Journal of Programming Languages **4** (1996) 169–183

Object-based subcontracting for parallel programming on loosely-coupled distributed systems

RUSHIKESH K. JOSHI and D. JANAKI RAM

Department of Computer Science and Engineering, Indian Institute of Technology, Madras-600036, India

Received 6 February 1995

Several languages have been proposed to support concurrency in object-oriented programming. However, these languages do not address issues which are specific to parallel programming on loosely coupled distributed systems such as dynamic load variation, fault tolerance and scalability. We propose a new paradigm called object-based subcontracting for parallel programming on loosely-coupled distributed systems. In this paradigm, a metaclass called Parclass is defined to create a metaobject. The metaobject aggregates objects belonging to a class. A member function of the class, which can be executed concurrently on all the objects, is invoked as a subcontract on the metaobject. The metaobject manages the subcontract by employing various nodes of the network. The subcontract invocation is made fault tolerant and scalable.

Keywords: concurrency, object-orientation, fault-tolerance, parclass, subcontracting

1. Introduction

Networks of workstations are becoming increasingly viable platforms for parallel programming. It is an attractive proposition to combine object orientation and parallelism on these systems to claim the advantages of both. Although several languages, such as Charm++ [1], Mentat [2], and concurrency extensions to Eiffel [3, 4], have been proposed to support concurrency in object-oriented programming, in general these languages do not address issues which are specific to loosely-coupled distributed systems. The three major drawbacks of these languages when applied to loosely-coupled distributed systems are as follows:

(1) Objects are mapped to processes and are located on machines resulting in *object-to-machine-affinity*. This affinity can become a cause for load imbalances in the system.
(2) Once the object-to-process mapping is done, the grain size of the object remains fixed. Thus the maximum number of grains in the system is equal to the total number of processes. This results in additional computing power remaining idle, if it becomes available at runtime. Thus programs do not automatically scale to suit runtime situations.
(3) The state of an object resides on a single machine in full. Failure of a machine results in loss of state of the entire object. Thus, objects extensively violate the *statelessness* property which

0963-9306

is highly desired for fault tolerance on loosely coupled systems. An example of this is the file servers in distributed systems, which are made stateless to provide fault tolerance against server crashes [5].

We propose a new model for concurrency called object based subcontracting for parallel programming on loosely-coupled distributed systems. Object-based subcontracting takes the approach of creating aggregate objects belonging to a base class. This approach is similar to the Parset scheme [6] developed for procedural languages. The implementation of object-based subcontracting is based on the distributed parset kernel which was developed for implementing Parsets.

A metaclass called Parclass is defined for a base class. A metaobject, which is a Parclass object, holds together multiple objects belonging to the base class. The Parclass defines an insert operator <<, with which objects can be inserted into the metaobject. Another operator "**..**", called the subcontract operator, is defined on the Parclass. With the subcontract operator, a *subcontract* message is sent to the metaobject to invoke a member function on all the objects held under the metaobject. The metaobject makes the execution of the subcontract parallel, fault tolerant and scalable. Parallelism exists within and also outside a subcontract. When different subcontracts execute in parallel, synchronization between them is achieved by using a locking mechanism.

In the following section, we discuss the disadvantages of existing approaches to concurrency in object-oriented languages in the context of loosely-coupled distributed systems. We propose *object-based subcontracting*, a new approach for concurrency in object-oriented systems which eliminates many of the problems with existing approaches. In Section 3, we discuss the subcontracting model in detail. Subcontracting was implemented as an extension to C++ [7] on a network of SUN workstations. The implementation details are presented in the last section along with performance studies.

2. Drawbacks of the existing approaches

Traditionally, concurrency in object-oriented programming is achieved by mapping objects to processes. A process spans one or multiple objects. Languages provide implicit or explicit mechanisms for mapping objects to processes. Examples of such mappings are the *chares* of Charm++, the *mentat* classes of Mentat and the *Concurrency* classes for Eiffel [4]. This mechanism is often referred to as making an object *active* [8]. This approach does not address issues specific to parallel programming on loosely-coupled distributed systems. These issues are discussed in detail below:

Object-to-machine affinity

When objects are mapped to processes, they are in turn mapped to machines. Very often objects remain on the same machine throughout their lifetime. This is termed as *object-to-machine-affinity*. It results in dynamic load imbalances. This is especially true in the case of workstation clusters.

Programs are not scalable

At compile time, it is not possible to know in advance the number of available machines and the load on them when a program actually starts executing. The scale of parallel programming should ideally be decided at runtime. When there are a large number of lightly loaded nodes available at runtime, the parallel programs should automatically scale up. In the worst case, when there are no lightly loaded nodes available, the programs should run efficiently even on one node. This is termed *reverse scalability* of a program. This factor is important in the context of parallel programming on loosely-coupled distributed systems.

Violation of statelessness

When an object is mapped to a process, the process stores the full state of the object. This leads to violation of the *statelessness* property. If a machine on which an object is located fails, the state of that object is permanently lost necessitating a fresh run of the whole program.

It can be observed that systems that primarily define a rigid object to processor mapping suffer from these drawbacks. For example the *create* construct of Mentat maps an object to a processor, either implicitly or explicitly. Failure of this processor would result in loss of the object's state. In Charm + +, a chare is mapped to a processor achieving dynamic load balancing. However, this choice is given only once, and hence if the node becomes loaded due to other programs running on it, the newly mapped chare may suffer.

We take a different approach, called subcontracting, to address these problems in parallel programming on loosely-coupled distributed systems. In our approach, objects are not mapped to processors, but a metaobject is given the responsibility to execute a subcontract in a fault-tolerant and scalable fashion. Object-based subcontracting does not create the *object-to-machine affinity*. As a result, an object never resides permanently on a remote node. Hence programs are executed preserving the property of statelessness and a failure of a processor does not lead to the loss of the object's state. Fault tolerance is achieved with the help of an object locking scheme. Programs do not need recompilation for changing numbers of nodes or varying load patterns within a network, making them scalable. We describe our model in detail in the following section.

3. Object-based subcontracting: a concurrency model

Traditional solutions to concurrent programming in object-oriented systems provide concurrency at object level by mapping objects to processes. Subcontracting models concurrency at member function level. Concurrent invocations of a member function on multiple objects form a subcontract. Each subcontract is independent and subcontracts can be executed in parallel. A subcontract is achieved through a metaclass declaration called *Parclass*. A metaobject belonging to the Parclass is responsible to accomplish the subcontract by employing the nodes available in the network.

The declaration

Parclass ParMatrix **holds** matrix;

ParMatrix P;

creates the ParMatrix metaclass which is specialized to hold the objects of base class matrix. An instance of the metaclass ParMatrix is the metaobject P which can hold a collection of instances of the base class matrix. Objects of type matrix may be inserted in object P with the insert operator

<<. The insert operator inserts elements into a metaobject and preserves the order of the inserted elements. A subcontract for a particular method invocation on every member of P can be given with the subcontract operator. The expression,

```
P..inverse( );
```

sends a subcontract message inverse ( ) to all the objects of metaobject P.

An explicit locking scheme similar to the scheme described in the Parset scheme [6] is employed to achieve synchronization between concurrently executing subcontracts. We describe the locking scheme later in this section in detail. Figure 1 demonstrates these features with an example program.

At line 17 and 18, base objects M1 and M2, belonging to base class matrix are inserted into metaobject P. At line 19, a subcontract is given to deliver the message inverse( ) to all the elements of P. The scope of the subcontract extends till the completion of the member function execution by every base object of P, in this case, both M1 and M2.

```
1.  ....
2.  class  matrix {
3.       ....
4.  public:
5.      WO void initialize (..);
          // initializes the
          // matrix with given data
6.      RW void inverse (void);
7.      RO void print_matrix(void);
8.  };
9.  ....
10. main( ) {
11. Parclass ParMatrix holds matrix;
12. ParMatrix P;
13. matrix M1, M2;
14.     ....
15.     M1.initialize(..); // fill in the values
16.     M2.initialize(..); // fill in the values
17.     P < < M1;
18.     P < < M2;
19.     P..inverse (U);
20.     P..print_matrix (O);
21. }
```

Fig. 1. Locking specifications in subcontracting

3.1 *Object locking in subcontracting*

Objects can be locked by three types of locks called read-only (**RO**), write-only (**WO**) and read-write (**RW**) locks. When a member function in an object requires the object's state to be read and not written to, it obtains an **RO** lock on the object. Multiple **RO** locks can be existing on an object at a time. A **WO** lock can be used by a member function if it writes to the state of the object. An **RW** lock indicates that the object is required for reading as well as writing. If an object is locked with a **WO** or an **RW** lock, no other lock can be obtained on it till the lock is released. The implications of locking are described later in this section.

At line 6 in Fig. 1, a lock keyword **RW** is added before the declaration of the member function inverse( ). This indicates that the member function reads the state of the object and also modifies it. Whereas print_matrix( ) only reads the state, and initialize( ) changes the state irrespective of the earlier state of the object. Hence these two member functions are declared as **RO** and **WO**, respectively.

3.2 *Implications of locking specifications*

The locking specifications are used for different purposes at various stages of a subcontract execution. The three implications of locks are that they guide parallelism dynamically, ensure fault-tolerant behaviour and help in reducing the network communication. We describe these benefits of locks below.

Guiding parallelism dynamically

Lock specifications guide parallelism in the control flow of the program. When a subcontract is given at line 19, the system may start two parallel activities to execute inverse( ) on M1 and on M2 by appropriately migrating each object state to a remote node. When the execution of inverse( ) on M1 starts, it first locks the object M1 in read-write mode. No other member function of M1 can be executed till the lock is released. Similarly, M2 is also locked. If M1 finishes earlier, it will be unlocked and the subcontract in line 20 can start printing the matrix M1. In this manner, locks guide parallel execution of subcontracts. For example, two subcontracts which require only read access to the state of an object can be executed concurrently. It can be observed that the control flow may reach the end of program while subcontracts are still in progress. The termination of the program has to wait till all the subcontracts are executed to completion.

Providing fault tolerance to subcontracts

A metaobject secures locks which are required for a subcontract. A subcontract may involve execution on a number of processors. The required object states are migrated to remote nodes at the time of execution of a subcontract. The locks are not released till the execution is complete. In the case of failures, the unfinished part of a subcontract can be re-executed at a new location maintaining the lock status.

Minimizing network communication

The lock specifications are also used to minimize network communication at runtime. If an execution on an **RO** object is to take place at a remote site, the object state needs to be exported to a remote site (upward communication) and need not be imported at the end of execution. The state of an **RW** object has to be exported as well as imported (upward and downward communication). **WO** object has to only import its state no matter what the earlier state was (downward communication).

3.3 *Subcontract directives*

With every subcontract, a subcontract directive is also supplied to the corresponding metaobject. The directive is provided as an argument to the subcontract message. Two subcontract directives **O** and **U** are defined. The directive **U** informs the metaobject that the member function on base objects can be invoked in any order leading to concurrency (unordered execution). The directive **O** dictates an orderly execution. For example, in Fig. 1, on line 19, the subcontract inverse( ) is specified as highly concurrent, whereas on line 20, the printing function has to print the matrices one after the other in an orderly fashion.

3.4 *Object interactions in subcontracting*

In the example in Fig. 1, an *inverse( )* message is sent to multiple base objects of class matrix. Since this message does not involve arguments, there is no interaction involved between different base objects. However, in some cases, a subcontract may operate on more than one object and may also need to know the states of other objects. Such an interaction can also be captured in the subcontracting model. This is explained with an example matrix multiplication problem as shown in Fig. 2.

A matrix multiplication procedure mult( ) takes two arguments. The first is the operand matrix m2, and the second is the output matrix m3. It can be observed that in the subcontract *P..mult( )* (line 20), Q and R form additional arguments. The arguments Q and R are metaobjects. The metaobject P holds objects M1 and M2, and the metaobject Q holds objects M3 and M4. One object each from metaobjects Q and R form arguments to each invocation of the member function *mult( )* which is invoked by the subcontract on P. There can be as many activations of *mult( )* as the number of elements in P, which in this case is two. Each activation gets the corresponding arguments from Q and R. Thus the following activations of the subcontract
RO mult(**RO** matrix m2, **WO** matrix m3) can be executed concurrently:

1. **RO** M1.mult (**RO** M3, **WO** output1);
2. **RO** M2.mult (**RO** M4, **WO** output2);

A read-only access is required for m2, since multiplication needs only the state of m2 to be read. Since m3 is an output matrix, it is specified as write-only. The member function *mult( )* operating

```
...
class matrix {
   // matrix data
public:
   ....
   RO mat_data peep (void);
      // returns the actual matrix data
   WO void initialize (..);
      // initializes the matrix
      // with given data
   RO void mult (RO matrix m2, WO matrix m3);
      // multiplies m2 with this matrix by
      // calling m2.peep( ), and outputs a
      // result matrix to m3
};
...
main( ) {
Parclass ParMatrix holds matrix;
ParMatrix P,Q,R;
matrix M1, M2, M3, M4;
   ...
   M1.initialize(..); // fill in the values
   M2.initialize(..); // fill in the values
   P << M1 << M2;
   Q << M3 << M4;
   P..mult (Q, R, U); // new elements are
      // formed in R
   R..print_matrix (O);
}
```

Fig. 2. Interobject interaction

on m1 is declared as read-only. This implies that *mult()* accesses m1 only for reading. It can be noted that lock specifications are required at the level of member functions and also at the level of arguments. A lock specification for a member function states the access type for the object itself, whereas a specification at the argument level states the access requirements for objects which form arguments to the subcontract. Hence arguments are passed to functions either for read-only (equivalent to pass-by-value), read-write (equivalent to pass-by-reference), or write-only (output) operations. The output of multiplication is performed through an argument m3, which is a specified as a write-only object. The outputs of multiple activations of *mult()* are automatically collected in the metaobject R.

```
...
class  TaskGenerator {
public:
    virtual void* split(void) {return NULL;};
};
class largeImage: public TaskGenerator {
    // image data
public:
    void *split (void);
     ...
 };
 class subTask {
    // subtask data
public:
    // subcontract members
 };
 main() {
 Parclass ParImage holds subTask;
 ParImage I;
 largeImage mega_image;
     ...
     mega_image.initialize(..);
     // load the image data
     I.build (&mega_image);
     I..transform (U);
     I..print_image (O);
 }
```

Fig. 3. Supporting task division

3.5 *Support for task decomposition*

The schemes described above require that the task decomposition be done by the programmer by generating multiple base objects and inserting them into the metaobject one by one. We describe a scheme which provides support for performing task division. Figure 3 shows an example to illustrate this scheme. A Parclass defines a member function *build()* which can be invoked on a metaobject. This function is used to build the metaobject's collection from a single larger base object which divides itself into multiple smaller base objects. The function *build()* takes a pointer to an instance of class TaskGenerator. This class defines a virtual member function called *split()*. The function *build()* repetitively calls *split()* on the TaskGenerator object to obtain a number of smaller base objects till a NULL is obtained which specifies the end of task division. A larger base object, which needs to be decomposed into smaller objects, inherits the class TaskGenerator and redefines the virtual member *split()*. The function *split()* must be written in such a way that every time it is called, a pointer to a smaller base object is returned. A NULL is returned at the end of the decomposition. The function *build()* obtains a polymorphic behaviour on its argument by accepting instances of any class derived from the TaskGenerator class. In this way, the metaobject internally inserts smaller base objects into itself by making repetitive calls to *split()*.

4. Implementation

A prototype implementation of object-based subcontracting is available as an extension to the C++ programming language. The implementation works in a distributed environment consisting of a network of SUN workstations. The implementation uses the distributed parset kernel developed for implementing Parsets [6]. A program is first translated into a C++ program which makes calls to the distributed parset kernel. An overview of the implementation is shown in Fig. 4. A three node configuration is shown with the host node in the middle.

4.1 *The resident and the volatile kernel*

The distributed parset kernel is divided into two parts called the *volatile kernel* and *resident kernel*. A resident kernel resides on the machines which are ready to participate in the execution of subcontracts arriving from different machines. The resident kernel looks at the overall functionality whereas the volatile kernel is created on demand for a particular subcontract execution. The volatile kernel consists of two types of processes called *P-processes* and *E-processes*.

P-processes manage the metaobjects whereas E-processes are the actual participants which execute subcontracts. The code for P-processes is written in a completely generic fashion independent of the type of the base objects. However, an E-process code needs to be linked with the subcontract information. The subcontract information is compiled into a remote instruction block (RIB). It consists of details about the objects and the member functions involved in the subcontract. Various communication patterns between these entities are shown in the figure.

4.2 *The kernel interface*

The translated C++ program makes various calls to the kernel. Whenever a new metaobject is created by a declaration in the program, a *create* message is sent to the resident kernel. An insert operator makes an *insert* call and a subcontract operator makes a *subcontract* call to the kernel. Whenever a metaobject is destroyed in the program, a *destroy* call is made to the kernel in order to destroy the corresponding P-process.

The create and insert calls

A *create()* call is made to the resident kernel, which in turn invokes a P-process on the local node. Each P-process manages one metaobject. The call returns a P-process handle to the user program.

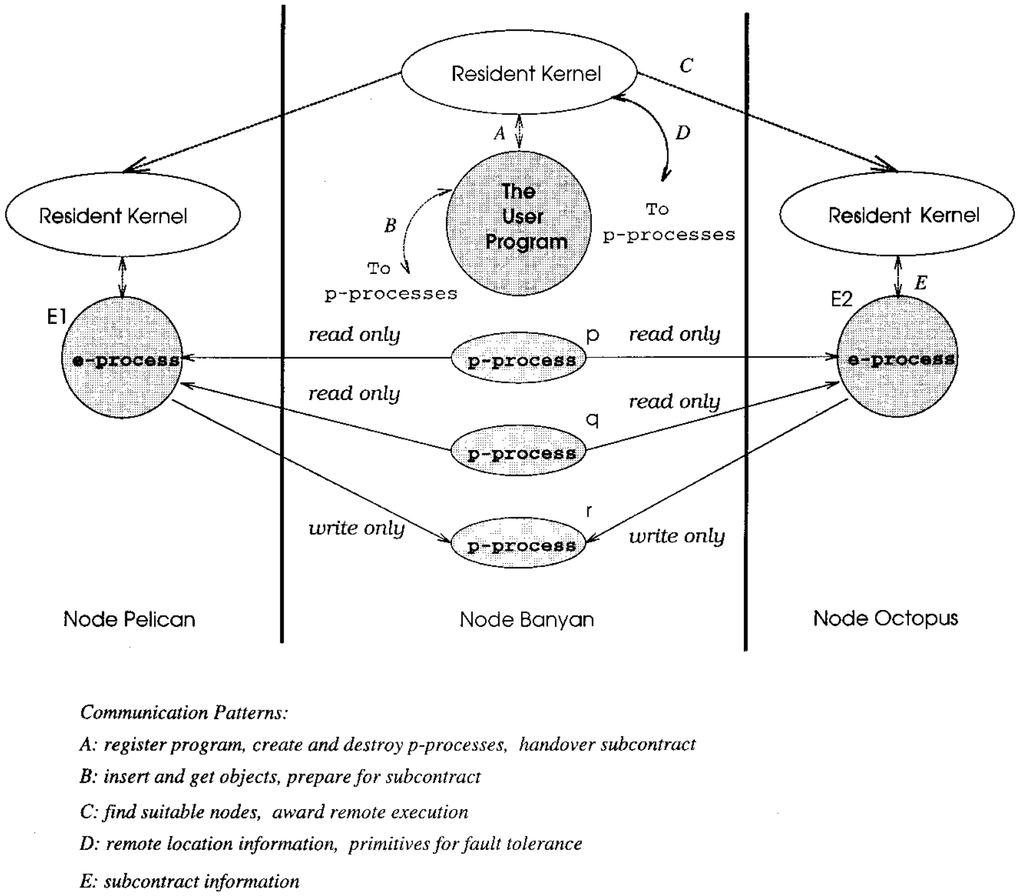

Fig. 4. The implementation of object based subcontracting

The handle is used for performing operations on metaobjects such as *insert*. Thus, calls for insertion into a metaobject do not need the involvement of the resident kernel.

The subcontract call

A subcontract operation informs the resident kernel about the metaobjects involved in the subcontract and their cardinalities. The resident kernel on the local node contacts remote nodes and E-processes are created on these nodes. The kernel chooses lightly loaded nodes for creating the remote E-processes.

A subcontract on a metaobject is further broken down into multiple grains and all grains may execute their part of the subcontract in parallel. Each grain corresponds to an object inserted in the metaobject. When a remote E-process is ready to execute a part of the subcontract, the states of the involved objects are migrated to and from the E-process as guided by the lock specifications. As seen in the figure, the communication is from P-processes to E-processes in the case of read-only P-processes p and q. Whereas, it is in the opposite direction for the write-only P-process r. While migrating an object's state to a remote node, pointers are not chased and it is required that objects manage their states in a contiguous memory space.

4.3 *Implementing fault tolerance*

A P-process does not release the locks till the metaobjects involved finish their part of the subcontract. If a failure of a node on which a subcontract is executing is detected, the subcontract is restarted on another node. This is explained further with an example. Consider the following subcontract:

RO Obj1.compute (**RO** Obj2, **WO** Obj3, **WO** Obj4)

The execution of this subcontract is via the following steps:

1. A remote E-process is created for the execution of *compute()*.
2. The four P-processes corresponding to Obj1..4 are notified of the address of the E-process.
3. The P-processes corresponding to Obj1 and Obj2 lock Obj1 and Obj2 in read-only mode and send their states to the remote E-process.
4. The P-processes corresponding to Obj3 and Obj4 lock these objects in write-only mode and wait for the results from the E-process so that the states of Obj3 and Obj4 can be refreshed.
5. The E-process executes the peer steps. It first receives the states of Obj1 and Obj2, then executes *Obj1.compute()*, and writes the output to two dummy objects. The states of these dummy objects are sent back to their respective P-processes.

It can be seen that a fault may occur during the execution of any one of the above steps. A subcontract is said to be successful only when all the P-processes which acquire write-only locks on objects receive their respective states. If any one of them fails due to a failure of the E-process, the complete sequence has to be re-executed at a new location. We have used TCP communication as a medium for interprocess communication, and we infer node failures by detecting a broken socket connection. When a broken connection is detected, the resident kernel is informed. The resident kernel finds a new location and the above steps are repeated with respect to the new E-process.

4.4 *Performance figures*

The implementation was tested for double precision matrix multiplication on a network consisting of Sun3/50 workstations. The tests were conducted under no load conditions. The details of the performance are discussed in the following sections. The program is listed in the appendix.

The scalability test

When a *subcontract* call is made by the user program, information about the number of grains involved in the subcontract is also supplied. Based on this information, the resident kernel can

Table 1. The scalability test

Number of nodes	*Speedup*
10	6.92
5	4.05
2	1.8
1	0.94

locate at the most k nodes, where k is the cardinality of a metaobject participating in the subcontract. If enough nodes are not available, grains are clubbed together into larger grains. Formation of effective grains is done at runtime and it does not require programs to be recompiled for a varying number of nodes. In worst cases, the whole subcontract can be executed on the host node itself. We define *reverse scalability* as the ability of a program to execute without recompilation using the maximum possible number of nodes down to one node. The program should not carry large overheads in satisfying reverse scalability.

Table 1 summarizes the results of the scalability test. The test program computes matrix multiplication for a size of $150*150$. The same compiled code was executed varying the number of machines from 10 down to 1 at runtime. The number of grains was fixed at 10 at compile time.

The speedup is calculated by comparing the program which uses object based subcontracting with optimized sequential code in C for the same task. It can be observed that the program which runs on 10 nodes also runs on one node without significant overheads. The test for single node indicates reverse scalability. The speedup of 0.94 on one node shows that the program designed for running on a higher number of nodes can also run on a single node without significant overheads.

Comparison with PVM

This test was conducted to observe the overheads of our implementation as compared to PVM [9], a commonly available platform for parallel programming on loosely coupled distributed systems. Table 2 shows the results of the test. The number of grains chosen was set equal to the number of available nodes for maximizing the performance. The results of the implementation show that object-based subcontracting carries a negligible overhead as compared to PVM.

5. Advantages of object-based subcontracting

Object based subcontracting is a new model of concurrency in object-oriented systems. It supports subcontracting which is based on concurrent member function activations on multiple objects belonging to a class. The subcontracts are executed by the kernel. It handles the details of migrating the subcontracts to remote machines which include data and code migration to an appropriate node.

Table 2. Comparison with PVM for double precision floating point matrix multiplication

Task size	*Number of nodes*	*Subcontracting time (s)*	*PVM time (s)*
50∗50	2	13.86	13.48
100∗100	2	100.3	97.82
	4	57.1	51.1
	5	47.18	42.34
150∗150	2	331.6	324.48
	3	228.48	219.16
	5	147.28	137.28
	6	130.32	117.14

There is no object-to-process mapping and hence object-to-machine affinity is not created. A suitable node is selected at runtime by the kernel. This alleviates the user from the burden of anticipating the runtime conditions such as load patterns and the number of nodes in the system.

A subcontract contains multiple member function invocations which can be executed in parallel. The scheduling is done internally by the kernel. Different runs of the same program may utilize a different number of nodes, thus providing scalability to the program. In extreme cases, a program can execute on a single node as a sequential program. The code need not be recompiled to handle a varying number of nodes in the system.

Object-based subcontracting is modelled on the *statelessness* property. No remote machine exclusively holds the state of an object. Hence, programs become fault-tolerant to machine failures. In the case of failures, the distributed kernel locates a new node and migrates the subcontract for re-execution.

6. Conclusions

We presented a new paradigm called *object-based subcontracting* for parallel programming on loosely coupled distributed systems. The paradigm uses metaobjects to hold together multiple base objects. A subcontract is expressed as a member function invocation on the metaobject. The metaobject handles internally the concurrent execution of the subcontract and provides scalability and fault tolerance to programs.

Acknowledgements

The authors thank Mr U. Shaji for providing an implementation for some of the ideas discussed in this document. The authors thank the anonymous referees whose comments have greatly helped in improving the quality of this manuscript.

References

1. L. V. Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++, *OOPSLA '93, ACM SIGPLAN Notices*, October (1993) 91–108.
2. A. S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat, *IEEE Computer*, **26**(5) (1993) 39–51.
3. D. Caromel. Toward a method of object oriented concurrent programming, *Communications of the ACM*, **36**(9) (1993) 90–102.
4. M. Karaorman and J. Bruno. Introducing concurrency to a sequential language, *Communications of the ACM*, **36**(9) (1993) 103–115.
5. E. Levy and A. Silberschatz. Distributed file systems, *ACM Computing Surveys*, 22 (1990) 321–374.
6. R. K. Joshi and D. Janaki Ram. Parset: a language construct for parallel programming on distributed systems, *Microprocessing and Microprogramming*, **41** (1995) 245–259.
7. B. Strustrup. *The C++ Programming Language* (Addison-Wesley Reading, MA, 1991).
8. B. B. Wyatt, K. K. Kavi and S. Hufnagel. Parellelism in object oriented languages: a survey, *IEEE Software*, November (1992) 56–65.
9. V. S. Sunderam. PVM: a framework for parallel distributed computing, *Concurrency: Practice and Experience*, December (1990) 315–339.

Appendix. The matrix multiplication program

```
// computes M3 = M1 * M2
// M1 is split to form Parclass P
// M3 is split to form Parclass R
// At the end of the execution, R holds the
// different parts of results
...
class =matrix {
    // matrix data
    const int grains = ..;
       // matrix is to be divided into grain
       // blocks, which are smaller matrices
public:
    ....
    WO initialize (..);
    RO matrix split (void);
       // returns grain blocks of this matrix
       // when called repeatedly
    RO void mult (RO matrix m2, WO matrix m3);
       // multiplies m2 with this matrix by
       // calling m2.peep( ), and
       // forms a new matrix m3.
       // Results are written to m3 by calling
       // m3.initialize( )
    RO void print_matrix ( );
};
...
main( ) {
Parclass ParMatrix holds matrix;
ParMatrix P,Q,R;
matrix M1, M2, M3;
...
    ...
    M1.initialize(..); // fill in the values
    M2.initialize(..); // fill in the values
    P.build (&M1); // P holds subblocks
    for (i=0; i < No_of_grains; i++)
       Q << M2; // read replication of matrix M2
    P..mult (Q, R, U); // R holds result blocks
    R..print_matrix (O);
}
```