NxN matrix T. The master processor sends the matrix T to all processors and then sends each column of T to each of the P processor to compute a matrix-vector product; the products are then sent back to the master. Communication time is needed for broadcasting the matrix T initially to all the processors and then sending each slave processor one column of T and getting back the result columns of SQ(T). That is totally we need messages to transmit N columns two times, and use (N**2)/P inner products distributed in P processors. In the sequential case we need time (N**2) c where c is the time for computing inner products of two vectors. Thus the ratio of time for sequential to parallel computation is

E (SQ(T)) = (N**2)c /[(B(T) + 2t N+ (c.N**2 /P)] .

Here B(T) is the time needed to send T to all processors, t is the time for transmission of a column of T. Assuming that B(T)= tN, we get

E (SQ(T)) = (N**2)c / [3tN+(c .N** 2 /P) = P/ [1+ 3Pt / c.N].

To get maximal efficiency close to P, we need the second term in the denominator to be significantly less than 1; that is : t << cN/ 3P.

Since Nc is equivalent to the time for computing N inner products (or computing one column of the result), the time for transmission of a column t must be much less than the time to compute each column of the product matrix. If c= 10**-5, N=32, Nc= 3.10**-4, and P= 4 then we need t < 10**-5. As N increases we get more efficiency. For small N, and small P , if P = N and t = c efficiency may fall to P/4. In fact, the sequential computation can be faster for small N and t/c > 1. For a large N , small P < N and t (transmission time) << c (computation time for innerproduct we can get near full efficiency for squaring a matrix T.

Computing G-Inverse by Recursive SQ(T).

We need the following total work for computing T(k) = T^{2^k}:

1. (k-1) transmissions for sending T(1),... T(k-1).This uses up a
total time (k-1) tN where t is the transmission time for a column.
2. (k-1)N column transmission time to Slaves using a time (k-1)tN,
where t is the transmission time for a column.
3. kN column return transmission time from Slaves using a
time ktN
4. Also we compute (k-1). N **2 inner products using a total time c. (k-1)N**2
where c is the time for computing inner products of two vectors.

In the Sequential computation case we need (k-1) c. N **2 time for computing the inner products. In the case of master-slave computation we have a speed-up
E (G-Inv) =[(k-1)c N**2] / {(k-1)tN+ktN+ [c(k-1)N**2] /P}= P/[1+3Pt/c.N], for k >>1. Thus the scalability carries over to G-Inversion (Successive squaring) if t < < cN/ 3P and we get maximal efficiency close to P. Note that for k=2 we get the expression for E(SQ(T)).

12. Concluding Remarks

Component oriented programming has the following features:

1.Provides for high concurrency, easy tailoring and maintenance.
2.Provides for the application of locality principle in program construction. Formal specification and refinement calculus can be used to provide for the choice of appropriate granularity of transactions and the level of parallelism [9, 21]. Due to the availability of object-object communications we can specify a communication network that is isomorphic to program communication and provide for the most efficient mapping topology.
3. Provides a general-purpose paradigm for programming.
4.Provides for Collective communications and computations.
5. Provides for Heterogeneous computing [6,18].
6. MPI/CORBA: The current version of MPI [6], [18] does not guarantee global serializability. The timestamp method could be incorporated to enhance its usefulness. Also CORBA and Java applications can be used.
7. Agent based Systems: At the next higher level ,the more powerful member in the hierarchy of programming paradigms is the agent-based paradigm where agents are specialised components that are mobile and autonomous. This will be described elsewhere.

13. References

[1] P.A.Bernstein, V.Hadzilacos, and N. Goodman, Concurrency Control in Database Systems, Addison Wesley, Reading, Mass., 1987.
[2] L. Chen, E.V. Krishnamurthy, and I. Macleod, "Generalized matrix Inversion and rank computation by successive matrix powering", Parallel Computing Vol. 20**,** 297-311, 1994.
[3] R.K.Ege, Programming in an Object oriented Environment, Academic Press, New York, 1992.
[4] K.Futasugi and S.Matsuoka, Object Technologies for Advanced Software,,Lecture Notes in Computer Science,Vol.1049, Springer Verlag, New York, 1996.
[5] R.F.Gamble, G.C.Roman, W.E.Ball, and H.C. Cunningham, : Applying formal verification methods to rule-based systems, International J. Expert Systems Research Applications, Vol.7(3), pp.203-237,1994.
[6] W.Gropp,E.Lusk and A.Skjellum, Using MPI, M.I.T Press, Cambridge, Mass., 1995.
[7] T.Ishida, Parallel, Distributed and multiagent Production Systems,Lecture Notes in Computer Science,Vol .890, Springer Verlag, New York, 1991.
[8] B.Jonsson, Compositional Specification and verification of distributed systems, ACM Trans. Programming languages and Systems, Vol.16, pp.259-303, 1994.
[9] E.V.Krishnamurthy and V.K.Murthy, Transaction Processing Systems, Prentice Hall, Sydney, 1992.
[10]D.A. Menasce and V.A..F.Almeida, Scaling for E-Business, Prentice Hall, New Jersey, 2000
[11] C. Morgan, Programming from Specification, Prentice Hall, Englewood Cliffs, New York, 1994.

[12] V.K.Murthy and E.V.Krishnamurthy, Probabilistic Parallel Programming based on multiset transformation, Future Generation Computer Systems, Vol.11, pp.283-293, 1995.

[13]V.K.Murthy and E.V.Krishnamurthy, Gamma programming paradigm and heterogeneous computing, Proc. Hawaii Intl. Conf. on System Sciences (Software Technology Track), HICSS-29, 273-281, IEEE Computer Society Press, USA, 1996.

[14] R.Orfali ,D.Harkey and J.Edwards, Client server/Survival Guide,, John Wiley,New York,1999.

[15]A.Orso, M.J.Harrold and D.Rosenblum,Component Metadata for software Engineering tasks, pp.130-144 in Engineering distributed objects, LNCS Vol.1999, Springer Verlag, New York, 2000

[16] M.Raynal, Networks and Distributed Computation, M.I.T. Press, Cambridge, Mass., 1988 .

[17]J.L.Rosenberger, Teach yourself CORBA ,Sam's Publishing Co., Indianapolis, 1998.

[18] M.Snir, S.W.Otto et al.,,MPI: The complete Reference, M.I.T.Press, Cambridge, Mass, 1996.

[19] A.W.Brown,Large Scale component-based development, Prentice Hall, New Jersey, 2000.

[20] A.W. Brown, An overview of components and component -based development, Advances in Computers, Vol. 54, pp. 1-34, Academic Press, New York, 2001.

[21] V.K. Murthy, Computing with heterogeneous components, to appear.