Investigations in amalgamating the concept of object orientation and parallel programming have already been made for several years. The need of merging these concepts is described in [25] i.e., need of parallelism in object orientation and need of object orientation in parallelism. Many languages are developed based on with respect to what objects stand for. Objects considered as process, shared passive abstract data type or encapsulation of multiple process and data. An excellent literature survey on concurrent object oriented languages (COOLs) is done by M. Philippsen in [69, 68, 67] and N.R. Scaife in [75]. M. Philippsen categories the COOLs based on the approaches of integrating the object orientation and parallel programming models, the memory model and the way of expressing the parallelism.

It is not possible to discuss and compare all the existing parallel programming models with OP++. This section highlights the contribution made already in connection with research work of OP++. It stresses the criteria of object orientation in parallelism, casual programming style similar to C++, flexibility and modularity. The main goal and the motivation of this research is to design an object oriented parallel programming language that supports both data and task parallelism with minimal syntactic constructor to express the parallelism and with precise the semantic in terms of C++. All parallel languages that are mentioned in this section are based upon or extension to C++ programming language. The parallel programming languages that exist with extension of other object oriented
languages like Ada, Eiffel, SmallTalk, Java etc., are not considered in this discussion. As this research is interested in parallel computing in NOW on top of message passing technique, this discussion have not considered and doesn’t criticize the systems such as pC++ [32], CC++ [18], Distributed Processing in C++ (DPC++) [77] and DoPVM [38] which are based on (distributed) shared memory. These types of systems are designed and targeted more often to the tightly coupled machines and employ some complex concepts like mutex, semaphores, deadlock, and critical sections and also not well suited memory architecture for the network environment. Even though the message passing technique does not involve such complex concepts, it raises the programming complexity in explicit way of expressing the parallelism and manages the synchronization points. Therefore, this research focused on to provide an object oriented programming model based on message passing technique, semi-explicit approach of parallelism and discussed some existing programming languages based on the given approach.

Many object oriented parallel programming systems have been developed based on distributed shared memory. But very few systems were developed using message passing. Such systems can be categorized in two ways. The first one is the system which provides object oriented library functions that supports message passing like PVM++[49], CPPvm[34], Para++[21]. PVM++ and CPPvm systems provide interfaces through object orientation model for existing message passing libraries. These systems do not support object orientation in perspective of programming methodology and these systems are good at system and not at
application level. Para++ approach is different from the programming model of PVM++ and CPPvm. It allows the user to bind the object(s) for the executable version of the decomposed part(s) of the problem. Each decomposed part is written in separate C program and the executable is managed through the object called ParaProcess. It consists of member functions to spawn the executable file that bound to the ParaProcess object and to maintain the synchronization points. However, the programming model of Para++ doesn’t provide object orientation in application level.

The second one is providing object orientation in programming methodology like Dome [8, 4], Mentat [35, 54, 36], Charm++ [43, 46] and ParoC++ [55, 56] that are closely relevant to the contribution of this dissertation.

Dome is based on C++, targeted to data parallelism and it runs on top of the PVM. It provides a library of classes to express parallelism. When an object is instantiated to the tailor made classes that support template, the data inside the object is partitioned and distributed among the processing node of the PVM network that performs the same code on different data. The distribution of the data can be controlled by the programmer by using various options available in the Dome. Usually, the data distribution depends upon the number of tasks (t) and the vector size (n) of the data. Each task that is assigned to the Dome class has n/t elements of the vector. This data distribution is done at the time of object creation. When the member function of the Dome object is invoked, the invoked member function will perform its operation in each task that is placed in the separate node
in parallel. Therefore, the invoked member function will act on different data in the assigned task to provide the data parallelism. Dome supports only SPMD (single program, multiple data) programming model. There are several differences between the OP++ approach and that of Dome. OP++ system supports both SPMD and MPMD (multiple programs, multiple data) programming model often within the same program. Each process object of OP++ is bound with separate worker process. Each process object's different member function can be invoked in parallel to achieve the task parallelism. The data parallelism is also accomplished by the array of process objects. The Dome system is fully constructed of tailor made classes and exploits the parallelism through templates. And also, the problems having functional parallelism cannot be implemented in Dome.

Mentat is a noteworthy C++ based object oriented parallel programming language. The philosophy followed in OP++ programming model and Mentat looks related to each other. But, the principle and the execution strategy are totally different from one another. One of the major concepts is the mechanism of constructing dependency graph. In OP++ system, the dependency is observed based on process object and not depends upon the method invocation or the value of the parameters of the method of the process object. But in Mentat the dependency graph is constructed based on the value passed to the method and not by Mentat object. However, the object creation and method invocation, which makes the cost of a call unclear to the programmer. That is, if Mentat object is instantiated, it is not the instance itself. The new object has been created for every
method invocation on that object. If N times the method is invoked in a Mentat object, it will create N instances of that class which may be executed in parallel. In such Mentat object (instance of regular mentat class), the state of the object is not maintained. It doesn’t keep the state of the member data of the object created at the time of method invocation. It forces to write only public member functions having computational task based upon the parameter of the member function often rather than the state of the object. The state of the object can be maintained by creating the object for other two types of class declarations called persistence and sequential Mentat classes. It raises the burden to the programmer to choose the type of class declaration that leads the confusion in instance of the object and the method invocation of different types of Mentat classes.

In OP++ each process object maintains the state (if the there is any member data) of the process object. The method invocation of the process object in OP++ is very clear in both aspect of the object orientation and parallel programming paradigm in spite of message passing model. The method invocation of the process object in OP++ depends upon the existing instance where else in Mentat, the instance depends upon the method invocation which leads to misunderstanding in aspect of programming execution model and expressing the parallelism. In Mentat, definition of each Mentat class must be written in a separate file and each file must be compiled separately. Even the Mentat supports inheritance, the compilation order is restricted. The base class must be compiled before the derived class. This gives burden to the programmer to maintain the inheritance hierarchies at the time of compilation. But, in OP++
system, programmer can write the program in a single file that consists of any number of classes derived from the class *Process*.

Charm++ is an object based data driven parallel programming language. It is targeted to tightly coupled multi processor parallel machine and supports only data parallel programming model on top of message passing mechanism. It is well suited for fine grained parallelism. The parallelism is attained by the parallel object called “Chare”. Chare objects are instance of a special type of class provided by the Charm++. Chare is mapped automatically by the Charm++ run time system. Chare consists of special methods known as entry method or entry point that can be invoked in asynchronous manner. The limitations of this entry method are that the type of the parameter should be an object called message and the entry method should not return any value. Even though Charm++ is an extension of C++, many new notations and constructors are introduced for exploiting the parallelism and the data types that cast down the causality of the programming style. The programmer has to maintain multiple files for the interface and the definition of the Chare object in different files. Charm++ compromise many advantages of the object orientation. The programming style of Charm++ is difficult and it is not easy that much of OP++. In OP++, the methods of the process objects are similar to the ordinary C++. The parameter can be passed similar to C++ and users need not to marshal the parameter like message object as in Charm++. The methods of the process object also return the value and the value is received in the appropriate synchronization point. The OP++ has more causal programming style than the Charm++.
ParoC++ is a parallel object oriented parallel programming language extended to C++. The run time system of ParoC++ is designed to manage the distributed computational environment. It accomplishes the parallel execution by the entity called “parallel object”. Each parallel object resides in a separate memory address and it should not share any global recourses. The method invocation in parallel object has various semantics. The user is having additional responsibilities to mention the semantic of each method invocation that falls on synchronous invocation, asynchronous invocation, sequential invocation, mutex invocation and concurrent invocation. The method invocation of the parallel object depends upon the pre defined definition of the parallel object and not depends upon the statement (request) of the method invocation. For each semantic, separate keyword has to be used before the method signature. The programmer has to write a C++ class in ParoC++ such that he/she has to keep track the type of method invocation and to decide all the possibilities of parallelization (by means of method invocation) along with the design of the decomposable part of the problem. But in OP++ the user need not to bother or mention the semantic of the method invocation of the process object. The method having no return type (ie., void) is always invoked in asynchronous manner and the method returning the result may be invoked asynchronous or synchronous manner depends upon the dependency graph of statements that resides in the par{} block. Therefore, the semantic of the method invocation is dynamically changed according to the dependency graph constructed by the OP++ system. This reduces the burden of the programmer in designing and writing code to the decomposed parts of the problem in OP++. The parallel objects in ParoC++ are
passive objects that become active only if the request is raised for the method invocation. In this model, process and object are separate entity. All object operations may be performed by separate processes. This model has no restriction on number of processes that can be bound to the object, but the binding of the process to an object is confusing, and programming becomes complex. In OP++, process object are considered as an active object model. For each process object, a separate process will bind and perform the desired operation according to the request. The active object gives more abstract and clear programming model in distributed memory rather than the passive object model that more comfortable in the shared memory model than the NOW.

Even though the OP++ has been implemented on top of the PVM, it can be compared with each other in aspect of programming model. As mentioned earlier in the Section 2.2.2, PVM is a run time system that provides the parallel computing environment in heterogeneous collection of computers and provides libraries for C language for writing explicit parallel programs. In PVM, the programmer decomposes the problem into separate programs. Each program is written in C and compiled to run on specific types of computers in the network. The programmer has to explicitly spawn the process, decide the asynchronous and synchronous point of execution by placing explicit send and receive primitives. The data to be sent and received must marshall and unmarshall explicitly.

OP++ adapts entirely different programming model from what the PVM provides. The programmer can write the decomposed part(s) as an ordinary C++
class(es) and can achieve the parallelism. There is no explicit process creation, no explicit send and receive primitives is used. All the communications are done through the method invocation in process objects that are similar to the sequential C++ in syntactic. The users need not to handle multiple files in the programming environment. It has been noticed that, the programming size in OP++ is smaller by 40-60% of the PVM and easier than the PVM programs. This can be noticed from the given example program in OP++ (Fig.3.10). The equivalent PVM program is much larger in size. This factor may increase the performance of OP++ program in considerable amount by reducing the surplus explicit code present in the PVM programs.

One of the highlights of OP++ system is, it allows to overload the arithmetic operators like +, -, \, * for parallel arithmetic expressions, function overloading, constructor overloading and virtual functions. OP++ system doesn’t support friend functions, which access the private data of the object.